REAL-T: Time Modularization in Reactive Distributed Applications

Luis Daniel Benavides Navarro[1(✉)], Camilo Pimienta[2], Mateo Sanabria[1], Daniel Díaz[1], Wilmer Garzón[1], Willson Melo[1], and Hugo Arboleda[2]

[1] Colombian School of Engineering Julio Garavito, Bogotá, Colombia
{luis.benavides,daniel.diaz,wilmer.garzon}@escuelaing.edu.co,
{mateo.sanabria,willson.melo}@mail.escuelaing.edu.co
[2] Universidad Icesi, Cali, Colombia
{hfarboleda,cfpimienta}@icesi.edu.co

Abstract. In this paper, we propose REAL-T, a distributed event-based language with explicit support for time manipulation. The language introduces automata for operational time manipulation, causality constructs and Linear Temporal Logic for declarative time predicates, and a distributed-time aware event model. We have developed a compiler for the language and a dynamic run-time framework. To validate the proposal we study detection of complex patterns of security vulnerabilities in IoT scenarios.

Keywords: Distributed programming · Event oriented programming
Explicit and implicit time management

1 Introduction

Time management requirements in distributed computer systems are becoming more complex. Intrusion detection systems, Internet of things networks (IoT), autonomous vehicles, and smart cities are all examples of reactive, concurrent, and distributed systems with complex real-time management needs. Those systems support millions of interconnected devices with complex and dynamic deployment topologies. However, mainstream distributed computing tools still support relatively simple and naive models of time. Namely, explicit time management using the system clock to tag events, and implicit time management by means of the *next-instruction* abstraction in programming languages and computer systems. These simple abstractions have created complex usage patterns to address massive parallelism (see, common concurrency patterns in [7]), frequent resource sharing errors (e.g., liveness and data-race errors [5,6]) and convoluted event ordering and synchronization algorithms (see, for example, [25]).

Several strategies for explicit time management have been proposed to address the problems described above. Synchronous and asynchronous state machines address the problem of event ordering, pattern recognition, and formal specification of concurrent systems. Other state machine variants consider

J. E. Serrano C. and J. C. Martínez-Santos (Eds.): CCC 2018, CCIS 885, pp. 113–127, 2018.
https://doi.org/10.1007/978-3-319-98998-3_9

implicit time management and explicit time management, e.g. timed machines (see [12] for a complete overview). Temporal logic has been used to address real-time system specification and verification [4,17], error detection in concurrent systems [9,22], and Intrusion Detection [3,24]. Logical clocks [19] and vector clocks [21] have been proposed to address causal ordering of events in distributed systems, and have been implemented in several systems to address detection of complex distributed event patterns and debugging and unit testing of distributed concurrent application, see [6,27]. All these approaches suffer of at least one of two problems. First, they provide only some abstractions for time management. Two, except for [6,27], they are non distributed, thus assume centralized access to the program trace. We argue that both restrictions severely limit the applicability of the mentioned tools, considering the current computing systems have complex and heterogeneous requirements for time management, distribution, and concurrency.

In this paper we investigate the implementation of several time management strategies in REAL-T, a reactive event-based distributed programming language. We also evaluate their applicability in the context of Intrusion Detection Systems for IoT networks. Concretely, we provide the following contributions: REAL-T, A decentralized, elastic, and time-aware event-based model for distributed programming and the corresponding language design; A prototype implementation of a compiler supporting: automata for complex pattern modelling, causal predicates, and Linear Temporal Logic to address explicit time aware predicates; Evaluation of usage scenarios in the context of Intrusion Detection Systems for IoT networks.

The paper is organized as follows. Section 2 motivates our research analyzing actual problems in intrusion detection systems over IoT networks. Section 3 discusses work related to the issue. Sections 4 and 5 present the event-based distributed time model and the corresponding language design. Section 6 presents the prototype implementation of the compiler and the run-time virtual machine. In Sect. 7 we present usage scenarios. Finally, we conclude and discuss future work in Sect. 8.

2 Motivation: Time Constraints in Intrusion Prevention Systems

Intrusion Detection Systems (IDS) and Intrusion Prevention Systems (IPS) have been in the security landscape for a long time. On these systems multi-step attacks can be modeled using an automaton to identify a sequence of specified events. If the automaton accepts such a combination of events, it could be evidenced that an attack is occurring, and the system may log the attack or stop the computation.

An example of a multi-step attack is a Worm Attack [28] since the attack pattern is based on scanning, exploiting and finally developing a malicious action. Let us imagine a worm attack over a victim host that has installed an IDS.

The first step is to scan the victim's ports and find one that allows it to infiltrate, which makes the IDS identify event *e1* referring to the detected port scan activity. Then, a new event *e2* will be generated by the IDS when it detects a suspicious file is entering the system (dropper). The goal of a dropper is to download and install a malware (payload), when this occurs the IDS registers event *e3*. The next event generated by the IDS *e4*, happens when the malware runs and tries to fulfill its purpose (ex-filtration, disruption, tampering, etc.). Finally, the malware can go through a state of self-destruction to erase any trace of its existence in the victim's host, which can also be registered by the IDS as event *e5*. The attack has gone through five (5) events to fulfill its objective, and each step depends on the fulfillment of the previous ones.

Now consider such an attack over a Internet of things Network. The Internet of Things is a technological paradigm envisioned as a global network of machines and devices capable of interacting with each other [20]. A mechanism to detect a simple sequence of events is not enough to recognize attacks on such heterogeneous network. Not even, network-based IDS, which capture packets of network traffic and analyze them to detect possible attacks. Most of mainstream IDS/IPS systems have simple implementations of event sequence detectors, and even small variations on the sequence of events may affect the detection of an attack. Consider, for example, an IoT network that is under attack, due to the non-deterministic nature of the network, and the distributed nature of the attack, events can be triggered in the right malicious order (actual order), but detection may happen in different order. Having support for detection of event sequences is not a guarantee of attack identification. A modern IDS/IPS solution needs more sophisticated mechanisms to detect possible attacks, for example, identifying causality relations, or defining predicates on complex time dependencies. Therefore, in this paper, we argue that such systems could be enriched with real-time detection of intricate patterns of distributed events with sophisticated time dependencies.

3 State of the Art

Several dimensions must be considered when implementing time models in computer Systems (see [12] for a complete discussion). Time models may be discrete or dense (continuous). They may model time simply by imposing order on events or by means a metric system, tagging each event with a clock reading. They may support linear time, where each state has only one successor and one predecessor, or time branching, where each state has one predecessor but could have several successors. They may model time via explicit concepts of time, e.g., a clock, or via implicit concepts of time, e.g., the *next* step in a sequential algorithm. Finally, the modeler should take into consideration concurrency and composition, in particular considering that the problem of synchronization of parallel activities have created a plethora of abstractions (e.g., thread, process, tasks) and several complex usage patterns (see, common concurrency patterns in [7]), frequent resource sharing errors (e.g. liveness and data-race errors [6]) and convoluted event ordering and synchronization algorithms [25].

According to Furia et al. [12] temporal models of time in modern computer systems may be classified in three categories: operational time models, declarative time models, and hybrid time models. Operational time formalisms describe the evolution of a system, starting from a certain initial state, transitioning to other states through events or transitions. Finite state automata, Statecharts [14], and Petri networks [26] are examples of operational formalisms. On the other hand, declarative models describe explicitly temporal properties that must endure during the execution of the system. Most of these models are based on temporal logic [4]. Temporal logic is a family of first order logic that has temporal operators on time-dependent propositions. Temporal logic allows programmers to describe complex temporal relations among events happening in a computation. Hybrid models include abstractions from operational and declarative formalisms. The model proposed in REAL-T is a hybrid model, including explicit abstractions for automata and explicit abstractions for Propositional Temporal Logic.

Several actual implementations of these models have been proposed, see for example [9,10]. Monitoring-Oriented Programming (MOP) frameworks aim to reduce the gap between formal specification and implementation [22]. MOP frameworks monitor whether the activities that are being performed by the software comply with a formal specification. The original MOP framework has been extended with state machines and temporal logic. However, the implementations of such frameworks, address only non-distributed applications and assume full access to the computation trace (see application of MOP frameworks to security [3]). REAL-T extends these ideas into a fully distributed framework for real time monitoring.

To complete these discussion, we augment the taxonomy above with a category for distributed logical time abstractions. Logical time has been proposed by Lamport [19] and Mattern [21] to address partial orders of distributed events without a global synchronized clock. Additionally, the order of messages is based on a causality relation among events. Several implementations of these concepts have been proposed, see logical clocks implemented in the Horus system [27] and automata with logical clocks implemented in [6]. REAL-T includes these concepts with the additional support of Propositional Temporal Logic.

4 A Distributed Event-Based Time Model

The proposed programming model has three main components: an event model, a message model, and a time model. The event model describes the general architecture of the distributed application and what events are considered. The message model describes how messages are exchanged and differentiates the messages that exchange meta-information of the events, and the messages comprising the distributed application. Finally, the time model describes different considerations of time for the programming model components.

4.1 The Event and Message Models

The model assumes the existence of a base distributed application where specific behaviors want to be detected or reinforced. The distributed application runs in distributed hardware, e.g., several servers and devices sending and receiving information. The network is a component of the distributed application, it connects all the devices and servers. The base application exchanges messages through the network to accomplish its purpose.

REAL-T's main constructs are event classes, those classes are instantiated with a singleton policy, i.e., each event class creates one event monitor on each running node of the distributed application. The proposed model considers only one type of event: method call. When a method call is detected in the base application, the meta-information of that call is broadcast to the nodes participating in the distributed application. This message does not interfere with the distributed messages of the base application. Furthermore, no restriction is imposed regarding synchronization with the messages of the base application.

Event monitors are the instances of the event classes. They consume messages with event information, and they react to those events. The reaction may be a simple notification, e.g., registering the event in a log file, or it may modify the original behavior of the base application. Complex patterns of events are detected using a predicate language. The current implementation supports finite deterministic automata, to detect complex sequences of distributed events [6], causal predicates to reinforce causality [21], and Linear Temporal Logic (LTL, sometimes called PTL or PLTL [15]) to address more complex temporal predicates [22].

The message model differentiates explicitly two types of messages in the application. First, the regular messages that address the purpose of the distributed application. Second, the messages representing the meta-data of events. The meta-data messages are exchanged over the REAL-T framework while the regular messages are exchange over the mechanisms defined by the distributed base application. Thus, even though regular messages and meta-data messages may be triggered by the same events, they travel over different distributed software infrastructures.

Figure 1 shows the main concepts of the event and message models. Nodes one, two, and three represent a distributed application with a three-layer architecture. Each node executes a multi-threaded component of the application and exchange messages through defined mechanisms depicted as bidirectional solid lines and white envelopes. Each node has an instance of a monitor. The figure shows only the computation that monitor three detects, thus we only show meta-event messages (black envelopes) arriving at monitor three. Finally, the time line at the bottom of the figure, shows the arriving order of messages with the event's meta-information at monitor three, emphasizing that different components may see different histories of the distributed application. The other monitors may see different histories, and even the nodes of the application may see different histories.

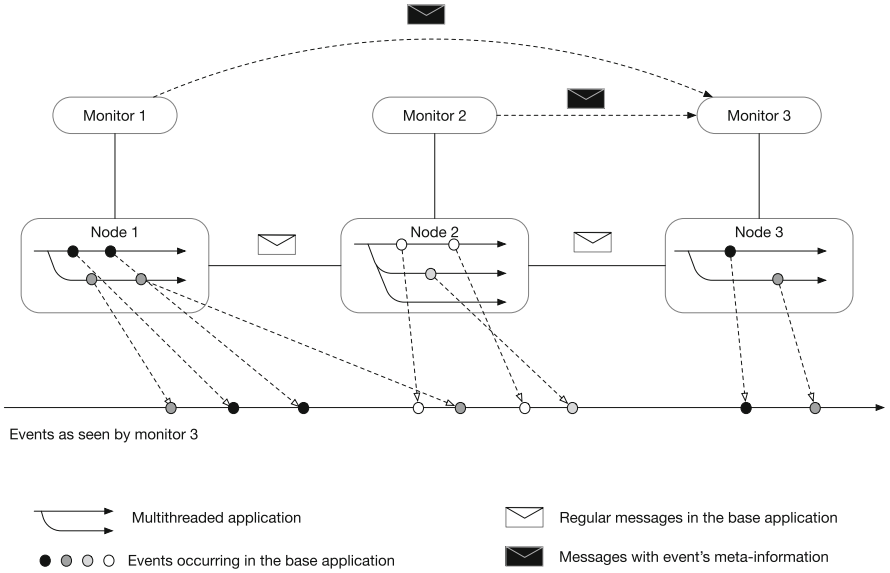


Fig. 1. Message model

4.2 Time Model

Above we describe the general event model and the distributed message model. Those models characterize the non-deterministic behavior of concurrent and distributed applications. We now introduce the time model for REAL-T. The time model considers three type of time models. The first is operational time, where time is not explicit but is only modeled through the order of messages. The second is logical time [19,21] to address partial orders of events, predicating, for example, over causal relations. Finally, we introduce declarative time using LTL where custom models of time are introduced by the programmers of event classes.

Operational Time: Operational formalisms [12] describe explicitly the evolution of software systems. In our model we use Deterministic State Automata to describe complex sequences of events. The automaton is concerned only with the possible next transitions, thus enforcing specific sequences of events. Each transition on the automaton may be guarded with a boolean guard. In our model each monitor may have an automaton definition and depending on the arrival order of messages each automaton will detect different histories of the distributed computation (see Fig. 2).

Logical Time: REAL-T also incorporates a notion of logical time, virtual time, and the global state of distributed systems, following ideas presented by Lamport and Mattern [19,21]. On this model, each event is tagged with a value from the

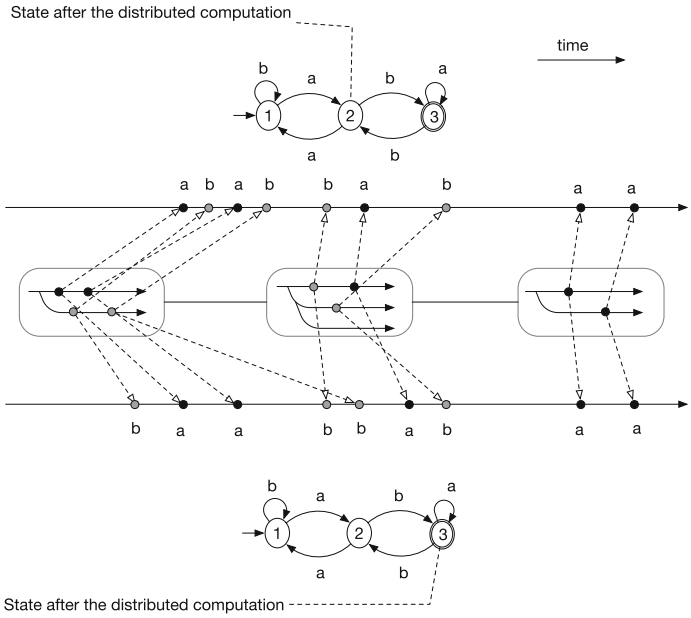


Fig. 2. Operational time model

logical clock instance deployed on the node where the event originated. Logical clocks are updated with the information from other logical clocks, such information arrives with event's meta-information. The model allows programmers to write predicates on the causality relationship among events, i.e., when an event has causal influence over another. To understand this relation, let us a look at the Happens-before relationship defined in [19], which states that a causal relationship must meet any of the following cases. Let `a`, `b` and `c` be events, two events are considered to have causal relation if:

- `a` and `b` are in the same process and `a` takes place before `b`; then `a` happens-before `b` (`a` is causally related to `b`).
- `a` represents an event of sending a message while `b` is an event of receiving a message, then `a` happens-before `b`.
- If `a` happens-before `b` and `b` happens-before `c`, hence, `a` happens-before `c` due to relationship transitivity.

In any other case, events are considered concurrent ([21]).

Declarative Time Model (Linear Logical Time): Finally, REAL-T incorporates a time model based on Propositional Temporal Logic (PTL) [11]. Using PTL programmers write temporal predicates asserting temporal relations among events in a sequence of distributed events. Concretely, REAL-T supports the operators described below, where ϕ and ψ are PTL formulas:

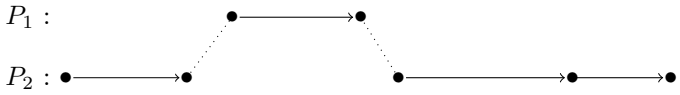
- $\bigcirc\phi :=$ "*Next:* In the next moment ϕ is true".
- $\Diamond\phi :=$ "*Eventually:* In some future or present moment ϕ is true".
- $\Box\phi :=$ "*Always:* ϕ is true in all future moments".
- $\phi \ \mathsf{U} \ \psi :=$ "*Until:* ϕ continues being true up until some future moment when ψ is true".
- $\phi \ \mathsf{W} \ \psi :=$ "*Unless:* ϕ continues being true unless ψ becomes true".

As an example, suppose that P_2 is a process that at some point of its execution over time sends a message to P_1. P_1 then receives the messages and sends a result to P_2 so that it can continue its execution (a basic example of distributed computation). This behavior allows us to infer a main property, at some point in the execution of P_2, P_1 will eventually happen. If you must specify the previous behavior this specification in PTL would be:

$$Exe(P_i) := \text{The process } \ P_i \text{ is being executed}$$

$$Spec : \Diamond(Exe(P_2) \ \Rightarrow \Diamond Exe(P_1) \ ) \wedge \Box(Exe(P_2) \ \Rightarrow \neg Exe(P_1) \ )$$

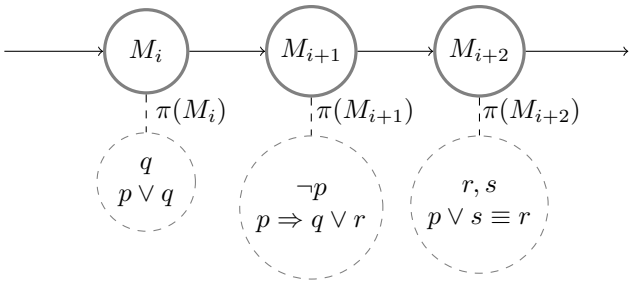
The second condition of the specification informs that the sending of messages between P_2 and P_1 is not immediate in the same way it could be ensured that the sending of messages between P_1 and P_2 is not immediate. A possible behavior of that specification would be:



LTL introduces the notion of time into a sequence of states or moments. Each state in the series is represented as a model at different moments in time. A model ($\mathcal{M}$) in LTL is composed of:

- A set of moments $\mathbb{M}$.
- A order relation $\prec$: $\mathbb{M} \times \mathbb{M} \rightarrow \{true, false\}$, the relation may be transitive, non-reflexive, linear, or discrete. This relation defines how moments are ordered and represents the temporal structure of the model.
- A function, $\pi : \mathbb{M} \rightarrow \mathbb{P}(prop)$ such as π maps each moment/state to a set of valid propositions, where $\mathbb{P}(prop)$ is the power set operator.

A concrete model may be represented as follows:



Where,

- $\mathbb{M} = \{M_i, M_{i+1}, M_{i+2}\}$
- $\prec$ is a linear order on $\mathbb{M}$, such that, $(\forall i, j | i < j : M_i \prec M_j)$
- The function :
  $\pi = \{(M_i, \{q, p \vee q\}), (M_{i+1}, \{\neg p, p \Rightarrow q \vee r\}), (M_{i+2}, \{r, s, p \vee s \equiv r\})\}$

The model of PTL we are studying considers a discrete and linear model of time, thus each moment of time has at most one successor. However, the event and message models described above require the time model to have specific characteristics. First, each monitor has a concrete instance of a formula attached to it. Second, the set of formulas defines a custom model of time, where the events of interest of each formula define the set of events that moves the model from moment to moment. The next moment is determined by the arrival of an event of interest, i.e., each formula is evaluated once an event of interest arrives to the node. Third, the model of time may vary from node to node. As mentioned before, each node may see a different history of the computation, then the temporal model may be different, especially the sequence of events of interest seen by each node. This implies that evaluation of the formula depends on the model seen by each formula instance. This non-deterministic behavior simplifies implementation and requires no-synchronized clocks.

5 REAL-T by Example: Time Aware Constructs

REAL-T incorporates constructs to implement the event model, the message model, and the time model. In this section, we are going to present the main elements of the language using a security test example. Consider a distributed application with several servers doing business computations and persisting data to a database replicated in a different set of servers. We are interested in detecting `write` commands on the database that are not in a secure session, i.e., they are not in between a `login` - `logout` sequence of events.

Figure 3 shows the implementation using an automaton in REAL-T. First, the event class is declared with the name `SecurityTest`. The events of interest (alphabet of the automaton) are defined from lines 2–10. The signature of the first event, `sessionLogin`, is defined with parameter `uid` of type `String`. The concrete event definition is a boolean expression. The `causal` construct indicates that the method call is only matched when there is a causal relation with the previous event. The `call` construct matches any call to the method `destroySession` on objects of type `SecurityManager`. The call may be executed on any host of the distributed application. The `args` construct bounds the parameter values to the variables. Note that, once a variable is bounded, subsequent events using the variable are only matched if the value of the event parameter is the same as the one in the variable. The `sessionLogout` event is defined similarly but it is interested in `destroySession` method calls on any host. On the other hand, the `write` event uses the construct `host(localhost)` to indicate that it is only interested in `write` events happening on the local host, i.e., writes that happen

```
1  eventclass SecurityTest{
2   event sessionLogin(String uid):
3     causal(call(* SecurityManager.createSession(String)))
4           && args(uid);
5   event sessionLogout(Fqn name, Object x):
6     causal (call(* SecurityManager.destroySession(String)))
7           && args(uid);
8   event write(String uid, Fqn memorySpace, Object value):
9     causal (call(* DataBase.write(String, Fqn, Object)))
10          && args(name) && host(localhost);
11
12   automaton securityViolationDetector(String uid,
13                                  Fqn memorySpace, Object value){
14     start init: (write(uid, Fqn, value) -> securityViolationt) ||
15                 (sessionLogin(uid) -> login);
16     login:(write(uid, Fqn, value) -> login) ||
17                  (sessionLogout(name, x) -> init);
18     end securityViolation;}
19
20  reaction before
21  securityViolationDetector.securityViolation(String uid,
22     Fqn memorySpace, Object value){
23        //Reaction to security violation}}
```

Fig. 3. Example of causal automaton implementing a security test

on the database host. Once the events are placed in lines 12 to 19 we define the automaton. The automaton has three states and four possible transitions. From the `init` state, the automaton may transition to the `login` state if a *login* event is received; or it may transition to the `securityViolation` state, if a *write* event is received before a *login* event. If the automaton is in the `login` state, it stays there if it receives a *write* event, or transitions to the init state if it receives a *logout* event. Lines 21 to 24 show the reaction definition which is executed before transitioning to the `SecurityViolation` state.

Figure 4 shows the same implementation but using a PTL formula. In this case the same set of events are defined, however, those events will determine the set of moments for the temporal model. Once a concrete event is detected, the temporal model moves to the next moment and evaluates the formula. The formula defined in lines 4 to 8 asserts that, in the system is always true (`always` construct), that immediately (`next` construct) after a *login* event, *write* events are received until a logout event (`until` construct). If the formula is violated the reaction is triggered.

```
1 eventclass SecurityTest{
2   // Event definition
3   ltl securityViolationDetector(String uid,
4           Fqn memorySpace, Object value){
5               always(sessionLogin(String uid) ->
6                   next(write(uid, memorySpace, value)
7                       until sessionLogout(name,x)))}
8   //Reaction definition}
```

Fig. 4. Example of PTL formula implementing the security test

6 Compiler Implementation

We have developed a runtime library and a compiler for REAL-T[1]. The event and messaging models are implemented using a group communication library [2]; this constitutes the core of the runtime framework. The compiler translates REAL-T programs into AspectJ [16] code, and then it is compiled into Java bytecode. Implementation of automata support, uses an automata library, [23] augmented with group communication. Detection of causal predicates uses vector clocks [21]. Finally, we translate propositional temporal logic formulae into Büchi automata Sect. 4.2, feeding the automata with distributed events. The implementation of automata support and causality support with distribution, follows similar techniques as those described in [5,21].

We now present an overview of the translation of temporal logic into Büchi automata. Büchi automaton [8] is an extension of classic finite automata created to read and evaluate infinite words [1,11,18]. The main difference with finite automata is that the acceptance criterion over an infinite word is that there exists a run of the automaton which visits infinitely often one or more final states. Further information regarding the details of the translation is beyond the scope of this work, we encourage the interested reader to see [11]. However, we use now an example to show the mechanics of the translated automata.

Consider the following formula defined in REAL-T:

$$\textbf{always}(\text{login} \rightarrow \textbf{next}(\text{write } \textbf{until } \text{logout}))$$

The formula describes a property where always, immediately after a `login` event, there is a sequence of `write` events until a `logout` event occurs. Once the first event occurs, it will only recognize `write` until a `logout` appears. Note that any other declared event will be considered as another element in the alphabet. So, if any other event happens between a `login` and a `logout` different to `write`, it is a violation of the temporal property. Note that the Büchi automaton of the formula (see Fig. 5) contains a transition labeled with `1`, this transition is followed if the implementation moves the clock to the next moment. Thus, even though, the automaton does not have a transition for a specific event, if an event

[1] https://github.com/unicesi/eketal.

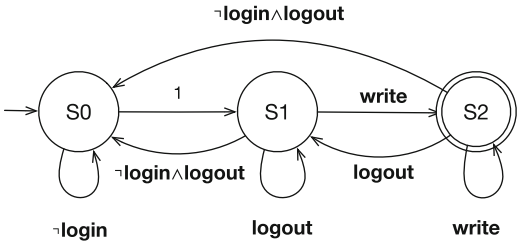


Fig. 5. Buchi automaton for LTL property

notification arrives, the model will move to the next moment. REAL-T translates the formula into a Büchi automaton using a library described in [13].

7 Securing an IoT Ecosystem

We evaluate the applicability of the proposed language in the context of an IoT scenario. Our scenario is composed of three main components. An IoT Sentinel monitoring several IoT devices at a particular site (e.g., home, factory, hospital, etc.). A network-based IDS (NIDS) monitoring messages flowing from the IoT devices to an IoT platform deployed on the cloud (e.g., Oracle IoT, Samsung Artik, Amazon IoT). A host-based IDS (HIDS) running on the storage server on the IoT platform infrastructure deployed on the cloud.

The IoT Sentinel generates three kind of events: (i) **urlAccess**, when an IoT device with device Id (dId) access to a web server (url), (ii) **dropperDownload**, when a dropper (Object X) is downloaded to an IoT device (dId) using a web server (url), (iii) **payloadDownload**, when a dropper (Object X) located in an IoT device (dId) downloads a payload (Object Y). The NIDS generates event **servicesScanning** when a scanning activity with a severity (sev) is performed from an IoT device (dId) over a server (target) on the IoT platform. At last, the HIDS generates an event **injectionAttack** when an injection (dataHash) coming from an IoT device (dId) is detected against itself (target). For such a distributed configuration, traditional IDS components detect those events as independent nonsuspicious events. Then, after an attack, an automatic system may notice the event relation when accessing the full trace of the computation, i.e., when all the trace logs from all IDS components are compared together. However, REAL-T can do better. In REAL-T, the specific sequence of related events can be described using a PTL formula over distributed events on the system. The use of PTL to detect this attack is shown in Fig. 6.

Figure 6 shows event definitions from lines 3 to 18, where the first three events are detected in the *IoTDEvices* group of hosts, and the other two events are detected at the *NIDS* and *HIDS* groups. In lines 20 to 27 a PTL formula is defined. The formula links together events through the common values `dId, X, Y, url` and `target`. The formula consumes events respecting causal order, and moves the model to the next moment each time a defined event arrives. The

```
1  eventclass SecurityTest{
2
3   event urlAccess(String dId, String url):
4     causal(call(* IoTSentinel.urlAccess(String))) && args(dId)
5           && host("IoTDevices");
6   event dropperDownload(String dId, Object X, String url):
7     causal (call(* IoTSentinel.download(String))) && args(dId)
8           && host("IoTDevices");
9   event payloadDownload(String dId, Object Y, Object requester):
10    causal (call(* IoTSentinel.download(String))) && args(dId)
11          && host("IoTDevices");
12  event servicesScanning(String dId, String sev, hostname target)
13    causal (call(* NIDS.scan(String)))
14            && args(dId) && host("NIDS");
15  event injectionAttack(String dId, String dataHash,
16                                                hostname target)
17    causal (call(* HIDS.inject(String injection)))
18            && args(dId) && host("HIDS");
19
20  ltl securityViolationDetector(String did,
21          String url, Object X, Object Y, String sev,
22          String target, string dataHash){
23              always(urlAccess(did, url)) ->
24                eventually(dropperDownload(did, X, url)) ->
25                  eventually(payloadDownload(did, Y, X)) ->
26                    eventually(serviceScanning(did, sev, target)) ->
27                      !eventually(injectionAttack(did, dataHash, target))
28
29  reaction before securityViolationDetector(String dId, String url,
30          Object X, Object Y, String severity,
31          String target, String dataHash){
32          // isolate IoT device and gather Forensic Evidence}}
```

Fig. 6. Example of PTL formula detecting an distributed IoT attack

last part of the formula is negated to force triggering of the reaction when the pattern is violated. Lines 29 to 32 define the reaction when the security violation has occurred. The reaction is to isolate the IoT device (dId) and gather forensic evidence for later adversary analysis. Notice, that the reaction is taken on all the IDS. Thus, using REAL-T, not only the attack is detected in real time, but the IDS take a common action against the attack.

8 Conclusions

We have presented REAL-T a programming language for real time monitoring of distributed applications. The language supports a fully distributed programming model, with a notion of distributed events and distributed messages, no

central control is needed for the run-time infrastructure of the language. The language also incorporates a model to detect complex temporal relations among events. It supports operational time management through automaton constructs, predicates over causal relations of events using logical vector clocks, and predicates of Propositional Temporal Logic. We have explored composition of time models, allowing automata transitions to depend on causal relations of atomic events. We have validated the time model implementing a functional compiler capable of monitoring distributed java applications. The implementation of the compiler has concrete constructs for the automata and translates PTL formulas into Buchi automata. The run-time framework supports logical clocks and presents a fully distributed framework based on group communication. We have evaluated the usage of the model in the context of intrusion detection systems for IoT networks.

Several open questions remain. First, we must explore run-time performance on real scenarios. We also should explore different semantics for the language, addressing how the instantiation policy affects the patterns that programmers may use. We also must explore applicability in other domains, for example, how real-time monitoring system may improve the performance and behavior of autonomous vehicles.

References

1. Baier, C., Katoen, J.P.: Principles of Model Checking. MIT press, Cambridge (2008)
2. Ban, B., Grinovero, S.: JGroups (2011)
3. Barringer, H., Goldberg, A., Havelund, K., Sen, K.: Program monitoring with LTL in EAGLE. In: 18th International Parallel and Distributed Processing Symposium 2004, April 2004
4. Bellini, P., Mattolini, R., Nesi, P.: Temporal logics for real-time system specification. ACM Comput. Surv. **32**(1), 12–42 (2000)
5. Benavides Navarro, L.D., Barrera, A., Garcés, K., Arboleda, H.: Detecting and coordinating complex patterns of distributed events with KETAL. Electron. Notes Theor. Comput. Sci. **281**, 127–141 (2011)
6. Benavides Navarro, L.D., Douence, R., Südholt, M.: Debugging and testing middleware with aspect-based control-flow and causal patterns. In: Issarny, V., Schantz, R. (eds.) Middleware 2008. LNCS, vol. 5346, pp. 183–202. Springer, Heidelberg (2008). https://doi.org/10.1007/978-3-540-89856-6_10
7. Benton, N., Cardelli, L., Fournet, C.: Modern concurrency abstractions for c#. ACM Trans. Program. Lang. Syst. **26**(5), 769–804 (2004)
8. Büchi, J.R.: Symposium on decision problems on a decision method in restricted second order arithmetic. Stud. Log. Found. Math. **44**, 1–11 (1966)
9. Chen, F., Roşu, G.: Java-MOP: a monitoring oriented programming environment for Java. In: Halbwachs, N., Zuck, L.D. (eds.) TACAS 2005. LNCS, vol. 3440, pp. 546–550. Springer, Heidelberg (2005). https://doi.org/10.1007/978-3-540-31980-1_36
10. Chen, F., Roşu, G.: MOP: an efficient and generic runtime verification framework. In: ACM SIGPLAN Notices, vol. 42. ACM (2007)

11. Fisher, M.: An Introduction to Practical Formal Methods Using Temporal Logic. Wiley, Hoboken (2011)
12. Furia, C.A., Mandrioli, D., Morzenti, A., Rossi, M.: Modeling time in computing: a taxonomy and a comparative survey. ACM Comput. Surv. **42**(2), 1–59 (2010)
13. Giannakopoulou, D., Lerda, F.: From states to transitions: improving translation of LTL formulae to Büchi automata. In: Peled, D.A., Vardi, M.Y. (eds.) FORTE 2002. LNCS, vol. 2529, pp. 308–326. Springer, Heidelberg (2002). https://doi.org/10.1007/3-540-36135-9_20
14. Harel, D.: Statecharts: a visual formalism for complex systems. Sci. Comput. Program. **8**(3), 231–274 (1987). https://doi.org/10.1016/0167-6423(87)90035-9
15. Haydar, M., Boroday, S., Petrenko, A., Sahraoui, H.: Propositional scopes in linear temporal logic. In: Proceedings of the 5th International Conference on Novelles Technologies de la Repartition (NOTERE 2005) (2005)
16. Kiczales, G., et al.: Aspect-oriented programming. In: Akşit, M., Matsuoka, S. (eds.) ECOOP 1997. LNCS, vol. 1241, pp. 220–242. Springer, Heidelberg (1997). https://doi.org/10.1007/BFb0053381
17. Konur, S.: A survey on temporal logics for specifying and verifying real-time systems. Front. Comput. Sci. **7**(3), 370–403 (2013)
18. Kröger, F., Merz, S.: Temporal Logic and State Systems. Springer, Berlin (2008). https://doi.org/10.1007/978-3-540-68635-4
19. Lamport, L.: Time, clocks, and the ordering of events in a distributed system. Commun. ACM **21**(7), 558–565 (1978)
20. Lee, I., Lee, K.: The internet of things (IoT): applications, investments, and challenges for enterprises. Bus. Horiz. **58**(4), 431–440 (2015)
21. Mattern, F., et al.: Virtual time and global states of distributed systems. Parallel Distrib. Algorithms **1**(23), 215–226 (1989)
22. Meredith, P.O., Jin, D., Griffith, D., Chen, F., Roşu, G.: An overview of the MOP runtime verification framework. Int. J. Softw. Tools Technol. Transf. **14**(3), 249–289 (2012)
23. Møller, A.: dk.brics.automaton – finite-state automata and regular expressions for Java (2017). http://www.brics.dk/automaton/
24. Naldurg, P., Sen, K., Thati, P.: A temporal logic based framework for intrusion detection. In: de Frutos-Escrig, D., Núñez, M. (eds.) FORTE 2004. LNCS, vol. 3235, pp. 359–376. Springer, Heidelberg (2004). https://doi.org/10.1007/978-3-540-30232-2_23
25. Ousterhout, J.: Why threads are a bad idea (for most purposes). In: Invited talk Given at USENIX Technical Conference (1996). https://web.stanford.edu/~ouster/cgi-bin/papers/threads.pdf
26. Petri, C.A.: Fundamentals of a theory of asynchronous information flow. In: IFIP Congress (1962)
27. van Renesse, R., Birman, K.P., Maffeis, S.: Horus: a flexible group communication system. Commun. ACM **39**(4), 76–83 (1996)
28. Robiah, Y., Rahayu, S.S., Shahrin, S., Faizal, M., Zaki, M.M., Marliza, R.: New multi-step worm attack model. arXiv preprint arXiv:1001.3477 (2010)