Received December 16, 2017, accepted February 24, 2018, date of publication March 8, 2018, date of current version March 28, 2018.

Digital Object Identifier 10.1109/ACCESS.2018.2813887

A Novel Runtime Verification Solution for IoT Systems

KORAY INCKI AND ISMAIL ARI
Department of Computer Science, Özyeğin University, 34794 İstanbul, Turkey
Corresponding author: Koray Incki (koray.incki@ozu.edu.tr)

ABSTRACT Internet of Things (IoT) systems promise a seamless connected world with machines integrating their services without human intervention. It's highly probable that the entities participating in such autonomous machine to machine interactions are to be provided by different manufactures. Thus, integrating such heterogeneous devices from many providers complicates design and verification of IoT systems at an unprecedented scale. In this paper, we propose a novel runtime verification approach for IoT systems. The contributions of our proposed solution include: exploiting the interactions in message sequence charts (MSC) to specify message exchanges of constrained application protocol-based IoT systems in terms of events, a novel event calculus for formally describing IoT system constraints specified by means of MSCs, and an event processing algebra that uses complex-event processing techniques for detecting failures in the system by monitoring the runtime event occurrences with respect to the system constraints defined by event calculus. We further demonstrate the viability of proposed solution with case studies.

INDEX TERMS Internet of Things, runtime verification, event calculus, complex-event processing, message sequence charts.

I. INTRODUCTION

Considering the complexity of modern computers [1], comprehensive verification techniques, such as model checking and theorem proving, can not practically analyze the system's correctness. On the other hand, functional testing can be considered the most suitable method for determining correctness, which is examined only by a subset of systemic behaviors. Nevertheless, functional tests may not reveal extraordinary cases that complicated software might exhibit during execution. Runtime verification (RV) [2] is a method in which monitors oversee the run of a system under test (SUT) in order to detect whether it meets a specific constraint, which is defined by a correctness property. Should the monitor notice that the system is in violation of the property, then it can activate the management mode, and therefore the system also leads to safe behavior. The capability of a monitor to assess the system's properties during execution and to take into account all system runtime properties and inputs from the surrounding domain promotes RV as the best method to make sure a computing system behaves as expected, especially in the field of IoT systems.

The main goal of our study is to facilitate the research and practice in IoT domain by enabling them to seamlessly conceptualize the system under development in terms of events occurring in it. Event Calculus (EC) provides tools for describing such systems in terms of events [3]–[5]. For an EC to specify a particular domain we need to *(i) specify simple events in the system; (ii) specify the algebra that correlate those simple events in order to deduce complex conclusions; (iii) define time-varying properties of the system*. An EC devised particularly for IoT domain would not only help specify the expected behavior of a system in a human-readable form, but it also would facilitate utilization of various event-processing engines for monitoring and verification of system behavior at runtime.

The contributions of this research extend our initial results in [6] such that a domain-specific event calculus is proposed for facilitating system specification and RV of IoT systems; thereby enabling specification of correctness properties as used for describing monitors in RV. IoT systems are heterogeneous systems such that each system may have different computing, memory, power, networking, sensing and actuating capabilities. A plethora of application layer protocols (e.g., CoAP, MQTT) are utilized for facilitating application development with such heterogeneous devices. Each protocol exhibits unique interaction model; so, the choice of application layer protocol determines the design and development phases of an IoT system. Thus, the application layer choice also alters the event calculus to be used for specifying an IoT system. In this research, we adopt CoAP as the application layer protocol, because it allows RESTful application development framework, and promotes IoT proliferation by means of its service-oriented messaging model. The proposed solution can be tailored for other protocols by following the steps explained in the paper.

Contributions of this paper are threefold; first, we present a domain-specific EC for CoAP-based IoT systems; second, we develop an EC-to-EPL statement mapping, in order to facilitate utilization of Esper CEP engine. Then, we demonstrate the applicability of this solution by case studies.

Section II gives a succinct literature review on runtime verification solutions proposed for IoT systems, embedded systems, or those using complex-event processing. Section III describes the case scenario that we will be referring to throughout the paper. In Section IVintroduces the techniques and methodologies used, so that the paper provides a thorough reading experience; and Section V explains how IoT interactions are expressed with EC, then in Section VI gives the event processing statements. Section VII details both the implementation and results of the experiment on the running example.

II. RELATED WORK

Chen *et al.* [7] have proposed a methodology for interoperability testing of CoAP implementations. As for the CoAP specifications [8], a set of interoperability tests was selected. They favored passive testing for two reasons: First, the passive test does not interfere with the execution of the SUT. That's why, it is best suited for testing interoperability at runtime. Second, passive tests do not introduce additional costs in network communication, so they are more suitable for resource-constrained domains such as IoT. Packets that are interchanged amongst CoAP endpoints are caught by a network sniffer and logged. Recorded execution logs are examined offline against the test scenarios by utilizing a test tool to determine if the runtime behavior complies with the expected behavior. Our approach also employs packet sniffer component, but we present a novel solution for online and non-intrusive testing of an IoT system.

Medhat *et al.* [1] present a novel RV methodology for real-time cyber-physical systems (CPS) that are real-time sensitive and have constrained physical resources such as memory. Their proposed solution relies on two concepts: (i) The runtime monitor is executed in certain periods. Events that happen between two monitor calls are buffered and handled later by the monitor when it's called. (ii) The monitor is assumed to be flawless, meaning that, it does not generate false outputs (neither positive, nor negative). Therefore, no event can be missed. The buffer for recording events that occur between successive monitor runs is assumed to be of a bounded-size. Their research deals with individual embedded system verification, and relies on code instrumentation in order to enable runtime monitoring of the system. Thus, it incurs memory overhead and possibly behavior alterations due to running monitoring threads.

Gaaloul *et al.* [5] propose an online RV technique that relies on certain mediation techniques, and use Complex-Event Processing (CEP) [9] to catch and mitigate invalid calls. The approach tries to make sure that IoT entities are requested with services that they are built for. The proposed architecture is composed of a mediation platform that processes services calls, by which they aim to prevent invalid service calls using CEP. Their proposed solution automatically produces the necessary components to verify the service calls at runtime. Nevertheless, the proposed solution depends on a mediation platform that is deployed as a special CoAP entity in the same network where the SUT reside. In such an approach, SUT does not exhibit the same execution trace as it would when it is deployed at the customer site without a mediation platform, which modifies how the service calls are handled.

Yu *et al.* [10] propose a predictive runtime verification solution for CPS. CPS generally consist of embedded IoT systems. The main purpose of the solution is to prevent any failure before it happens by means of prediction. The programs on CPS devices must be instrumented in order to generate runtime events. Predictive monitors trigger controlling operations such as stopping or repair for tuning the application behavior whenever they detect or predict a failure. Their approach doesn't deal with behavior of system of IoT devices that is composed of more than one IoT device. Moreover, the SUT must be instrumented in order to generate runtime verification data, which is known to incur performance, behavior and memory footprint overheads.

Kane [11] proposes an runtime monitoring architecture for observing safety-critical vehicular systems through their black-box components. The proposed solution consists of a passive bus-monitor that addresses particularly the CAN network used in vehicles. The bus-monitor can analyze system properties that are observable on the bus. Such monitor implementations manage all SUT components as black-box. Note that aforementioned monitor implementations are crucial for such systems that are composed of several sub-components provided by various manufacturers. Thus, the intrinsic behavior of those components cannot be attained easily. The monitor observes the CAN bus communication amongst the system components by attaching itself directly on the system bus. This connection is associated with a semi-formal interface that tracks the bus and generates atomic projections for a monitor based on the observed bus status, which reflects the recorded image of the monitor. The execution trace is a sequence of those recorded images. Our runtime verification approach for IoT systems assumes a system of black-box IoT entities as the problem domain, just as this monitoring approach treats the system of CAN bus attached devices as a system of systems and attacks the RV of such system of systems as black-boxes. It depends on the formal specification of component communication amongst those devices. On the other hand, we present an event calculus framework for formally specifying IoT system interaction and runtime monitor constraints, and consequently facilitating use of CEP techniques for RV purposes.

III. RUNNING EXAMPLE

In case when the network needs to be self-healed, self-organized and be deprived of any centeralized features, the Wireless Token Ring Protocol (WTRP) is applied [12]. WTRP features high quality provisions for networks of limited bandwidth and bounded latency situations. WTRP is best suited for resource constraint networks such as IoT, because it supports constructing ad-hoc networks dynamically, provides energy saving measures and efficient transport mechanisms. Token ring protocol dictates observance of a predetermined order of messaging between participating endpoints. The sequential order in such a system might be broken due to several reasons, such as endpoints' power shortage or movement of endpoints to out of reach. WTRP relies on individual nodes to employ specific algorithms to bring back a functioning network whenever a failure occurs; we assume that the sensor nodes are non-byzantine [13], but they might fail due to random system failures due to poor programming skills. Thus, in order to overcome such failures at runtime, our proposed passive sniffing solution would ensure the robustness of the network.

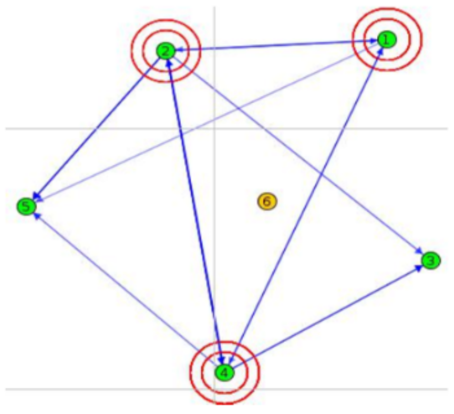


FIGURE 1. Cooja simulation of WTRP.

Fig.1 demonstrates a simulation of WTRP network in Cooja [14]. We will use the event calculus proposed in this paper to express sequential relations between request and response events. Note that, the token ring protocol is assumed to pass around the token in increasing order of mote ids in the network.

IV. BACKGROUND

A. CoAP-BASED IoT SYSTEMS

Many of the Internet utilities have proliferated thanks to the utilization of web services that are architected according to RESTful APIs [15]. The special IETF working group on CoRE (Constrained RESTful Environments) was formed particularly to outline a viable RESTful framework for the resource constrained devices and networks in IoT domain. They generated an application layer (OSI layer 7) protocol called Constrained Application Protocol (CoAP) for facilitating utilization of RESTful APIs. CoAP introduces an easy to use application phenomenon for resource constrained devices, primarily for those devices with limited battery, low memory footprint, and limited computation power.

CoAP adopts a Client-Server based communication pattern as in other RESTful services. A CoAP client sends a Request Message to a CoAP Server, stating the required action with a special Method Code on a resource of the Server. Those resources resident on servers are identified by URIs (unified resource identifier). Should the Server accomplish to process the corresponding Request, then it sends a Response Message back to the originating Client with a proper Response Code. It's noteworthy that any CoAP device (entity) can behave both as a client and a server in M2M interactions.

The messaging model of CoAP is an asynchronous interaction model. The messages are exchanged over UDP packets [8]. There are four different message types: Confirmable (CON), Non-confirmable (NON), Acknowledgement, Reset (RST). Even though the communication is based on UDP, an optional reliability is provided with exponential back-off. Thus, those four message kinds are exchanged in a Request/Response type of interaction model. A Request can be conveyed via both a CON and NON message; and, the Response of a Request might be separately sent in a CON/NON message, as well as piggybacked in an ACK message.

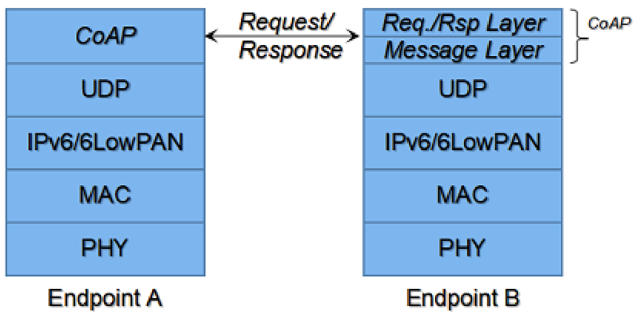


FIGURE 2. CoAP OSI layering.

Request and Response semantic of CoAP interactions enables us to consider the protocol layer as consisting of two intrinsic logical layers (Fig.2). This logical representation allows us to manage request/response messaging by means of matching Method Code and Response Code (i.e., Request/Response layer), while the communication layer details of UDP and asynchronous messaging are handled in a different logical layer (i.e., Message layer).

An entity participating in a CoAP network is called an endpoint (which can be both Client and Server). A message is uniquely identified by a MessageID field contained in the message format of protocol. MessageID field is utilized for removing duplicate messages, as well as for providing optional reliability. If a message is to be transmitted reliably, then it has to be sent in a CON message. If an endpoint does not receive an ACK message for a CON message in a predefined timeout, then several re-transmissions might be issued with exponential back-off, until a valid ACK is received with the same MessageID. Note that, an endpoint might respond with a RST message, if it is not capable of processing a CON message.

Inference on CoAP behavior can be elaborated by using Message Sequence Charts (MSC), which is a formal description technique developed by ITU-T [16] for providing a trace language for the specification and description of the communication behavior of system components and their environment by means of message interchange. Since in MSCs the communication behavior is presented in a very intuitive and transparent manner, particularly in the graphical representation, the MSC language is easy to learn, use, and interpret. Therefore, we can obtain an event-based representation of CoAP interactions. We will elaborate on this concept in Section-V.

B. EVENT CALCULUS REVISITED

An event is defined as any happening in a context at certain time, that is irrevocable, which causes the system state to change. Event calculus allows for generating commonsense decisions about actions and corresponding changes [17]. The event calculus comprises of *events*, and time-dependent attributes, called *fluents*, and *timepoints*. The events are assumed to happen on a single time axis. Commonsense reasoning is defined as humans making inferences on everyday situations [3]. If we can automate the process by which we derive conclusions about happenings around us, then we can develop much better user experience. There are two concrete happenings that rely on these building blocks; (i) an event can occur in a unique time instance, (ii) a system property is true only at a single time-point.

According to Mueller [17], some problem domains are much easily represented with event calculus due to their very nature, such as partially-ordered events, triggered events, etc. In order to make use commonsense reasoning for event calculus, we must first identify the area of interest, and then provide common knowledge about that area. The resulting situations that arise after an event happens at some time are described. For instance, a certain event occurring under certain conditions might initiate a particular system property. That is, if the event happens at a certain point in time in a given context, then the corresponding system property becomes true after that very same time instance. Likewise, yet another event might terminate a certain system property, so the system property becomes false after that time-point when the event happens [3].

Event calculus algebra make use of predicate logic for elaborating time-varying properties of a system, namely *fluents*. Researchers proposed various versions of versions of event calculus. The event calculus we use conforms to Discrete Event Calculus (DEC) [17]. The predicate functions and corresponding descriptions are given in Table-1.

TABLE 1. DEC predicates and meaning.

Predicate	Meaning
Initially(f)	*f is True at timepoint 0*
HoldsAt(f,t)	*f* is true at *t*
Happens(e,t)	*e* occurs at *t*
Initiates(e,f,t)	if *e* occurs at *t*, then *f* is true and not released from the commonsense of inertia after *t*
Terminates(e,f,t)	if *e* occurs at *t*, then *f* is false and not released from commonsense of inertia after *t*
Releases(e,f,t)	if *e* occurs at *t*, then *f* is released from commonsense of inertia after *t*
Trajectory(f_1, t_1, f_2, t_2)	if f_1 is initiated by an event that occurs at t_1, then f_2 is true at $t_1 + t_2$

Happens(e,t) states that an event e happens at a timepoint t. Initiates(e,f,t) (respectively, Terminates(e,f,t)) means that if an event e happens at time t, then it makes fluent f true (respectively, false) instantly. HoldsAt(f,t) states that fluent f is true at timepoint t.

DEC allows us to descriptively specify event-driven requirements of an IoT system. Our approach aims to provide an EC formulae to verify IoT interaction behavior. EC elements facilitates directly representing such computing systems in terms of events. Therefore, EC allows using the logic theory for verification of both design artifacts and runtime system. As pointed out in [5], EC provides a representation that is very similar to interaction models such as RESTful APIs. Besides, EC formulae involves a definitive time value; thus, allowing us to distinguish between events occurring at the same time in an event-based system, such as IoT.

C. COMPLEX-EVENT PROCESSING

Complex-event processing, CEP for short, provides techniques and tools to reveal complicated reasoning about large-scale domain-specific software systems. CEP is usually deployed with the aim of decision making on runtime systems. Some examples are database systems, network communication, intrusion detection, etc. Even though those complex systems of systems produce terabytes of information, only a fraction of those are necessary for coming up with intelligent decision. In order to achieve that, CEP engines correlate basic (or so called simple) events through special transformation and aggregation functions.

CEP engines, such as Esper [9], allow us to declare event descriptions with special purpose languages. Event Processing Language (EPL) is used in Esper engine. By using EPL, you can identify certain event-patterns that you seek for, and register those EPLs with the engine. Esper engine runs along the system under inspection, and provided that certain events of interest occur, it raises a flag for each pattern of interest (e.g., for detecting failures in a system). The APIs provided by such engines enable us to design continuous queries and complex causality relationships between disparate event streams with an expressive EPL. EPL statements are continuously executed as live data streams are pushed through. Esper has built-in support for Java language, thus enabling us to use Plain Old Java Objects (POJO) classes to represent events.

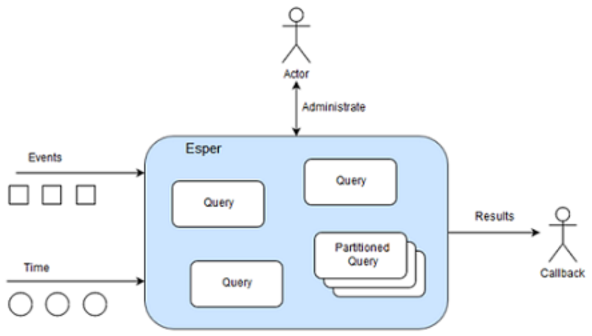


FIGURE 3. Esper is a container for EPL statements.

Essentially, Esper acts as a container of EPL statements Fig.3. EPL statements fundamentally specifies queries that analyze events and time, and then detect situations. Esper provides a nest for EPL queries and organizes their lifecycle and execution.

EPL is a SQL-like declarative language. It is used for aggregating information and deriving knowledge from one or more event streams. Those enable also to join and merge event streams. Events are inserted and processed as continuous streams of information. The basic syntax for EPL is as follows:

$$SELECT < select\_list >$$
$$FROM < stream\_def >$$
$$WHERE < search\ conditions >$$

The *select* clause in EPL specifies the event properties or events to retrieve in *select_list*. The *from* clause specifies the event stream definitions and stream names to use. The *where* clause specifies search conditions that specify which event or event combinations to search for. There are other clauses such as *having*, *group by*, *order by* in the language. For example, the following statement returns the time-stamp from a Token-Ring event stream whenever the mote with $id = 1$ broadcasts a message.

$$select\ timestamp\ from\ TokenRing\ where\ moteId = 1$$

V. IoT REDEFINED: AN EVENT CALCULUS FOR SYSTEM SPECIFICATION

A. REPRESENTING IoT WITH EVENTS

In our endeavor for representing IoT systems by using event calculus, we first need to express the interaction between endpoints in terms of simple events. In order to achieve this goal we will utilize Message Sequence Charts (MSC) [16], a common graphical language derived by ITU for describing communication scenarios in the industrial applications. It provides a graphical language that handles asynchronous interactions in communication systems, such as CoAP.

MSC's are frequently used in verification of communication systems in the literature ([18]–[20]). A major difference in interpretation of MSC with respect to those literature is that we only deal with *Request* and *Response* asynchronous messages in our approach. Because, we restrict ourselves to the *send message* events that are observable from the network in a black-box fashion, whereas others elaborate on process level *receive message* events, which are solely observable via a process-level code instrumentation.

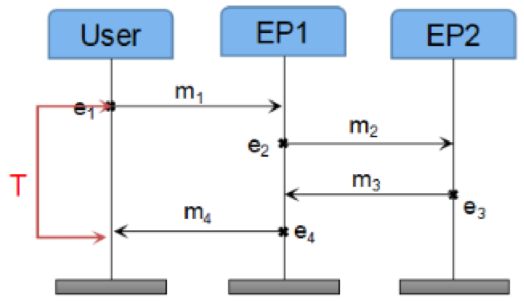


FIGURE 4. MSC for a CoAP service *S*.

We will utilize MSC to extract a specification for the SUT in terms of events occurring in the system. Let's consider the MSC of a CoAP system as in Fig.4. The vertical lines in the figure are lifelines for each endpoint in a CoAP system, which represent the time axis for each endpoint. The time increases downwards. Endpoints engage in interaction by sending and receiving asynchronous messages (i.e., m_1, m_2, m_3, and m_4). Each message send action generates an observable event in the network (i.e., e_1, e_2, e_3, and e_4). Note that, in a *Request-Reply* interaction model, each *Reply* message corresponds to a previous *Request* message; but, a *Request* message does not have to cause a *Reply* message, as it might be the case that the *Request* is issued just as a control function, not a query. For the sake of simplicity, we'll assume in Fig.4 that (m_1, m_4) and (m_2, m_3) constitute *(Request,Reply)* pairs of messages. This example demonstrates a CoAP scenario in which a *User* requests a *service* from an endpoint ($EP1$), then in turn, $EP1$ requests some other service from another endpoint ($EP2$), such that there is a causal relation between events appearing on the vertical lines.

Let $EP = \{EP_1, EP_2, \ldots, EP_n\}$ be a set of endpoints in an IoT system, and let A be a message alphabet for the network, where $[n]$ denotes the number of endpoints. We represent each asynchronous message m with the label $send(i, j, msg)$, which indicates the event of an endpoint EP_i sending a message msg to an endpoint EP_j. We further define the set $E = \{send(i, j, msg) \mid i, j \in [n]\ msg \in A\}$ as the set of all send events. Remember that an EP can behave as both a client and a server in a CoAP network; thus, a $send(i, j, msg$ event can be either a *Request* or a *Reply* event. Therefore, E can be partitioned into E^R and E^r subsets representing set of *Request* events and *Reply* events, respectively. $\varepsilon = \varepsilon^R \cup \varepsilon^r$ is the set of all *send* events, where $\varepsilon^R = \{send(i, j, msg_{req}) \mid i, j \in [n]\ msg \in A\}$ and $\varepsilon^r = \{send(i, j, msg_{rep}) \mid i, j \in [n]\ msg \in A\}$, respectively. The MSC M then can be described as

1) a set of *send* events, E, containing two distinct sets of *send* events, ε^R and ε^r.

2) a mapping function *ep* that maps each event to an endpoint, $ep : E \mapsto [n]$
3) a bijective mapping between each (*Request, Reply*) message pairs, $f : \varepsilon^R \mapsto \varepsilon^r$
4) a labeling function, l that identifies each event as either *Request* or *Reply*, $l : E \mapsto \varepsilon$
5) $\forall i \in [n]$, there exists a total order $\prec_i$ on the events of endpoint i, such that the transitive closure of the relation $\prec \doteq \cup_{i\in[n]} \prec_i \cup \{(r, f(r))) \mid r \in \varepsilon^R\}$ is a partial order on E.

Let's consider the MSC in Fig.4. The label for e_1 for sending of message m_1 is $send(User, EP1, m_1)$. Note that, we have another event with $f(e_1)^{-1}$ such that m_4 is a *Reply* message to m_1; therefore, $f(e_1) = e_4$. Even though, degenerate MSC might occur in a CoAP network, we restrict our MSCs to non-degeneracy condition. An MSC is degenerate, if there are two *send* message events e_1 and e_2 such that $l(e_1) = l(e_2)$, where $e_1 \prec e_2$ and $f(e_1) \prec f(e_2)$. For a thorough coverage of non-degeneracy condition and MSC formalization the reader should refer to [19] and [18].

Now that we have a definition for an MSC, we can use this definition to express a specification that an MSC can deliver. We define the specification of an MSC by its *linearisation*. A *linearisation* of an MSC M, which is represented by a word $w = w_1\, w_2 \dots w_n$ over M (e.g., $w_1 = l(e_1)$, $w_2 = l(e_2)$, $w_3 = l(e_3)$, and $w_4 = l(e_4)$ for Fig.4), is attained by a total order of events in E; and it is considered as a string over ε. In other words, a *linearisation* is said to exist if a total order of $(e_1\, e_2 \dots e_n)$ exists between the events in E such that whenever $e_i \prec e_j$ we have $i \prec j$, and for $w(i) = l(e_i)$.

An MSC represents event interactions for a single service composition scenario in a CoAP-based IoT system; therefore, the specification of an IoT system, Γ, that delivers N distinct services, would consists of a disjoint set of N MSC linearisations. That is, specification contains N MSCs $M_1, \dots, M_N$ each for a distinct service implementation, in which $E_1, \dots, E_N$ are disjoint event sets. Let $\Sigma = \cup_{j=1}^{N} E_j$ be the disjoint sets of events in Γ; $\Upsilon = \cup_{j=1}^{N} A_j$ be the message alphabet of Γ, and $\Psi = \cup_{j=1}^{N} EP_j$ be the set of endpoints in Γ. Then the language of an MSC Specification Γ is the union of languages of all MSCs in Γ. Note that the message alphabets and endpoints in different MSCs can be similar, because an endpoint may engage in several similar interactions in various service compositions.

We have shown that a linearisation of a single MSC $M_j \in \Gamma$ can be achieved by means of *send message* events occurring on each endpoint, $EP_i \in \Psi_j$ where Ψ_j is the set of endpoints for M_j. MSC guidelines [16] provides various graphical operations such as *coregion, par* for detailed elaboration of communication scenarios. However, we will assume no such operations exist on the MSCs we deal with; those are to be handled in model-based testing approach we are working on. We will utilize this *event* phenomenon in facilitating an *event calculus* for IoT systems in the next section.

B. EVENT CALCULUS FOR CoAP

Considering an execution of a MSC M_i, the trace can be monitored in terms of *send message* events in the network. As pointed out in [21], testing is an event-centric activity; and events recorded as indications of actions in the execution trace should match with the sequence of events occurring in the linearisation of MSC M_i. The sequence of events in a trace implicitly exhibit a *follows* relation between each pair of consecutive events. Our aim is to formulate an event calculus that is succinct enough to express both the expected behavior captured in the linearisation of a MSC in terms of events, and the observed behavior captured as the trace of events from a CoAP network. Consequently, we can compare both behaviors to conclude with a *Pass/Fail* decision at runtime.

Before we dive into the formulation of event calculus, let's elaborate on types of relations that might identify the correlation between events in a MSC. Remember that, there is a visual and temporal/causal correlation between the events on the vertical lines of a MSC for a CoAP scenario. Considering Fig.4, e_1 happens both visually and temporally *before* e_2, because the events exhibit a causal relation in order to deliver the required service. Note that, the *follows* relation is *transitive*, meaning that if e_2 *follows* e_1 and e_3 *follows* e_2, then e_3 *follows* e_1. Based on these definitions, we can define following relations for event sequences of a MSC M_k:

1) $f(e_j, e_i) = e_i \prec e_j$ where $e_i, e_j \in E_k$ and $i \prec j$ for $i, j \in [n]$: defines the *follows* relation in M_k
2) $f_i(e_j, e_i) = e_i \prec^i e_j$ where $e_i, e_j \in E_k$ and $i \prec^i j$ for $i, j \in [n]$: defines the *immediately follows* relation between (e_i, e_j) such that $\nexists e_m \in E_k \mid (e_i \prec e_m) \wedge (e_m \prec e_j)$ where $(i \prec m) \wedge (m \prec j)$.
3) $t(e_i, e_j) \leq T$: defines a temporal relation between two events such that e_j happens in at most T time after e_i happens.
4) $s(e_i, e_j)$: defines an domain-specific semantic relation between two events; for instance, e_j carries a token id that is bigger than e_i.

where E_k is the event set of MSC M_k. Those four relations will enable us to express complex relations between events in terms of event calculus. Note that, relations (1) and (2) must always be observed in a runtime verification scenario, but relations (3) and (4) are observed only when they are defined in the domain of application under test. Note also that, $\prec$ and $\prec^i$ relations are

- irreflexive, $\neg(e_i \prec e_i)\forall e_i \in E_k$, and
- asymmetric, $\nexists e_i, e_j \in E_k \mid e_i \prec e_j \wedge e_j \prec e_i$

As an example, let's try to express a requirement of CoAP standard stating that every *CON* type message must be followed by an *ACK* type message in *EXCHANGE_LIFETIME* [8], by using the relations defined above. We can express this requirement as

$$Req(CON) = f(e_{ACK}, e_{CON}) \wedge [t(e_{ACK}, e_{CON}) < EXCHANGE\_LIFETIME], \quad (1)$$

where (e_{CON}, e_{ACK}) are any pairs of send events for a *CONfirmable* message and its corresponding *ACKnowledgement* messages [8].

Eq. 1 states that every *CON* message must be followed by an *ACK* message in *EXCHANGE_LIFETIME* time. Hereby, we can use this equation to express runtime monitors for failure and success situations of the requirement in terms of events. In order to conclude with a success verdict, the equation must hold TRUE for *(CON, ACK)* event pair. However, in order for the system fail for this requirement we must have

$$\begin{aligned}\neg Req(CON) &= \neg\{f(e_{ACK}, e_{CON}) \\ &\quad \wedge [t(e_{ACK}, e_{CON}) < EXCHANGE\_LIFETIME]\} \end{aligned} \quad (2)$$

or

$$\begin{aligned}\neg Req(CON) &= \neg f(e_{ACK}, e_{CON}) \\ &\quad \vee [t(e_{ACK}, e_{CON}) >= EXCHANGE\_LIFETIME] \end{aligned} \quad (3)$$

Event linearisation for the sample MSC in Fig.4 constitutes an expected behavior of message interactions between endpoints, such that it represents the specification for the service *S Requested* by event e_1:

$$Req(S) = e_1 \prec e_2 \prec e_3 \prec e_4, \quad (4)$$

where $Req(S)$ represents the requirement for service requested by e_1. In a *Request/Reply* interaction model such as CoAP, *User* represents another endpoint that requests a service provided by endpoint *EP1* with event e_1. In order for this scenario to fail, event trace monitored at runtime must deviate from that of Eq.-4. This linerarization can be interpreted in event relations of MSC as

$$\begin{aligned}Req(S) &= f_i(e_2, e_1) \wedge f_i(e_3, e_2) \\ &\quad \wedge f_i(e_4, e_3) \wedge [t(e_4, e_1) < T] \end{aligned} \quad (5)$$

Note also that, the requirement can be satisfied only with a conjunction of all the relational components that represent causal order of events in the expected behavior MSC. In Eq-5, $t(e_4, e_1) < T$ constitute a temporal constraint between events e_4 and e_1. In case any of those relations is not observed at runtime, then the requirement is not satisfied (Eq-6).

$$\begin{aligned}\neg Req(S) &= \neg f_i(e_2, e_1) \vee \neg f_i(e_3, e_2) \vee \neg f_i(e_4, e_3) \\ &\quad \vee [t(e_4, e_1) >= T] \end{aligned} \quad (6)$$

The core CoAP network can provide communication among hundreds, even thousands of endpoints delivering multitude of services. Thus, we need a concept of context for distinguishing similar events based on the surrounding conditions. A *context* for a CoAP interaction scenario can be defined as the set of events sequences that are visually traced on an MSC diagram in order to accomplish a *Request* for a certain service. The events that are not related to the expected behavior is not relevant to the to the *Requested* service, thus they are out of context. So, only those events that appear on the diagram are context events, provided that the diagram is complete.

A context C_{MSC} can be described with the following definitions:

1) C_E: set of context events that appear on an MSC diagram
2) e_0: an initial *Request* event for the service of context
3) a set of pairwise *follows* relations: $f(e_j, e_i)$,
4) an optional set of pairwise temporal constraints: $t(e_j, e_i)$,
5) an optional set of pairwise semantic constraints: $s(e_j, e_i)$,

where $e_i, e_j \in C_E$. Let A_{comp} be a subset of C_E such that $A_{comp}(e_j, e_i) = C_E \setminus (e_j, e_i)$.

Now that we have defined all the relations of an IoT system, we can interpret those with event calculus. As SEC/DEC defines in its fundamental predicate logic, relations that identify time dependent properties of a system constitute the *domain-specific fluents* for that system. Thus, the relations defined for an MSC are *fluents* of CoAP-based IoT system. We can tailor those in order to represent any combinations of complex relations between event traces. By using the predicates of Table-1 we can express the relations of MSC as

$$\begin{aligned}f_i(e_j, e_i) &= Happens(e_j, t_j) \wedge Happens(e_i, t_i) \\ &\quad \wedge \neg Happens(e_k, t_k), \end{aligned} \quad (7)$$

where $e_k \in A_{comp}(e_j, e_i)$ for $(t_i < t_k)$, $(t_k < t_j)$, and $(t_i < t_j)$. Note that this is an *immediately follows* relation defined over *context* C_E, thus only those events $e_k \in A_{comp}(e_j, e_i)$ can cause this relation to fail. It is important to note that the investigation for $f_i(e_j, e_i)$ begins with occurrence of e_i, therefore $Happens(e_i, t_i)$ sets a precondition for $f_i(e_j, e_i)$. Rooting on that precondition, $\neg f_i(e_j, e_i)$ can be expressed as

$$\neg f_i(e_j, e_i) = f_i(e_k, e_i) \vee f(e_i, e_j) \quad (8)$$

where $e_k \in A_{comp}(e_j, e_i)$. Eq.-8 states that $f_i(e_j, e_i)$ fails iff e_i is followed by an event $e_k \in A_{comp}(e_j, e_i)$ or e_i follows e_j. Eq.-8 can be elaborated in event calculus terms by expanding f_i's as in Eq.7

$$\neg f_i(e_j, e_i) = \begin{cases} Happens(e_k, t_k) \\ \quad \wedge Happens(e_i, t_i) \\ \quad \wedge \neg Happens(e_j, t_j), & \text{if } Cond_A \\ Happens(e_j, t_j) \\ \quad \wedge Happens(e_i, t_i), & \text{if } Cond_B, \end{cases} \quad (9)$$

where $Cond_A \equiv \{e_k \in A_{comp}(e_j, e_i)\} \wedge t_i < t_k) \wedge (t_k < t_j) \wedge (t_i < t_j)$, and $Cond_B \equiv t_i > t_j$. Eq.9 states that e_j does not *immediately follows* e_i iff either e_j happens before e_i or $e_k \in A_{comp}(e_j, e_i)$ happens *immediately after* e_i.

If we visit the sample MSC in Fig.4 again, we can elicit all the event traces that cause the sample scenario to either succeed or fail as in Table-2. The event relations appearing in *Success* and *Failure* rows of the table identifies *runtime monitors* for the MSC in Fig.4. So, we can elaborate those

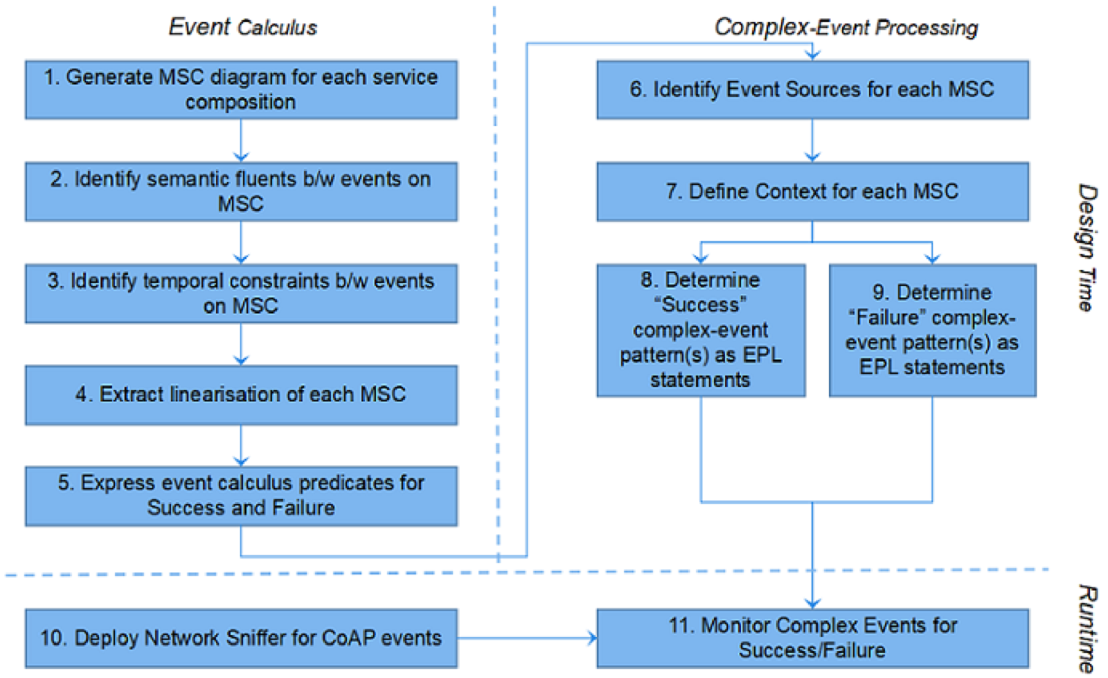

Event Calculus
1. Generate MSC diagram for each service composition
2. Identify semantic fluents b/w events on MSC
3. Identify temporal constraints b/w events on MSC
4. Extract linearisation of each MSC
5. Express event calculus predicates for Success and Failure
Complex-Event Processing
6. Identify Event Sources for each MSC
7. Define Context for each MSC
8. Determine "Success" complex-event pattern(s) as EPL statements
9. Determine "Failure" complex-event pattern(s) as EPL statements
Design Time
10. Deploy Network Sniffer for CoAP events
11. Monitor Complex Events for Success/Failure
Runtime

FIGURE 5. RV process with complex-event processing.

TABLE 2. Context and event verdicts for Fig.4.

Verdict	EPL Statement for RV
C_E	$\{e_1, e_2, e_3, e_4\}$
e_0	e_1
Success	$f_i(e_2, e_1) \wedge f_i(e_3, e_2) \wedge f_i(e_4, e_3)$
Failure	$f_i(e_3, e_1) \vee f_i(e_4, e_1) \vee f_i(e_4, e_2) \vee f_i(e_1, e_2) \vee f_i(e_3, e_4) \vee f_i(e_2, e_3)$

event relations as in Eq.7 and Eq.9 so that we get a representation of runtime monitors in event calculus. In the next section, we are going to explain how we can translate those basic event calculus predicates and constraints into complex event processing statements.

VI. EVENT PROCESSING FOR RUNTIME VERIFICATION

RV of a system requires representing system specifications of a SUT in terms of monitoring framework. Afterwards, the monitoring framework observes the behavior of the SUT to conclude with particular verdicts of *Success, Fail or Inconclusive* for certain constraints. In this section, we present a transformation mechanism that will facilitate to develop EPL statements from EC relations defined in Section-V, and a reference architecture that employs the proposed framework.

In order to systematically define how an event processing solution can be tailored for runtime verification, we follow a series of consecutive transformation steps. Fig.5 summarizes the process that we defined for generating an event processing solution for runtime verification. We have developed our reference architecture incorporating Esper CEP engine by following those steps process. However, one can tailor the process for event processing engines other than Esper by customizing the steps 7 through 11.

CEP engines that we use should allow us to define a context for each service, and help us express complex-event relations for *success* and *failure* verdicts. We can achieve those goals by employing certain constructs in Esper CEP engine as listed in Listing.1. Esper provides a notion of context, which enables us to define a set of circumstances or facts that surround a particular event [9].

```
Create a Context per e0 of each MSC
Insert each MSC event into Variant Stream
Apply MATCH_RECOGNIZE pattern on Variant
  Stream to observe Success
Apply EVERY pattern on Variant Stream
  to observe Failure
End context at last event or timeout
```

Listing 1. Event processing steps.

A context takes a cloud of events and classifies them into one or more event sets that are called *context partitions*. An event processing operation that is associated with a context operates on each of these context partitions independently. By this notion of context, we can analyze the events associated with a particular MSC diagram under a certain context partition. The context partition would be started with e_0 for each C_{MSC} (Table-2). The context can be terminated by receiving of an end event or a timeout value defined for the particular MSC behavior. The timeout value can be set as the maximum time it takes between the starting event and the finishing event for the MSC under test.

Moreover, the *CoAP Events* sniffed from the network should be filtered such that only the context events are processed at runtime monitors (i.e., EPL statements). Therefore, we utilize another construct of Esper, *variant stream* (*RVSpec*) for maintaining an event stream that consists only of those defined under the context C_{MSC}. Remember that, in order to conclude with a *Success* verdict all the event correlations must be observed on the runtime event trace. Thus, we use *match_recognize* pattern processing construct of Esper. The *[pattern* element of *match_recognize* construct enables us to indicate an exact trace of events, each of which *immediately follows* each other. Let us remind that the *Fail* can occur whenever any of the *immediately follows* relations (Eq.-9) is violated. Thus, we need a runtime monitor for each *not immediately follows* relation ($\neg f_i$).

```
1  create context CtxSample
   initiated by pattern
3  [every-distinct(startevent.srcId, startevent.dstId,
        startevent.mId)
   startevent = CoAPEvent(srcId = e1.id)] @inclusive
5  terminated by pattern
   [endevent = CoAPEvent(srcId = startevent.destId) or
        timer:interval(T)];
7
   context CtxSample
9  create variant schema RVSpec as CoAPEvent;

11 context CtxSample
   insert into RVSpec
13 select * from CoAPEvent where srcId = e1.id or srcId
        = e2.id or srcId = e3.id or srcId = e4.id;
```

Listing 2. EPL statements for context-based RV.

Let us now give an example on how to write the EPL statements for an MSC by writing those for Fig.4. Code listing in List.2 summarizes the basic EPL statement that we use for determining a *Success* verdict at runtime. The context is initiated for each distinct service invocation and terminated when receiving the terminating event or a timeout T passes. Notice that each distinct service invocation is uniquely identified by the triplet ($srcId$, $destId$, $msgId$) because of CoAP features. As you can see from the code listing between lines 13 through 16, we insert only those events that are associated with the context into the variant stream. The code listing in List.3 provides an example of how to detect a pattern of events that observe the visual order as in MSC of Fig.4. Note that we tag each EPL statement with *@Name*(..) so that we can distinguish visually the outputs of each statement. *FAIL* statement returns the ending event for the context. If the context ends before the end event arrives, then *FAIL* statement returns a *nullpointer*, thereby we can deduce that the runtime monitor yielded *Fail*. However, if *SUCCESS* statement returns a *count* of 1, then it means that the runtime monitor yielded a *Success* verdict. *FAIL* statement is the complement of *SUCCESS* statement in List.3, so it's not a comprehensive *Fail* monitoring statement.

The code listing in List.4 presents a case of *Fail* verdict as an EPL statement. The *every* $\rightarrow$ operator allows us to represent custom event patterns that follow each other. We can

```
@Name('FAIL')
context CtxSample
select context.endevent from RVSpec.std:lastevent
output snapshot when terminated;

@Name('SUCCESS')
context CtxSample
select count(*) as SuccessVerdict from RVSpec
match_recognize (
measures A.srcId as aId
pattern (A B C)
define
  A as A.srcId = e1.id,
  B as B.srcId = e2.id,
  C as C.srcId = e3.id
) output when terminated and context.endevent.srcId
     = e4.id;
```

Listing 3. EPL statements for success verdict.

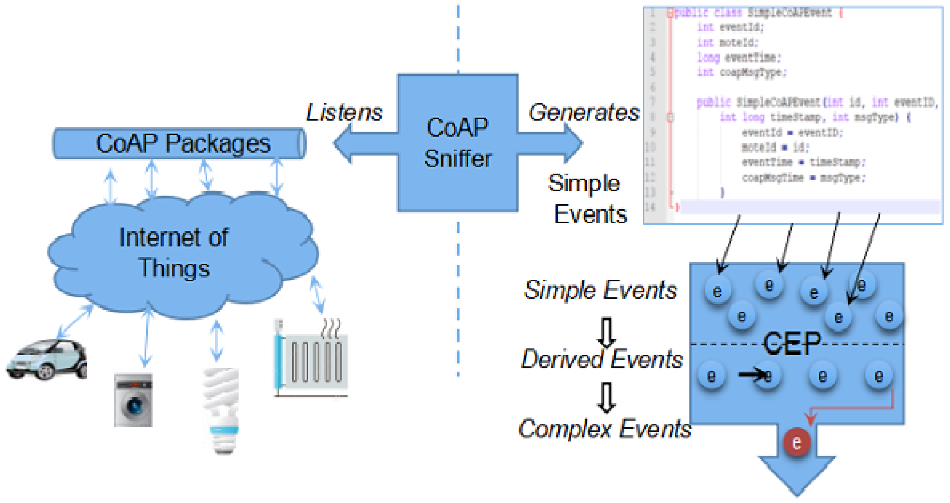
```
@Name('FAIL-1')
context CtxSample
select count(*) as FailOneVerdict from pattern [
every
rsp1 = RVSpec(srcId= e1.id) -> ((rsp2=RVSpec(srcId =
      e3.id ) or
rsp2 = RVSpec(srcId = e4.id)) and not rsp3 = RVSpec(
     srcId = e2.id ))
]
output when terminated;
```

Listing 4. EPL statements for fail verdict.

filter those events according to certain properties, such as event id. The $FAIL-1$ statement in List.4 states that when an event with id $e1$ is followed by an event with id $e3$ or an event with id $e4$ and not by an event with id $e2$, then return the count for that occasion. So, for each context, if that count equals to 1, then this is a case for yielding a *Fail* verdict. Similar statements can be written for other failure situations easily.

Fig.6 shows how the reference architecture was designed in order to reveal failure cases from simple coap events. The detailed design of *CoAP Sniffer* is provided in [6]. *CoAP Sniffer* listens to the IPv6 network for any CoAP communication. As soon as it captures a new CoAP message, it is parsed into an event representation in terms of *SimpleCoAPEvent Class*. Each *SimpleCoAPEvent* instance represents a simple event, which is later injected into the CEP engine (e.g., Esper). The EPL statements that are developed specifically for the constraints of SUT processes those simple events, consequently resulting in verdicts of Success or Failure in terms of complex events (i.e., red events in the figure). Each instance of *SimpleCoapEvent* instance is uniquely identified with an *eventId*, and instantiated with a timestamp value indicating occurrence of event, a *destId* for destination identification, and a *srcId* for identifying the source of the message.

The loosely-coupled design approach, thanks to RESTful-like CoAP, in the reference architecture (Fig.6) enables us to modify the building blocks of the architecture without compromising the integrity. The Esper CEP engine, which is implemented in Java, can be deployed on any platform that supports Java Virtual Machine (JVM).



```
public class SimpleCoAPEvent {
    int eventId;
    int moteId;
    long eventTime;
    int coapMsgType;

    public SimpleCoAPEvent(int id, int eventID,
        int long timeStamp, int msgType) {
            eventId = eventID;
            moteId = id;
            eventTime = timeStamp;
            coapMsgTime = msgType;
        }
}
```

Figure. 6. CEP assisted runtime verification reference architecture.

VII. EXPERIMENT

A. IMPLEMENTATION

We have used a prominent real-time operating system that is particularly designed for resource constrained embedded devices, Contiki [22]. The WTRP is implemented on each mote of type Zolertia-Z1 according to [12]. The simulation in Fig.1 is run on Cooja [14]. The configuration of the computer that we performed the simulation is Intel i7-6700HQ CPU that runs at 2.6GHz with 16GB of RAM.

The simulation environment consists of 5 motes, each of which is uniquely identified with an increasing value of mode ids, and a border router that is used for providing connectivity over IPv6 network. Each mote (m_i) transmits a broadcast message to the network when it owns the token and then passes the token on to the next mote (m_j) with id that satisfies $m_j - m_i = 1$ relation.

CoAP messages are passively captured by a sniffer as described in [6]. Captured messages are converted to *SimpleCoapEvent* instances and injected into Esper CEP engine. A failure case is fictitiously generated by adding a random seeded error function in WTRP algorithms of motes, which randomly causes a mote to tranmit a message without owning the token. The failure situation is diagnosed by monitoring for a sequence of messages that violate Eq.-9. This condition renders the predicate function *OO(Diff,t)* to become *False*. The mote that randomly transmits an erroneous message also outputs an appropriate message to indicate the error situation in Cooja simulation environment.

Fig.7 demonstrates how the EPL statements are organized in order to yield a verdict about the order of ownership relation. Each simple event is decorated with event id and time stamp. After that, all token events are maintained in an ordered event window, which is an Esper EPL specific element that enables managing events in data views. The order of ownership relation is later enforced on that window. Note that, order of event occurrence might differ from order

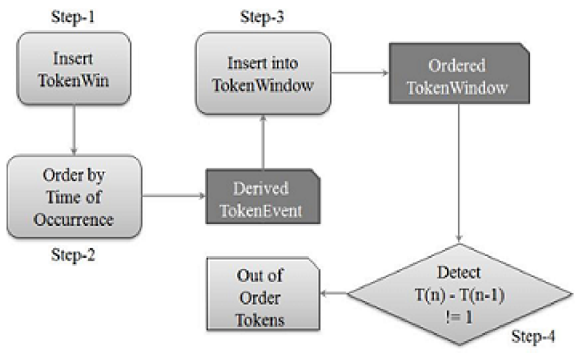


Figure. 7. EPL statement flow for Fig.4.

of event arrival, thus, we must make sure that events are processed with respect to order of event occurrence. That's why we order the token window with respect to order of event occurrence data.

TABLE 3. Predicates and meanings for WTRP.

Predicate	Meaning
TO(m,t)	*m* owns the token at *t*
Diff$(TO(m_i,t_i),TO(m_j,t_j))$	*TO* mote id difference between successive messages from m_i *and* m_j, *True* if 1, *False* otherwise
OO(Diff, t)	*Diff* predicate holds at *t*

The predicates and corresponding meanings for WTRP are given in Table-3. *Diff* (m_i, m_j) is a predicate that is assigned a boolean value depending on the mote id difference between two consecutive token events (i.e., *True* if 1, *False* otherwise). *Diff* is an example of a domain-specific constraint $s(e_i, e_j)$ as we defined in Section-V-B. *OO(Diff,t)* is a predicate function that determines whether or not the order of ownership relations is preserved during successive transmissions at any time t. Thus, having a $OO(Diff, t) \neq 1$ at time t indicates

```
@Name('SUCCESS')
context CtxSample
select count(*) as SuccessVerdict from RVSpec
match_recognize (
 measures A.mId as a_Id
 pattern (A B C D)
 define
  A as A.moteId = m1,
  B as B.moteId = A.moteId + 1,
  C as C.moteId = B.moteId + 1,
  D as D.moteId = C.moteId + 1
) output when terminated and context.endevent.moteId
      = m5;
```

Listing 5. **EPL statement for success in WTRP.**

```
@Name('FAIL-1')
context CtxSample
select count(*) as FailOneVerdict from pattern
 [every
  rsp1 = RVSpec (srcId=2) -> ((rsp2=RVSpec (srcId=3)
     or rsp2=RVSpec (srcId=4)) and not rsp3=RVSpec (
     srcId=6))
 ] output when terminated;
```

Listing 6. **EPL statement for failure in WTRP.**

a failure case. As explained in [6] OO relation must satisfy $HoldAt(OO, t)\ \forall\ t$.

The EPL statements in List.3 can be tailored to reflect event properties specific to the WTRP case, but we can also represent those new predicates (i.e., semantic relations) in EPL statements as shown in List.5. As for the *Fail* cases, we can use the EPL statements shown in List.6, as well as those in List.4. Note that the $Diff(m_i, m_j)$ predicate indicates that $m_j - m_i = 1$ for any consecutive $TO(m_j, t_j)$ and $TO(m_i, t_i)$ predicates where $t_j > t_i \wedge \nexists t_k | t_i < t_k < t_j$. Therefore, the difference between mote ids can also be expressed as $m_j = m_i + 1$.

B. RESULTS AND DISCUSSION

The results of simulation are attained by observing the number of errors logged in Cooja simulation environment and the number of errors captured in Esper CEP engine. The performance of the solution is evaluated by considering the percentage of errors that are successfully captured in Esper.

TABLE 4. **Experiment results.**

Tr.Period	# of CjFaults	# of EspFaults	Performance
20	117	117	100
15	195	193	98.97
10	363	363	100
5	568	568	100
3	993	991	99.79

Table-4 shows the results for various communication scenarios. The scenarios are tailored such that we can observe the performance of our solution approach on different network loads, consequently with increasing numbers of events and errors. In order to achieve such results, we ran each simulation for 10 minutes, in each of which each mote had possessed the token for periods of 3, 5, 10, 15, and 20 seconds, so it can transmit messages. As seen on the Table-4, the performance of our verification approach reaches almost 100% success rate. However, there are two cases where we could not find all the errors in an event trace. We believe that duplicate messages that might occur due to network condition can cause such deviations.

VIII. CONCLUSION

The event calculus (EC) for CoAP-based IoT system interactions is provided in this paper. The EC is a tool for specifying requirements of an IoT system in terms of its expected behavior as sequence of events that occur due to messaging model of CoAP. We further presented a transformation method for mapping EC algebra into Esper EPL statements. The case study demonstrated that once a domain-specific EC algebra is developed, it's straightforward to generate runtime monitors in terms of EPL statements so as to utilize a complex-event processing engine for runtime verification. The EC also will allow us to derive a protocol-specific metamodel that can be used in representing IoT systems with modeling languages such as UML. The MSC approach presented in this paper lays the foundation for our ongoing and future work on model-based testing of IoT systems. We believe that RV verification scenarios including the lower-layer IoT network protocols (i.e. a multi-layer MSC approach) will be of interest to the community.

REFERENCES

[1] R. Medhat, B. Bonakdarpour, D. Kumar, and S. Fischmeister, "Runtime monitoring of cyber-physical systems under timing and memory constraints," *ACM Trans. Embedded Comput. Syst.*, vol. 14, no. 4, 2015, Art. no. 79.

[2] S. Colin and L. Mariani, "Run-time verication," in *Model-Based Testing of Reactive Systems: Advanced Lectures*. Cham, Switzerland: Springer-Verlag, 2005.

[3] E. T. Mueller, "Automating commonsense reasoning using the event calculus," *Commun. ACM*, vol. 52, no. 1, pp. 113–117, 2009.

[4] G. Spanoudakis, C. Kloukinas, and K. Androutsopoulos, "Towards security monitoring patterns," in *Proc. 22nd ACM Annu. Symp. Appl. Comput.*, 2007, pp. 1518–1525.

[5] W. Gaaloul, S. Bhiri, and M. Rouached, "Event-based design and runtime verification of composite service transactional behavior," *IEEE Trans. Serv. Comput.*, vol. 3, no. 1, pp. 32–45, Mar. 2010.

[6] K. Inçki, I. Ari, and H. Sözer, "Runtime verification of IoT system using complex-event processing," in *Proc. IEEE 14th Int. Conf. Netw., Sens., Control (ICNSC)*, May 2017, pp. 625–630.

[7] N. Chen, C. Viho, A. Baire, X. Huang, and J. Zha, "Ensuring interoperability for the Internet of Things: Experience with CoAP protocol testing," *Automatika*, vol. 54, no. 4, pp. 448–458, 2013.

[8] Z. Shelby, K. Hartke, and C. Bormann, *The Constrained Application Protocol (CoAP)*, Standard IETF RFC7272, 2004.

[9] *EsperTech: Complex Event Processing Streaming Analytics*. Accessed: Aug. 28, 2017. [Online]. Available: http://www.espertech.com

[10] K. Yu, Z. Chen, and W. Dong, "A predictive runtime verification framework for cyber-physical systems," in *Proc. IEEE 8th Int. Conf. Softw. Security Rel.-Companion*, Jul. 2014, pp. 223–227.

[11] A. Kane, "Runtime monitoring for safety-critical embedded systems," Ph.D. dissertation, Dept. Elect. Comput. Eng., Carnegie Mellon Univ., Pittsburgh, PA, USA, 2015.

[12] F. Wei, A. Men, X. Zhang, and H. Xiao, "A modified wireless token ring protocol for wireless sensor network," in *Proc. IEEE 2nd Int. Conf. Consum. Electron., Commun. Netw. (CECNet)*, Apr. 2012, pp. 795–799.

[13] L. Lamport, R. Shostak, and M. Pease, "The Byzantine generals problem," *ACM Trans. Programm. Lang. Syst.*, vol. 4, no. 3, pp. 382–401, Jul. 1982.

[14] F. Osterlind, A. Dunkels, J. Eriksson, N. Finne, and T. Voigt, "Cross-level sensor network simulation with Cooja," in *Proc. 1st IEEE Int. Workshop Pract. Issues Building Sens. Netw. Appl. (SenseApp)*, Tampa, FL, USA, Nov. 2006, pp. 641–648.

[15] L. Richardson and S. Ruby, *RESTful Web Services*. Sebastopol, CA, USA: O'Reilly Media, 2007.
[16] *Message Sequence Chart (MSC)*, Standard Rec. ITU-T Z.120, 2011.
[17] E. T. Muller, "Event calculus," *Handbook of Knowledge Representation*. Amsterdam, The Netherlands: Elsevier, 2008, pp. 671–708.
[18] B. Mitchell, "Resolving race conditions in asynchronous partial order scenarios," *IEEE Trans. Softw. Eng.*, vol. 31, no. 9, pp. 767–784, Sep. 2005.
[19] R. Alur, K. Etessami, and M. Yannakakis, "Inference of message sequence charts," *IEEE Trans. Softw. Eng.*, vol. 29, no. 7, pp. 623–633, Jul. 2003.
[20] H. Dan and R. M. Hierons, "Conformance Testing from Message Sequence Charts," *2011 Fourth IEEE Int. Conf. Softw. Test., Verification Validation, Berlin, 2011*, pp. 279–288.
[21] F. Belli, M. Beyazit, and A. Memon, "Testing is an event-centric activity," in *Proc. IEEE 6th Int. Conf. Softw. Security Rel. Companion*, Gaithersburg, MD, USA, Jun. 2012, pp. 198–206.
[22] A. Dunkels, B. Gronvall, and T. Voigt, "Contiki—A lightweight and flexible operating system for tiny networked sensors," in *Proc. IEEE Workshop Embedded Netw. Sens. (Emnets-I)*, Tampa, FL, USA, Nov. 2004, pp. 455–462.

KORAY INCKI received the B.Sc. degree in electrical and electronics engineering from Cukurova University in 1997 and the M.Sc. degree in computer networks from the University of Southern California in 2000. He is currently pursuing the Ph.D. degree in runtime verification of embedded systems with the Computer Science Department, Özyeğin University, under the supervision of Dr. Ari. He has been a software engineer, a senior software engineer, a project manager, and the director in various industries since 2001. After starting his career as a Software Engineer in Silicon Valley, he worked in several indigenous technology research and development projects for the Turkish Defense Industry at TÜBİTAK. His research interests include software verification and validation, model-based testing, complex-event processing, service-oriented architectures, data stream processing, Internet of Things, cloud computing, software engineering, safety-critical systems, and real-time operating systems.

ISMAIL ARI received the Ph.D. degree from the Computer Science Department, University of California at Santa Cruz in 2004. From 2004 to 2009, he was a Researcher with Hewlett Packard Labs, Silicon Valley, CA, USA. His research interests include cloud computing, service-oriented architectures, data mining, data stream processing, complex event processing, and networked storage systems. He has international publications and U.S. patents related to these topics. In 2009, he joined Özyeğin University. He is a member of ACM and a founding member of the IBM Cloud Academy. He has received several awards and research grants, including the IBM Top Faculty Contributor Award, the EU Marie Curie International Reintegration Grant (IRG), and the TÜBİTAK (Turkish NSF) National Young Researcher Career Award.