BULGARIAN ACADEMY OF SCIENCES

CYBERNETICS AND INFORMATION TECHNOLOGIES • Volume **24**, No 1
Sofia • 2024 Print ISSN: 1311-9702; Online ISSN: 1314-4081
DOI: 10.2478/cait-2024-0005

Checking Temporal Constraints of Events in EBS at Runtime

Thanh-Binh Trinh[1], *Hanh-Phuc Nguyen*[2], *Dinh-Hai Nguyen*[3], *Van-Khanh To*[3], *Ninh-Thuan Truong*[3]

[1]*Faculty of Computer Science Phenikaa University, Hanoi, Vietnam*
[2]*VMU – Vietnam Maritime University, Haiphong, Vietnam*
[3]*VNU University of Engineering and Technology, Hanoi, Vietnam*
E-mails: binh.trinhthanh@phenikaa-uni.edu.vn phucnh@vimaru.edu.vn 19020278@vnu.edu.vn khanhtv@vnu.edu.vn thuantn@vnu.edu.vn

Abstract: *As a kind of software system, the Event-Based Systems (EBS) respond to events rather than executing a predefined sequence of instructions. Events usually occur in real time, so it is crucial that they are processed in the correct order and within temporal constraints. The objective of this work is to propose an approach to check if events of EBS at runtime preserve the specification of temporal constraints. To form the approach by logic process, we have formalized the EBS model, through which, we have proved that the complexity of the checking algorithms is only polynomial. The approach has been implemented as a tool (VER) to check EBS at runtime automatically. The results of the proposed method are illustrated by checking a real-world Event Driven Architecture (EDA) application, an Intelligent transportation system.*

Keywords: *Checking tool, Event-based systems, Runtime verification, Temporal constraints.*

1. Introduction

Software verification [4] is a crucial process in software engineering that involves ensuring that a software system or component meets specified requirements and behaves as intended. The goal of verification is to identify and fix issues early in the development process, reducing the likelihood of defects and improving the overall quality and reliability of the software. Runtime verification [3, 18] is a part of the software verification. It is a computing system analysis and execution approach based on extracting information from a running system and using it to detect and possibly react to observed behaviors satisfying or violating certain properties. Some very particular properties, such as data race and deadlock freedom, are typically desired to be satisfied by all systems and may be best implemented algorithmically. Other properties can be more conveniently captured as formal specifications. Runtime verification can be used for many purposes, such as security or safety policy monitoring, debugging, testing, verification, and validation.

Event-based systems are often designed using Event-Driven Architecture (EDA), which is a style of software architecture that emphasizes the use of events and event processing. An EDA [5] can help organizations achieve a flexible system that can adapt to changes and make decisions in real-time. Real-time situational awareness means that business decisions, whether manual or automated, can be made using all of the available data that reflects the current state of software systems. Events are captured as they occur from event sources such as Internet of Things (IoT) devices, applications, and networks, allowing event producers and event consumers to share status and response information in real-time.

Apache Kafka (see [5]) is a distributed data streaming platform that is a popular event processing choice. It can handle publishing, subscribing to, storing, and processing event streams in real-time. Apache Kafka supports a range of use cases where high throughput and scalability are vital, and by minimizing the need for point-to-point integrations for data sharing in certain applications, it can reduce latency to milliseconds.

The verification of event-based systems requires specialized techniques and tools that are designed to handle the challenges of these systems. Actually, several methods have been proposed for the verification of EBS systems with temporal constraints specification. However, these methods mainly concentrate on system models by defining a language design [10], or representing EBS using Petri nets [20]; these methods are without executing the program, also known as static verification. The runtime verification of event-based systems is a challenging work. The reason is that:

- Large-scale distributed systems: Many event-based systems are large-scale distributed systems that involve multiple components running on different machines, making it difficult to coordinate and verify the behavior of the system as a whole.
- Difficulty in getting the execution time of events: execution events are considered as threads or processes in parallel programs. The execution environment may provide limited or no built-in support for measuring the execution time of events, requiring developers to implement their own timing mechanisms.

In this paper, we propose an approach to check the execution of events in the system if it satisfies their specification (runtime verification). The specification here includes the temporal constraints, precisely, the relationship on the occurrence of events in the system. In this approach, we get log files of the start times and end times of events when the system executes. We then give algorithms to check that the execution of events satisfies the temporal constraints in the specification. The implementation in a support tool helps us to check the events-based systems in an automatic manner.

The rest of the paper is organized as follows. Section 2 presents foundational knowledge of EDA and briefly introduces some related works. Our main approach for checking temporal constraints of events in EBS at runtime is presented in Section 3. To illustrate the approach, we have implemented a support tool and give a case study in Section 4. We conclude the paper and give some directions for future works in Section 5.

2. Background and related works

Event-Driven Architecture (EDA) [5] is an approach to software design and architecture that emphasizes the production, detection, and consumption of events. Events are meaningful occurrences or changes in a system, such as a user action, a sensor reading, or a database update. In an EDA, events are used to trigger and coordinate the communication between various components or services within a system.

EDA provides a loosely coupled, scalable, and resilient architecture that can handle large amounts of data and processing. It is often used in modern applications such as microservices, IoT, and cloud computing, where scalability and responsiveness are essential. EDA allows services to be decoupled from each other, making them easier to maintain and update, as well as enabling the creation of complex workflows and business processes.

In EDA, components can act as event producers, event consumers, or both. Event producers generate events and publish them to a message broker or event bus, while event consumers subscribe to events and receive them as they occur. This decoupling of components allows for asynchronous communication and processing, which can improve performance and scalability. The source of an event can be from internal or external inputs. Events can be generated from a user, like a mouse click or keystroke, an external source, such as a sensor output, or come from the system, like loading a program.

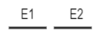
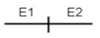
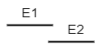
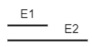
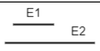
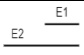
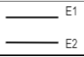
Timing and order of events can be relevant for the processing of events and are of special interest in EDA. Timing constraints are requirements on the timing characteristics of a software design, usually expressed in terms of maximum or minimum values for delays, clock frequencies, setup times, hold times, etc. Timing constraints are used to ensure that the design meets its performance requirements and operates correctly.

In event processing applications, it is common to have temporal constraints on events. Temporal constraints are constraints on the temporal relationship between events in a system. They specify when events must occur in relation to each other, such as "*event A must occur before event B*" or "*event C must occur within 100 milliseconds of event D*". Temporal constraints are used to ensure that the system behaves correctly and meets its timing requirements.

An event is defined as the state change of one or more entities over a period of time. Events occur over intervals of time and are correlated by their temporal relationships. According to Allen's axiomatization of time periods [13], there are thirteen atomic relations *b, bi, m, mi, o, oi, s, si, d, di, f, fi, eq* that can hold between two events, and they, respectively, represent as *before, meets, overlaps, starts, during, finishes, equal,* and their inverses, as shown in Table 1.

There are many methods of studying how to integrate Allen's thirteen atomic interval temporal relations into software systems. However, these methods often check event-based systems with temporal constraints that are usually done at a static level [10, 20].

Table 1. Allen's thirteen atomic interval temporal relations to represent the temporal relations between two events E_1 and E_2

Relations	Symbol	Inverse	Pictorial meaning
E_1 before E_2	b	bi	
E_1 meet E_2	m	mi	
E_1 overlaps E_2	o	oi	
E_1 starts E_2	s	si	
E_1 during E_2	d	di	
E_1 finishes E_2	f	fi	
E_1 equal E_2	eq	eq	

In article [1], the authors discuss several extensions to the Event-Condition-Action (ECA) rule paradigm to support advanced applications, including alternative actions for capturing security violations, generalization of events and rules for modeling a wider range of applications, and event detection modes for capturing complex policies or situations. They also propose an extension of rule detection modes to ensure the correct enforcement of specified rules and discuss the extensions made to event detection graphs to implement these extensions. Overall, the focus of the paper is on using the access control domain to drive the extensions needed for expressiveness, specification, and execution of policies using the ECA paradigm. The authors cover alternative actions, generalized event specification and detection, event detection modes, and extensions to event detection graphs.

Article [19] discusses the linking of security policy to event-based systems, which allows for formal reasoning about information security. The applications addressed in the article involve highly confidential data that must be shared dynamically and for historical analysis. In these applications, principals with rights to access the data may be widely distributed across a federation of independent administrative domains. Domain managers are responsible for the data held within domains and transmitted from them, and security policy must be specified and enforced to meet these obligations. The article uses healthcare as a running example because the confidentiality of healthcare data must be guaranteed over many years. The authors first consider how to enforce authorization policy at the client level through parametrized Role-Based Access Control (RBAC) [16], taking context into account. They then discuss the additional requirements for secure information flow through the infrastructure components that contribute to communication within and between distributed domains. Finally, the article shows how this approach supports reasoning about event security in large-scale distributed systems.

Paper [11] discusses the issue of event inference in event-based systems. While some events are generated externally and flow across distributed systems, others must be inferred by the system itself. The challenge is to balance the need for inferring events with certainty using complete information and providing quick notification of newly revealed events. However, the actual occurrences of events may not match the ability of event-based systems to accurately infer them due to uncertainty stemming from unreliable data sources, networks, fuzzy terminology, or the inability to determine with certainty whether a phenomenon has occurred. The article presents the state-of-the-art in event processing over uncertain data, including a classification of uncertainty, a model for event processing over uncertain data, algorithmic solutions for handling uncertainty, and a simple pattern language that supports uncertainty. The article also highlights open issues and challenges in this research area.

In the article [14], the authors present an Event-Driven cloud Architecture (REDA) that helps process a large volume of real-time application data generated by connected devices in the Internet of Things (IoT) network. REDA utilizes technologies such as AWS IoT [6], MQTT [8], Apache Kafka, and Java Spring to build an event-based data processing environment that is capable of efficiently and cost-effectively processing real-time data. The article [9] discusses the use of Event-Driven Architectures (EDA) for traffic management systems, which need to handle a large volume of events generated by sensors. Traditional software architectures are not optimized for the efficient processing of continuous event streams, making EDA a new paradigm for event-based applications. The authors propose a reference architecture for event-driven traffic management systems that allow for the analysis and processing of complex event streams in real time. This approach is particularly useful for decision support in sensor-based traffic control systems. The authors illustrate their proposal with a case study in the domain of road traffic management. Specifically, they report on the redesign of An Intelligent Transportation Management System (ITMS) prototype for the high-capacity road network in Bilbao, Spain.

In the area of runtime verification, there are several visual tools such as Tracealyser [17], TuningFork [4], WindView [12], Vampir [15], Zinsight [21], and TraceCompass [7] that have been deployed to display traces and provide insights into the behavior of programs. Most of these tools focus on presenting traces in a timeline view, which allows users to follow the flow of a program and understand its behavior over time. This timeline view is convenient for understanding individual program executions. However, none of these tools, to the best of our knowledge, extend or leverage timing analysis in event based systems.

Because event-based systems rely heavily on the ordering and timing of events, these systems are designed to respond to events as they occur in real-time, so it is crucial that the events are processed in the correct order and within the specified time constraints. Therefore, it really makes sense to check temporal constraints at runtime of event based systems. If an event is processed out of order or if it takes too long to process, it can cause significant problems for the system. Imagine an event-based system that processes online orders for a retail store. If the system processes the "payment" event before the "add to cart" event, the order will fail because the system does not have a record of the items the customer wants to purchase. Similarly, if the system takes too long to process the "checkout" event, the customer may become frustrated and abandon the purchase. However, the traditional verification methods are not considered for verifying the occurrence period of events in EDA at runtime and may lead to overlooking errors that occur during execution.

3. Model formalization and checking approach

In this section, we present the approach to check the temporal constraints of events in event-based systems. To form the approach by logic process, first of all, we will formalize the model of event systems and its elements, we will then provide the runtime verification method using checking algorithms.

3.1. Model formalization

In practice, there are several architectures for an Event Based System (EBS). However, they all adhere to the same operating principles. In order to easily discuss the model and its elements participating in the verification process, we provide formal definitions of elements in EBS and their operations.

Definition 1 (Event based system). An Event Based System is a tuple $\langle E, P, S, B, \mathrm{SP}\rangle$ which composed of a set of Events *E*, a set of Publishers *P*, a set of Subscribers *S*, a Broker *B* which maps events from publishers *P* to subscribers *S*, and the SPecification of the system SP.

Events are crucial elements in EBS, as they enable real-time communication, coordination, and responsiveness. By leveraging the power of events, EBS can provide timely and accurate information, automate processes, and improve overall system performance and scalability. We define events as the Definition 2.

Definition 2 (Event). An event in EBS is a tuple $\langle \mathrm{en}, a, T\rangle$, denoted by $\mathrm{evt} = \langle \mathrm{en}, a, T\rangle$ where:

- en is the event name
- a is the set of actions
- $T \in [\mathrm{startTime}, \mathrm{endTime}]$

Causal relationship between events refers to the relationship between cause-and-effect, where one event is the cause of another event. In this context, the causality relationship specifies that one event must happen with another event in temporal constraints and that the occurrence of the first event is what triggers the second event.

Definition 3 (Causal relationship). Two events evt_i and evt_j are said to have a causal relationship, denote by $\mathrm{cr} = \langle \mathrm{evt}_i, \mathrm{evt}_j\rangle$ or $\mathrm{cr}\langle \mathrm{evt}_i, \mathrm{evt}_j\rangle$ if and only if evt_j occurs as a result of evt_i (activation, influence) or vice versa.

From the causal relationship between two events in Definition 3, we define the Allen's thirteen atomic interval temporal relations in an EBS as the following definitions.

We say that an event occurs before another one if an instance of the first event ends, then the second event is triggered, we formalize this relation as in Definition 4.

Definition 4 (Before relation). An event evt_i has a before relation with event evt_j, denote by $\mathrm{before}(\mathrm{evt}_i, \mathrm{evt}_j)$ iff:

- $\mathrm{cr}\langle \mathrm{evt}_i, \mathrm{evt}_j\rangle$
- $\tau_i \in [\tau_i.\mathrm{startTime}, \tau_i.\mathrm{endTime}] \wedge \mathrm{validTimeInterval}(\tau_i)$

- $\tau_j \in [\tau_j.\mathrm{startTime}, \tau_j.\mathrm{endTime}] \wedge \mathrm{validTimeInterval}(\tau_j)$
- $\tau_i.\mathrm{endTime} \leq \tau_j.\mathrm{startTime}$
- $\mathrm{validTimeInterval}(\tau) := \tau \in [\tau.\mathrm{startTime}, \tau.\mathrm{endTime}] \wedge \tau.\mathrm{startTime} < \tau.\mathrm{endTime}$

Two events are met each other when the interval of the first ends exactly when the interval of the second event starts, we formalize this relation as in Definition 5.

Definition 5 (Meet relation). Two events evt_i and evt_j meet each other, denoted by $\mathrm{meet}(\mathrm{evt}_i, \mathrm{evt}_j)$ iff:

- $\mathrm{cr}\langle \mathrm{evt}_i, \mathrm{evt}_j \rangle$
- $\tau_i \in [\tau_i.\mathrm{startTime}, \tau_i.\mathrm{endTime}] \wedge \mathrm{validTimeInterval}(\tau_i)$
- $\tau_j \in [\tau_j.\mathrm{startTime}, \tau_j.\mathrm{endTime}] \wedge \mathrm{validTimeInterval}(\tau_j)$
- $\tau_i.\mathrm{endTime} \equiv \tau_j.\mathrm{startTime}$
- $\mathrm{validTimeInterval}(\tau) := \tau \in [\tau.\mathrm{startTime}, \tau.\mathrm{endTime}] \wedge \tau.\mathrm{startTime} < \tau.\mathrm{endTime}$

Two events overlap each other if the interval of the first has not ended meanwhile the interval of the second event is triggered. We formalize this relation as in Definition 6.

Definition 6 (Overlaps relation). Event evt_i is said to have an overlap relation with event evt_j, denoted by $\mathrm{overlap}(\mathrm{evt}_i, \mathrm{evt}_j)$ iff:

- $\mathrm{cr}\langle \mathrm{evt}_i, \mathrm{evt}_j \rangle$
- $\tau_i \in [\tau_i.\mathrm{startTime}, \tau_i.\mathrm{endTime}] \wedge \mathrm{validTimeInterval}(\tau_i)$
- $\tau_j \in [\tau_j.\mathrm{startTime}, \tau_j.\mathrm{endTime}] \wedge \mathrm{validTimeInterval}(\tau_j)$
- $((\tau_i.\mathrm{startTime} > \tau_j.\mathrm{startTime}) \wedge (\tau_i.\mathrm{endTime} > \tau_j.\mathrm{endTime})) \vee ((\tau_i.\mathrm{startTime} < \tau_j.\mathrm{startTime}) \wedge (\tau_i.\mathrm{endTime} < \tau_j.\mathrm{endTime}))$
- $\mathrm{validTimeInterval}(\tau) := \tau \in [\tau.\mathrm{startTime}, \tau.\mathrm{endTime}] \wedge \tau.\mathrm{startTime} < \tau.\mathrm{endTime}$

An event starts another if an instance of the first event starts at the same time as an instance of the second event, but ends earlier. We formalize this relation as in Definition 7.

Definition 7 (Starts relation). Event evt_i is said to have a start relation with event evt_j, denoted by $\mathrm{start}(\mathrm{evt}_i, \mathrm{evt}_j)$ iff:

- $\mathrm{cr}\langle \mathrm{evt}_i, \mathrm{evt}_j \rangle$
- $\tau_i \in [\tau_i.\mathrm{startTime}, \tau_i.\mathrm{endTime}] \wedge \mathrm{validTimeInterval}(\tau_i)$
- $\tau_j \in [\tau_j.\mathrm{startTime}, \tau_j.\mathrm{endTime}] \wedge \mathrm{validTimeInterval}(\tau_j)$
- $(\tau_i.\mathrm{startTime} \equiv \tau_j.\mathrm{startTime}) \wedge (\tau_i.\mathrm{endTime} \leq \tau_j.\mathrm{endTime})$
- $\mathrm{validTimeInterval}(\tau) := \tau \in [\tau.\mathrm{startTime}, \tau.\mathrm{endTime}] \wedge \tau.\mathrm{startTime} < \tau.\mathrm{endTime}$

An event occurs during another event if the interval of the first is contained in the interval of the second, we define this relation as in Definition 8.

Definition 8 (During relation). Event evt_i is said to have a during relation with event evt_j, denoted by $\mathrm{during}(\mathrm{evt}_i, \mathrm{evt}_j)$ iff:

- $\mathrm{cr}\langle \mathrm{evt}_i, \mathrm{evt}_j \rangle$

- $\tau_i \in [\tau_i.\text{startTime}, \tau_i.\text{endTime}] \wedge \text{validTimeInterval}(\tau_i)$
- $\tau_j \in [\tau_j.\text{startTime}, \tau_j.\text{endTime}] \wedge \text{validTimeInterval}(\tau_j)$
- $(\tau_i.\text{startTime} > \tau_j.\text{startTime}) \wedge (\tau_i.\text{endTime} < \tau_j.\text{endTime})$
- $\text{validTimeInterval}(\tau) := \tau \in [\tau.\text{startTime}, \tau.\text{endTime}] \wedge \tau.\text{startTime} < \tau.\text{endTime}$

One event finishes another one if an occurrence of the first ends at the same time as an occurrence of the second event, but starts later. We formalize this relation as in Definition 9.

Definition 9 (Finishes relation). Event evt_i is said to have a finish relation with event evt_j, denoted by $\text{finish}(\text{evt}_i, \text{evt}_j)$ iff:

- $\text{cr}\langle \text{evt}_i, \text{evt}_j \rangle$
- $\tau_i \in [\tau_i.\text{startTime}, \tau_i.\text{endTime}] \wedge \text{validTimeInterval}(\tau_i)$
- $\tau_j \in [\tau_j.\text{startTime}, \tau_j.\text{endTime}] \wedge \text{validTimeInterval}(\tau_j)$
- $(\tau_i.\text{startTime} > \tau_j.\text{startTime}) \wedge (\tau_i.\text{endTime} \equiv \tau_j.\text{endTime})$
- $\text{validTimeInterval}(\tau) := \tau \in [\tau.\text{startTime}, \tau.\text{endTime}] \wedge \tau.\text{startTime} < \tau.\text{endTime}$

Two events are equal if they happen right at the same time, they start and end the execution together, we formalize the equal relation as in Definition 10.

Definition 10 (Equal relation). Event evt_i is said to have an equal relation with event evt_j, denoted by $\text{equal}(\text{evt}_i, \text{evt}_j)$ iff:

- $\text{cr}\langle \text{evt}_i, \text{evt}_j \rangle$
- $\tau_i \in [\tau_i.\text{startTime}, \tau_i.\text{endTime}] \wedge \text{validTimeInterval}(\tau_i)$
- $\tau_j \in [\tau_j.\text{startTime}, \tau_j.\text{endTime}] \wedge \text{validTimeInterval}(\tau_j)$
- $(\tau_i.\text{startTime} \equiv \tau_j.\text{startTime}) \wedge (\tau_i.\text{endTime} \equiv \tau_j.\text{endTime})$
- $\text{validTimeInterval}(\tau) := \tau \in [\tau.\text{startTime}, \tau.\text{endTime}] \wedge \tau.\text{startTime} < \tau.\text{endTime}$

The soundness of an event-based system refers to its ability to ensure that all events in the system are valid and consistent with the system's rules and constraints. In other words, a sound event-based system will ensure that events are processed correctly and that the system remains in a valid state at all times. In this paper, we consider the soundness of EBS in the meaning that, the system must preserve temporal constraints specification of events.

Definition 11 (Soundness of event-based system). An event-based system EBS is sound at runtime if and only if with two events evt_i, evt_j which have causal relationship $\text{cr}\langle \text{evt}_i, \text{evt}_j \rangle$ and constrained in the specification SP, $\text{SP} := \text{before}(\text{evt}_i, \text{evt}_\text{j}) \vee \text{meet}(\text{evt}_i, \text{evt}_j) \vee \text{start}(\text{evt}_i, \text{evt}_\text{j}) \vee \text{overlap}(\text{evt}_i, \text{evt}_j) \vee \text{finish}(\text{evt}_i, \text{evt}_j) \vee \text{during}(\text{evt}_i, \text{evt}_j) \vee \text{equal}(\text{evt}_i, \text{evt}_j)$, then all executed events evt_i, evt_j must hold SP.

Proposition 1 (Checking the soundness of EBS). The correctness of an EBS can be checked by demonstrating that all pairs of events $(\text{evt}_i, \text{evt}_j)$ occurring before a time point *t* (*t* < current_time) must hold SP.

Proof. We know that programs are typically designed to be deterministic, meaning that given the same inputs and initial state, they will produce the same outputs. Determinism ensures that the program's behavior remains consistent regardless of when it is executed. In addition, consistency is facilitated by maintaining a stable execution environment. This includes using a stable operating system, hardware, and software dependencies. Changes in the environment can introduce variability and potentially impact the behavior of the program.

As the determinism of the software programs and suppose that an EBS system has a stable environment, instead of proving that EBS is soundness at all times, we just need to prove that all events (evt_i, evt_j) taking place before a time point t (t < current_time), they must hold SP.

With Proposition 1, if we detect the violation of the events executed in EBS with its specification, then we conclude that the EBS is unsound, but if we have not detected the violation yet, we cannot conclude anything about the soundness of the EBS. Proposition 1 is the basis on which we build a tool to check automatically the soundness of an EBS because the data in the log file that records the events' execution can be used for checking against temporal constraints specification.

Proposition 2 (Solvable algorithm). An event-based system can be checked for soundness at runtime in polynomial time.

Proof. Suppose that, we have m constraints declared in SP, the log file of the EBS has n tuples $(\mathrm{event}R_k,\ \mathrm{startTime}_k,\ \mathrm{endTime}_k),\ k\ =\ 1,\dots,\ n$. The checking program performs the following operations to check the soundness of EBS:

For each pair (eventS_i, eventS_j) in SP do: Check if $\exists(\mathrm{eventR}_i$, $\mathrm{eventR}_j)$ hold the specification $(\mathrm{eventS}_i, \mathrm{eventS}_j), \forall i,j \mid i \leq n, j\ \leq n.$ /*this requires at most n^2 comparisons operations */

Thus, we can conclude that the time complexity of the checking program is polynomial which is proportional to the square of the log file size and the number of temporal constraints in the specification ($O(m \times n^2)$).

3.2. Approach to checking EBS at runtime

From the analysis of the model formalization of EBS, we propose an approach to verify the soundness of an event-based system concerning its specification at runtime.

The verification process (Fig. 1) can be described that, with an event-based system running, we get the properties of executed events (event name, start time, end time, etc.) of EBS into a log file. Data preprocessing uses techniques to prepare raw data for establishing a table with three columns (event name, start time, and end time) before writing it into a log file. From the data provided in the log file and specification file of the EBS system, we implement algorithms to check the soundness of EBS (see Proposition 1, Proposition 2).

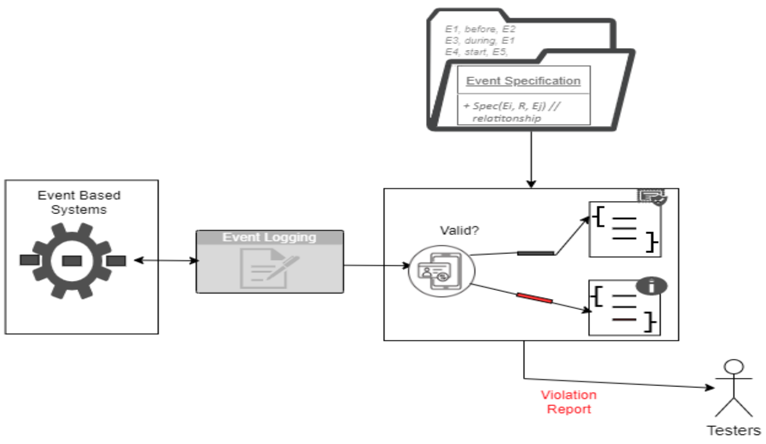


Fig. 1. Checking the soundness of Event Based Systems

```
**Algorithm 1.** Checking temporal relation before($\mathrm{evt}_i$, $\mathrm{evt}_j$)
*Input:* Log data of events, specification before($\mathrm{evt}_i$, $\mathrm{evt}_j$)
*Ouput:* Check if all events $\mathrm{evt}_i$, $\mathrm{evt}_j$ hold the specification
Extract all events type $\mathrm{evt}_i$, into a set $\mathrm{SE}_i$
Extract all events type $\mathrm{evt}_j$ into a set $\mathrm{SE}_j$
for each $\mathrm{evt}_j \in \mathrm{SE}_j$ do
    if $\nexists \mathrm{evt}_i \in \mathrm{SE}_i$ such that causality($\mathrm{evt}_i$, $\mathrm{evt}_j$) then
        return FALSE
    else
     if$(\mathrm{evt}_i.\mathrm{endTime} = \mathrm{UNDEFINED}) \vee (\mathrm{evt}_i.\mathrm{endTime} \geq \mathrm{evt}_j.\mathrm{startTime})$ then
            return FALSE
     end
    end
end
return TRUE
```

We may consider an example of the algorithm to check the before relationship between two events evt_i, evt_j. In the normal case, if the first event will end before the occurrence of the second event, we log data and verify all the conditions of before relation in Definition 4. In an exceptional case, suppose that, the log file of the system is get before the moment t, event evt_i is still running, we cannot determine when evt_i will end, and when the event evt_j will occur. Then, if the event evt_i is logged, it has already started but it has not ended yet, the end time of the event, in this case, is interpreted by endTime = UNDEFINED. The algorithm checking the relation before between two events evt_i, evt_j presented in Algorithm 1, is of time complexity $O(n^2)$.

Note that, when building algorithms, for each pair of events (evt_i, evt_j) which needs to check the temporal relation, we have defined a boolean function causality(evt_i, evt_j) which aims at checking whether two events evt_i and evt_j have the relation of causality (see Definition 3).

Similarly, we can build other algorithms to check the remaining temporal constraints in Allen’s thirteen atomic interval temporal relations.

4. A case study and a support tool

In this section, we present a case study of Intelligent Transportation System that is deployed using Event Driven Architecture. We also introduce a support tool VER of the approach proposed in the Section 3 and illustrate how the tool VER can check the temporal constraints of events in the case study.

4.1. Intelligent transportation system

The Intelligent Transportation System (ITS) focusses on the aspects of Traffic Control. Various architecture systems have been implemented depending on the project’s requirements and the features needed in the system. However, the architecture for Traffic Control Systems in this paper is deployed on the Event Driven Architecture. This architecture allows components of the system (such as sensors, mobile applications, servers, etc.) to send and receive traffic-related events such as traffic flow information, road conditions, accidents, and traffic jams. The components of the ITS architecture (Fig. 2) can be summarized as follows.

- Broker. It is the intermediate component between publishers and subscribers. The broker receives events from publishers and forwards them to registered subscribers. The broker can also perform tasks such as filtering and classifying events before sending them to subscribers. Examples of popular broker technologies are Apache Kafka, RabbitMQ, or AWS SNS.
- Publishers. These are components that create events and send them to pre-registered channels (topics). In a road traffic monitoring system, publishers can be sensors placed on the streets, mobile applications that allow users to report traffic conditions, or intelligent license plate reading systems. The events that publishers create can include information about road conditions, traffic flow, travel speed, etc.
- Subscribers. These are components that subscribe to receive events from corresponding channels. In a road traffic monitoring system, subscribers can be mobile applications that display traffic information to users, monitoring systems to detect traffic accidents and issue warnings, or fleet management systems to update traffic conditions. Subscribers can subscribe to receive events from multiple channels to handle more complex situations.
- Channels. These are the channels through which events are sent. In a traffic monitoring system, channels can be classified by road conditions, geographic location, vehicle type, or by agents causing traffic situations such as accidents or congestion. Channels help to classify events and deliver them to the corresponding subscribers for processing.

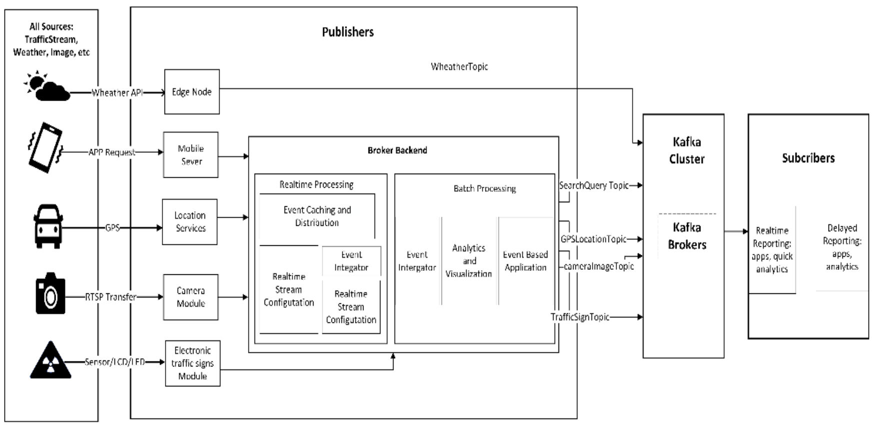


Fig. 2. Overview architecture of the Intelligent Transportation System

Besides, in an intelligent road traffic monitoring system, depending on the system's requirements and scale, there can be a variety of different events such as *Road accidents occur, Vehicles exceed the speed limit on the road, Vehicles run on red lights*, etc. We describe certain temporal relationships between events of the ITS as follows.

- When a train runs in a residential area, the warning bell system will ring. To achieve this purpose, the system must install sensors on the rails near the residential area. When the sensors detect a train starting to enter the residential area, these sensors will activate the warning bell system located in that specific area, so that other vehicles know not to enter during this time. Therefore, the event of the warning bell ringing must know the cause of the train running on which railway and in which area it caused the warning.
- The event of the traffic light turning red occurs right before the event of the traffic light turning green. At intersections, traffic light systems are installed. The lights have three colors: green, yellow, and red, which constantly change states. Vehicles must stop when they encounter a red light and are allowed to move when the light changes to green.
- The event of detecting a vehicle running a red light triggers the event of sending a notification to the system: Sensors and surveillance cameras are installed at intersections to detect vehicles running red lights, and immediately send information to the monitoring system for processing. The detection and notification events occur simultaneously and carry information about each other to determine the location and cause of the event.
- Event of card swiping at tollbooths occurs before the event of barrier gate opening: The driver swipes the card at the tollbooth, and the system uses sensors or license plate recognition systems to identify the vehicle approaching the tollbooth. After the system successfully verifies the payment, the barrier gate will automatically open for the vehicle to pass through. Each card-swiping event can only have one barrier gate opening event, and they have information about each other.

The description of temporal relations between events of ITS can be summarized in the Table 2.

Table 2. The temporal relationship between events

Symbol	Event name	Relation
E_1	Train running on the track	E_3 during E_1
E_2	Train running in residential area	E_2 equal E_3
E_3	Bell ringing	E_3 during E_1
E_4	Red traffic light	E_4 meet E_5
E_5	Green traffic light	E_5 start E_1
E_6	Vehicle running a red light	E_6 start E_7
E_7	Send a notification to the system	E_6 start E_7
E_8	Scan license plate	E_8 before E_9
E_9	Door open	E_8 before E_9

4.2. A support tool

We have used Java programming language to build a support tool, named VER (The source code of the tool can be found at **https://github.com/Hai0612/VER**). VER helps system developers to analyze automatically and resolve errors that have occurred during event-based system execution, locate the position of errors, and provide solutions for fixing them. The overview architecture of VER is presented in Fig. 1.

As we can see in Fig. 1, the core of VER is the verification module which analyzes the events and contains algorithms to check the soundness of the EBS. The inputs of the tool are the logging file and temporal constraints specification. The log file gets the properties of executed events (event name, start time, end time, etc.). The system's specification constrains temporal relations between events. If the system is sound then there is no violation reported in the output screen. But in case the tool detects the unsound of the system, the tool will point out report violations and indicate the reason.

Applying the case study of ITS with the VER tool to the experimental results shown in Table 3, we can see that, the events E_2 and E_3 have an "equal" relationship in the specification but there exists an occurrence of event E_2 with an end time not equal to the one of event E_3, then the VER tool must display a violation report. Similarly, the "start" relationship between events E_6 and E_7 is described in the specification but the start time of event E_6 and event E_7 is different. Then the notification of violation reports of VER is shown as in Fig 3.

Table 3. Experimental results

Symbol	Event name	Start, ms	End, ms	Relation (specification)	Verification result
E_1	Train running on the track	167975649114	167975650733	E_3 during E_1	soundness
E_2	Train running in residential area	168079590482	168085653451	E_2 equal E_3	not sound
E_3	Bell ringing	168079590483	169875650228	E_3 during E_1	soundness
E_4	Red traffic light	167975652337	167975652841	E_4 meet E_5	soundness
E_5	Green traffic light	167975652841	167975653342	E_5 start E_1	not sound
E_6	Vehicle running a red light	168079589874	169075665008	E_6 start E_7	not sound
E_7	Send a notification to the system	168079590380	167975665509	E_6 start E_7	soundness
E_8	Scan license plate	167975652340	167975652841	E_8 before E_9	soundness
E_9	Door open	167975653043	167975653343	E_8 before E_9	soundness

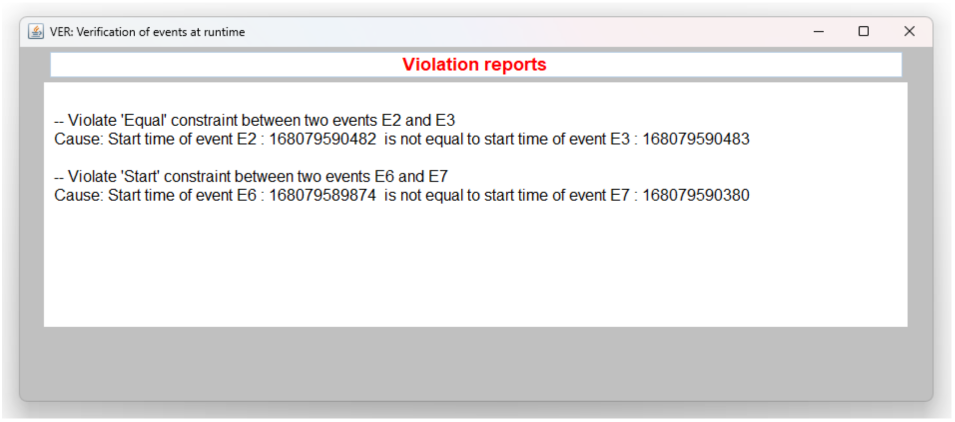
4.3. Analysis of results

The result of the VER tool determines whether the software system is currently running in accordance with its specifications. In cases where the software deviates from the specified requirements, the tool will identify the violations in the events that do not adhere to the specified constraints. This tool serves as a means for system developers to analyze and address errors that occur during program execution, pinpoint the location of errors, and provide solutions for correction.

The VER tool is built on algorithms that analyze information from events, their timing, and the sequence of their occurrences. Simultaneously, these algorithms are designed to detect constraint violations between events in real time, helping minimize damage in case of incidents.

We propose implementing an event-driven system as a smart traffic monitoring system. It provides a flexible testing framework that allows users to test events in various situations. This system enables recording and analyzing traffic-related events such as traffic flow status, traffic light conditions, traffic accidents, traffic law violations, and many other events. This system is not just a simulation program to build a verification tool; it also holds significant practical value.

With this tool, developers can easily test and monitor events happening in the system, helping them detect and rectify errors as well as improve the performance and reliability of the application. Through testing on a simulated system, the verification results are confirmed to be accurate and comprehensive. The verification results will be evaluated and reported back to the system to help them better understand the accuracy of events in the system.



```
-- Violate 'Equal' constraint between two events E2 and E3
Cause: Start time of event E2 : 168079590482  is not equal to start time of event E3 : 168079590483

-- Violate 'Start' constraint between two events E6 and E7
Cause: Start time of event E6 : 168079589874  is not equal to start time of event E7 : 168079590380
```

Fig. 3. Violation reports of the support tool

Despite the positive results achieved above, the verification tool also has some limitations and challenges. Due to the system’s use of the publish-subscribe mechanism, handling and distributing events become challenging when the number of events increases dramatically, leading to congestion in the system. This requires enhancing processing capabilities and re-evaluating event processing algorithms in the system.

5. Conclusion and future works

In conclusion, this paper presented an approach for checking temporal constraints of events in EBS at runtime. The properties of software systems should be analyzed and checked to ensure that the running system is in compliance with its requirements, including temporal constraints of events in EBS. The proposed approach uses the data in the log file of event occurrence periods and the specification of temporal constraints as inputs for the checking process. A series of algorithms are provided to check the satisfaction of temporal constraints. The approach has been implemented as a support tool that has been applied to a real-world EDA application, an intelligent transportation system. The experimental results demonstrate its effectiveness in detecting violations of temporal constraints.

However, like the software testing technique, we say that the system is not sound if the checking approach detects violations of the running system with its specification, we cannot ensure the soundness of the events system if we have not found the violations.

Future work could explore the integration of the proposed approach with other runtime verification and monitoring techniques, such as model-based testing and runtime monitoring, to enhance the reliability and robustness of the system. Additionally, the scalability and performance of the approach could be further evaluated on larger and more complex systems; we are also improving algorithms in the VER tool to work with the big log file. Overall, the proposed approach and its tool offer a promising solution for ensuring the correctness of event-based systems with other properties at runtime.

***Acknowledgement*:** This research has been done under the research project QG.21.29 "Approaches in Security Verification and Testing of Software Systems" of Vietnam National University, Hanoi.

References

1. Adaikkalavan, R., S. Chakravarthy. Generalization of Events and Rules to Support Advanced Applications. – In: S. Helmer, A. Poulovassilis, F. Xhafa, Eds. Reasoning in Event-Based Distributed Systems. Berlin, Heidelberg, Springer, 2011, pp. 173-193.
2. Bacon, D., P. Cheng, D. Frampton, D. Grove, M. Hauswirth, V. Rajan. Demonstration: Online Visualization and Analysis of Real-Time Systems with Tuningfork. – In: Compiler Construction. Berlin, Heidelberg, Springer, 2006, pp. 96-100.
3. Bartocci, E., Y. Falcone. Lectures on Runtime Verification. Introductory and Advanced Topics. Springer, 10457, LNCS, pp. 1-240 (in press).
4. Bloem, R., R. Dimitrova, C. Fan, N. Sharygina. Software Verification. – In: Proc. of 13th International Conference, VSTTE 2021, New Haven, CT, USA, 2021.
5. Ben, S. Designing Event-Driven Systems. O'Reilly Media, 2018.
6. Blokdyk, G. AWS IoT A Complete Guide. 5STARCooks, 2021.
7. T. Compass. Trace Compass, 2015. **https://projects.eclipse.org/projects/tools.tracecompass**
8. Cope, S. MQTT For Complete Beginners: Learn the Basics of the MQTT Protocol. Kindle Edition, 2020.
9. Dunkel, J., A. Fernandez, R. Ortiz, S. Ossowski. Event-Driven Architecture for Decision Support in Traffic Management Systems. – In: Proc. of 11th International IEEE Conference on Intelligent Transportation Systems, Beijing, China, 2008., pp. 7-13.

10. E c k e r t, M. Complex Event Processing with XchangeEQ: Language Design, Formal Semantics, and Incremental Evaluation for Querying Events. PhD Thesis, Ludwig Maximilians University Munich, Germany, 2008.
11. G a l, A., S. W a s s e r k r u g, O. E t z i o n. Event Processing over Uncertain Data. – In: S. Helmer, A. Poulovassilis, F. Xhafa, Eds. Reasoning in Event-Based Distributed Systems. Berlin, Heidelberg, Springer, 2011, pp. 279-304.
12. H i s s a m, S., G. M o r e n o, D. P l a k o s h, I. S a v o, M. S t e l m a r c z y k. Predicting the Behavior of a Highly Configurable Component Based Real-Time System. – In: Proc. of Euromicro Conference on RealTime Systems, Prague, Czech Republic, 2008. pp. 57-68.
13. A l l e n, J. F. Maintaining Knowledge about Temporal Intervals. – Communications of the ACM, Vol. **26**, 1983, pp. 832-843.
14. K h r i j i, S., Y. B e n b e l g a c e m, R. C h́e o u r, D. E. H o u s-S a i n i, O. K a n o u n. Design and Implementation of a Cloud-Based Event-Driven Architecture for Real-Time Data Processing in Wireless Sensor Networks. – The Journal of Supercomputing, 2022, pp. 1-28.
15. K n u p f e r, A., H. B r u n s t, J. D o l e s c h a l, M. J u r e n z, M. L i e b e r, H. M i c k l e r, M. M u l l e r, W. N a g e l. The Vampir Performance Analysis Tool Set. – In: Tools for High Performance Computing. Berlin, Heidelberg, Springer, 2008, pp. 139-155.
16. L i, N., Z. M a o. Administration in Role-Based Access Control. – In: Proc. of 2nd ACM Symposium on Information, Computer and Communications Security (ASIACCS'07), ACM, 2007, pp. 127-138.
17. M u n k, P. Visualization of Scheduling in Realtime Embedded Systems. PhD Thesis, University of Stuttgart, 2013.
18. P a c e, G. J., C. C o l o m b o. Runtime Verification. Springer, 2022.
19. S h a n d, B., P. P i e t z u c h, I. P a p a g i a n n i s, K. M o o d y, M. M i g l i a v a c c a, D. M. E y e r s, J. B a c o n. Security Policy and Information Sharing in Distributed Event-Based Systems. – In: S. Helmer, A. Poulovassilis, F. Xhafa, Eds. Reasoning in Event-Based Distributed Systems. Berlin, Heidelberg, Springer, 2011, pp. 151-172.
20. V a n d e r A a l s t, W. M. P. Formalization and Verification of Event-Driven Process Chains. – Inf. Softw. Technol., Vol. **41**, 1999, No 10, pp. 639-650.
21. W i m, P., H. S t e p h e n. Zinsight: A Visual and Analytic Environment for Exploring Large Event Traces. – In: Proc. of 5th International Symposium on Software Visualization (SOFTVIS'10), ACM, 2010, pp. 143-152.

Received: 11.09.2023; Second Version: 10.11.2023; Accepted: 28.11.2023