Runtime Verification of IoT Systems using Complex Event Processing

Koray İnçki[T,ð], İsmail Arı[T], Hasan Sözer[T]
[T]Department of Computer Science
Ozyegin University, İstanbul, Turkey
koray.incki@ozu.edu.tr, {ismail.ari, hasan.sozer}@ozyegin.edu.tr
[ð]Department of Computer Engineering
Adana Science & Tech. University, Adana, Turkey
kincki@adanabtu.edu.tr

Abstract—**Internet of Things (IoT) is a new computing paradigm that is proliferated by wide adoption of application level protocols such as MQTT and CoAP, each of which defines different styles of sequential interaction of events. Even though there is a considerable effort in the literature for verification of such complex and distributed systems, a practical solution for IoT systems that supports runtime system verification is still missing. In this paper, we present a runtime monitoring approach for IoT systems that exploits event relations expressed in terms of sequential interaction messaging model of Constrained Application Protocol (CoAP). We propose the use of Complex-Event Processing (CEP) to detect failures at runtime by exploiting complex event patterns defined via predetermined event algebra. We further present a simple case scenario to demonstrate the applicability of the approach on Wireless Token Ring Protocol execution.**

Keywords—Internet of things; runtime monitoring; verification; CoAP; complex-event processing; event algebra

I. Introduction

In 1999, when IoT phrase was first coined by K. Ashton [1] the computing phenomenon experienced another paradigm shift. However, it took a decade for the champions of computing in academia and industry to discover this new phenomenon. Ashton asserted that "We need to empower computers with their own means of gathering information, so they can see, hear and smell the world for themselves." [1]. In this world, computers and things are to be integrated and interconnected seamlessly so that they both "sense and act" on their environment without requiring any intervention from humans. In recent studies by Fortino, et.al [2,3], devices in IoT are considered to be smart-objects. These objecs are able to sense/actuate, store and interpret information that they generate or they gather from the environment. They interact with each other via several middleware constructs. These constructs mainly follow service-oriented architecture (SOA) [4,5,6] specifications which facilitate collaboration of 'things' [4,7].

Devices employed in IoT systems are generally regarded as resource-constrained devices; therefore, a new application layer protocol, CoAP [7], has been standardized for enabling IoT system architectures with RESTful services [8,9] whilst respecting the resource limitations. Service requests and responses that are implemented according to CoAP messaging model can be described as an interaction of simple Request/Response events, which collectively form complex results to yield desired behavior of the system. There are considerable attempts towards verification of CoAP standard implementations [10], but those studies lack any support for system-level verification.

Verification of heterogenous systems such as IoT is inherently troublesome as it might involve devices from various manufacturers, which complicates the verification process by requiring knowledge of system interface contracts or development details of individual components in the system. We propose a novel and generic approach for verification of IoT systems. We assume that these systems are designed with SOA principles. Our approach involves the analysis of simple events occurring in a CoAP system for composing complex-event patterns to detect failure cases at runtime. We discover complex relations among simple events via consolidating them in a Complex Event Processing (CEP) [11] engine. We use Esper CEP Engine [12] for correlating simple events and synthesizing complex reasoning. The approach does not intervene with the operational system and as such it does not incur any overhead in system communication or it does not alter the system behavior. CEP techniques have already been utilized for verification purposes such as network congestion control and intrusion detection [11,13,14]; however to the best of our knowledge, our study is the first of its kind to utilize CEP techniques for verification of an IoT system. The contributions of this paper are twofold; first, we present a subtle way to articulate CoAP actions with event-calculus; then, we employ CEP techniques to analyze the identified events for runtime verification of IoT systems.

The rest of the paper is organized as follows. Section 2 details related work on runtime verification of service-oriented and embedded IoT systems, while Section 3 presents background on basic event calculus (EC), EC for simple CoAP messages, and explanation of case study with EC. Section 4 describes a case scenario on wireless token ring protocol, while Section 5 gives the implementation details with open-source software such as Esper and Californium. Section 6 describes the experimentation setup and gives the results. Then, we present a discussion on the results in Section 7. Finally, we conclude the paper and present future work in Section 8.

978-1- 5090-4429- 0/17/$31.00 ©2017 IEEE

This research was partially sponsored by TÜBİTAK BİDEB 2211-C Graduate Thesis Support.

II. Related Work

Software is said to have a failure when the observed behavior deviates from the required behavior [15]. Software verification is a process of checking whether the observed behavior of a system meets its predetermined specifications. It aims to lead software quality, which has been studied in major body of recent research papers [16,17,18]. Run-time verification is an approach that differentiates from classic verification (i.e. theorem proving, model checking, and testing) [19] by the fact that it deals with an actual run of a system. Runtime verification techniques rely on special tools, called monitors, which operate over certain execution traces of a system and make decisions on particular correctness properties [19]. A correctness property enables monitors to yield a certain decision of *True/False* depending on the satisfiability of that property. A run of a system is usually expressed in terms of a sequence of system states, which consists of certain variables defining the context in the state. Such an approach for verification can be achieved for those system specifications which can be described by a sequence or a collection of events.

Verification of embedded systems requires delicate effort on resource utilization, because the verification devices and implementations might alter the performance and behavior of the system, thus jeopardizing the whole process. In [20], authors architect a verification solution for embedded motes that run the TinyOS. They propose an approach which necessitates instrumentation of the application to be tested, and requires another verification application to run on the same device, which collects data emitted by the instrumentation code. This approach alters both individual device behavior and system behavior which is composed of such devices. A less intrusive approach, that does not instrument the SUT, is favorable in order not to cause deviation in timing and functional behavior. Therefore, in our approach, we choose to adopt a passive monitor for observing the events in the system.

Certification of products involves testing of the product against certain well-defined scenarios to yield a verdict indicating whether or not it conforms to the standard specifications. For example, CoAP Plugtest events [21] were designed to reveal interoperability issues between different implementations of the CoAP draft [7], and consequently unearth the standard specifications. In [10], Chen, et al. describe their approach to this problem. They propose a verification architecture that uses an open-source packet sniffer (Wireshark) to capture live CoAP network traffic and save it in certain files to work on them later. After the run of the system is completed, the solution approach passively (offline) tries to verify compliance of the implementation to the standard specification. This research does not allow for testing the application at runtime (online); and their approach merely deals with protocol compliance testing, which means that one cannot verify a system that employs CoAP as a communication model by using this approach.

Competitive approaches [10,22,23] for verification of IoT systems, either necessitate a certain amount of intervention with the application code or provide offline testing techniques. Another solution is devised solely for protocol testing [10], meaning that it does not address application testing of a system that employs CoAP as a messaging model. On the contrary, our approach neither requires an intervention with the application code, nor operates on historical records of a run of a system; while enabling a system test capability. In [22], Cubo, et al. attempt to identify an approach for verification of Web of Things by using CEP. Their claimed architecture lacks any descriptive details; moreover, they did not implement their proposed solution.

III. BACKGROUND

A. *Basic Event Calculus*

In real world, we use natural language assertions to indicate an occurrence of an event. For example, ‘The weather turned rainy’ sentence implies that the weather has changed state from ‘not raining’ to ‘raining’. Information systems can also be specified in terms of events occurring in the system. Event calculus, first introduced by Kowalski and Sergot [24], is a method of representing occurrences and actions with respect to temporal relations. Even though it was initially used for understanding database transactions, it can also be used for program specifications [24]. Since its proposition, there have been several attempts to extend its representational expressiveness. As Mueller states [25], event calculus allows a native computing paradigm in concurrent events, continuous time, events with duration, partially ordered events, and triggered events. Kowalski [26] proposed a simplified version of event calculus (SEC) that replaced the event occurrences with event types. Table-1 presents some of the predicates of SEC. *Fluents (f)* allow for representing time-varying properties of a system.

TABLE I. A SET OF SEC PREDICATES [26]

Predicate	Meaning
Initially(f)	*f is True at timepoint 0*
HoldsAt(f,t)	*f is True at t*
Happens(e,t)	*e (event) occurs at t*
Initiates(e, f, t)	*if e occurs at t, then f is true after t*
Terminates(e, f, t)	*if e occurs at t, then f is false after t*

B. *Simple Event Calculus (SEC) for CoAP*

From a system (network) viewpoint sending a request message and sending a response message in a CoAP application both constitute an event. The motes in a CoAP network behaves either as a client or a server [7].

Each method call (GET, POST, PUT, DELETE) in a request message requires a corresponding response message, where each method call represents a different event type. For the remainder of the paper, let us assume that e_1 represent a *send_request_event* from a client, and e_2 represent a *send_response_event* from a server, respectively. The temporal ordering between e_1 and e_2 ($e_1 \preccurlyeq e_2$) pairs can be described with the following axiom by using SEC:

$$Follows(e_1, t_1, e_2, t_2) \equiv \exists\ e_1, e_2, t_1, t_2\ (Happens(e_1, t_1) \wedge Happens(e_2, t_2) \wedge (t_1 < t_2)) \quad (1)$$

Hereby, e_1 is either of GET, PUT, POST, DELETE and e_2 is any valid response code. Any time-varying system property (fluent) that relies on sequential-ordering of messages can be represented by this predicate.

Based on SEC, a *fluent (f)* that is initiated with the occurrence of an event will hold *True* until occurrence of a terminating event.

$$HoldsAt(f, t) \equiv \exists\, e_1, e_2, t\ (Initiates(e_1, f, t_1) \wedge Follows(e_1, t_1, e_2, t_2) \wedge Terminates(e_2, f, t_2) \wedge (t_1 < t) \wedge (t < t_2)). \qquad (2)$$

Hereby, f can be any system specific time-varying property.

C. Complex-Event Processing (CEP)

CEP is a technique that was initially introduced for deducing complex decisions in business processes [11]. CEP tools enable us to make high-level decisions about event-driven systems by inferring complex meanings out of simple events in various domains [13,14]. Simple events can be consolidated into complex events through several transformations such as threshold-based filtering, joining, sequencing, and well-known aggregation functions.

Any software system with distributed function calls can also be described as a collection of events [27]. Interactions between components of an IoT system are also events, as we will see in Section 6. Simple event logs can be synthesized to deduce information about runtime behavior of a system compared to its expected behavior. Every service request and response that occurs according to CoAP primitive messaging methods causes various chains of events.

IV. A Protocol Case Study: WTRP

Wireless Token Ring Protocol (WTRP) is a MAC-layer Protocol that is frequently used in WSN, where the network is demanded to achieve self-healing, self-organization and no-center features [28]. Its inherent characteristics make it suitable for providing quality of service in terms of bounded latency and reserved bandwidth, which are quite important for real-time applications [29]. Improvements introduced in [30] promote dynamic ad-hoc network structure expansion, energy saving and transport efficiency enhancements, which are valuable attributes for CoRE (Constrained RESTful Environments) [7] devices as employed in IoT. Token ring sequence order might be broken due to several reasons (e.g., battery of wireless node drains, mote moves out of scope). The WTRP is designed to recover such situations by requiring each node to implement certain algorithms in the protocol stack.

Assuming that the network in Fig. 1 is a token-ring network, we can represent the effects of the request/response events by using following event calculus (For the sake of simplicity, token is assumed to passed between motes in order of mote id, i.e., first mote 1, then mote 2 and so on).

TO(m, t) is a predicate fluent function that determines which mote owns the token at time t. For example, if m_1 possesses the token at time t_1, then $TO(m_1, t_1)$ will be True, otherwise False.

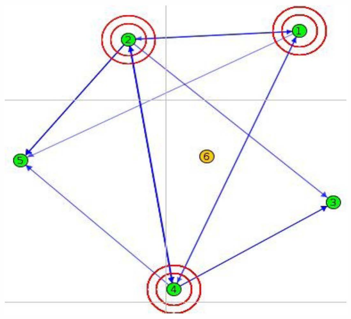


Fig. 1. WTRP Network Implemented on Cooja

$$Initially(TO(m, 0)) \text{ is True for where } m \equiv 1. \qquad (3)$$

Send(m, t) describes an event where mote m sends a network message and also passes token to the next mote in the topology.

OO(t) is a predicate fluent function that enables monitoring order of ownership of the token. Based on the initial assumption that token is passed among motes in order of increasing mote ids *OO(t)* must always be 1.

$$OO(t) \equiv 1, \forall t. \qquad (4)$$

$$HoldsAt(OO(t), t) \equiv \forall\, m_1, m_2\ \exists\, t_1, t_2\ (Happens(Send(m_1, t_1), t_1) \wedge Initiates(Send(m_1, t_1), TO(m_2, t_1), t_1) \wedge Terminates(Send(m_1, t_1), TO(m_1, t_1), t_1) \wedge (m_2 - m_1 = 1) \wedge Happens(Send(m_2, t_2), t_2) \wedge (t_2 > t_1)). \qquad (5)$$

During the normal operation of WTRP, the last equation must always hold. Thus, by monitoring the validity of *HoldsAt(OO(t), t)* predicate, we can make sure that token-ring protocol is running according to its specification. These predicate functions, i.e. TO, OO, are *fluents* of WTRP.

V. Architecture

A. CoAP Event Generator

Our runtime monitoring approach necessitates clearly determining the faulty behavior of a system, and expressing each case in the language of CEP engine, called the Event Processing Language (EPL). The solution architecture shown in Fig. 2, is composed of a network packet listener+parser and a CEP engine. The CoAP parser is a non-intrusive network listener+parser that listens (implemented by Java class Parser in Fig. 3) to a specified IPv6 network [31] interface for all exchanged packets on that network, filtering only CoAP packets, and finally generating simple events based on captured CoAP packets. A simple event consists of the main descriptive parameters of a CoAP message [7]. Such events are then sent to a CEP engine for detection of abnormal situations.

Our CoAP parser (Fig. 3) utilizes an open-source Java network packet capture library, named JPcap [32], for intercepting packets exchanged on an interface. Basic events are produced by CoAP Parser by utilizing Californium open-source library [33]. Californium is an implementation of CoAP protocol stack in Java, which allows for writing RESTful applications based on CoAP protocol. We inherited raw packet processing parts of this library, and added new features for producing simple CoAP Events.

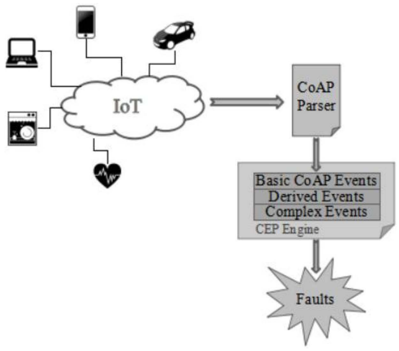
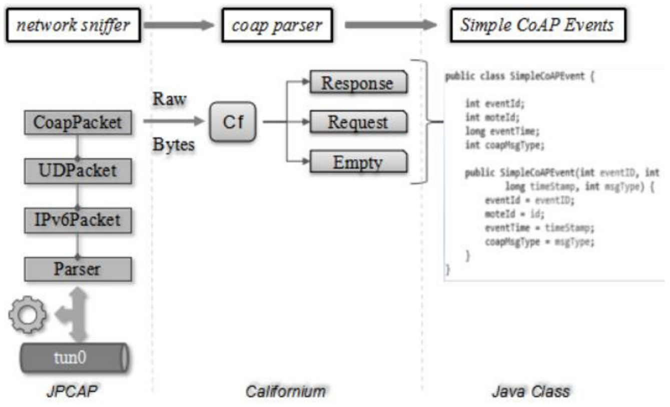


Fig. 2. Event Generation out of CoAP Messages

The architecture for event generator in Fig. 3 does not strictly depend on the open-source libraries used in the case study implementation; thus, any variant of the same architecture, which might involve any other open-source libraries or proprietary implementations for sniffing CoAP packets other than JPcap and Californium can be easily carried out, provided that they generate proper CoAP Events for the Esper engine. It consists of two main components, a *network sniffer* and a *CoAP parser. Network sniffer* must be able to capture CoAP Packets passively. Raw packets captured from network interface (*tun0*) are handled by *Parser*, which implements *RawPacketListener* Interface in JPcap library. It further instantiates instances of *IPv6Packet*, *UDPacket*, and *CoapPacket* classes for parsing CoAP messages from incoming raw packets. Then, it relays those to the *coap parser*; which parses each packet and classifies them as either *Request, Response, or Empty* message [7]. Afterwards, it generates a unique instance of *SimpleEventClass*. An instance of *SimpleEventClass* must contain a unique identifier for each event (*eventId*), a timestamp for each event (*eventTime*), an identifier for the owner of event (*moteId*) and the message type field (*coapMsgType*). We preferred JPcap and Californium open source libraries for implementation of *network sniffer* and *coap parser*, respectively, for their wide community support. Any other implementation that generates aforementioned *SimpleCoapEvent* class instances with other libraries can easily be developed.



```
public class SimpleCoAPEvent {

    int eventId;
    int moteId;
    long eventTime;
    int coapMsgType;

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

Fig. 3. CoAP Event Generator Architecture

VI. Experiment

The motes in our experiment run on Contiki-OS, a real-time operating system (RTOS) for IoT devices. Contiki-OS is an open-source RTOS [34], which comes with implementation of many protocols such as CoAP and a simulation environment, called Cooja [35]. Cooja is not only a wireless network simulation environment, but also an emulation environment that exhibits instruction level demonstration of user applications for several embedded platforms (e.g., Zolertia).

We demonstrate a token ring network scenario among Zolertia-Z1 motes in Cooja. The experiment is set-up as shown in Fig, 1. Motes 1 thru 5 are regular motes and Mote-6 is a border router. According to the scenario, each mote has to possess a token to send a broadcast message to the network. The motes in the simulation are engaged in broadcast communication with all the other motes in range. If a message is received by any mote that comes from a mote whose id violates *HoldsAt(OO(t), t)* predicate, then it is an indication of a fault in the WTRP. In order to demonstrate such faulty conditions, we added random seeded errors in the source code of the application that runs on motes so that a mote randomly decides to communicate with other parties without possessing the token. Mote 6 (border router) does not participate in the token ring communication. A border router is utilized in order to setup a connection between the simulation environment and host platform through a serial-line internet protocol interface. The border router passively listens to network traffic in the token ring network, and relays all intercepted messages to the host platform using a CoAP client-server connection. We included one client and one server in our scenario in order to demonstrate the capability of passively verifying a network of non-CoAP endpoints through a CoAP-installed border router.

The predicate function, *HoldsAt(OO(t),t),* evaluates to True if and only if $m2 - m1 = 1$ (with one exception, when token is passed from mote-N to mote-1, in a network of N motes). Thus, if we can detect cases where $m2 - m1 \neq 1$ by monitoring the broadcast messages of motes, then we can create a complex-event identifying a failure situation. So, in CEP engine, we should be looking for correctness of $\neg$*HoldsAt(OO(t),t)* function.

We used Esper CEP engine [12] to derive complex events out of basic CoAP events and detect failure situations. Esper is an open-source CEP engine which supports development in Java and C# programming languages. It provides a particular grammar and language, called EPL, which allows introducing event definitions and correlations among those events inside the CEP engine.

Fig. 4 shows event-processing steps that we perform in Esper. Each CoAP Message intercepted by *CoapParser* is injected into Esper as a new *TokenEvent* that is extension of *CoapEvent* class. As we stated earlier, the events received from the network might be out of order, thus we order all *TokenEvent* events with respect to their time of occurrence properties in Esper (Step-2) in order to avoid false alarms.

Esper maintains events in streams, and EPL statements (Table-2) are exerted on stream of events to retrieve special relations or properties in those streams.

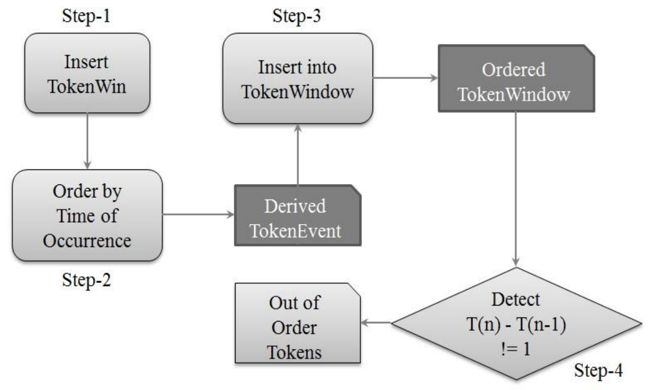


Fig. 4. Event Flow for Out-of-Order Token Detection in CEP Engine

As Esper "select" statement selects events from a stream, those events are removed from the stream and update listeners are notified. But, in order to correlate sequential events with respect to their timeOfOccurrence properties, we need to retain those events. Esper provides a particular data construct, called window, that help us to decorate, maintain, and store events until it is undefined or the events are explicitly discarded. Ordered TokenEvent events are inserted into the OrderedUniqueTokenWin. The pattern for detecting out-of-order token processing is executed on the OrderedUniqueTokenWin events. This last step generates complex-events that make *¬HoldsAt(OO(t),t) = True.*

TABLE II. ESPER EPL STATEMENTS

	EPL Statements
1	*create window OrderedUniqueTokenWin.std:unique(eventId). win:keepall() as select eventId, moteId, eventTime from TokenEvent*
2	*insert into OrderedUniqueTokenWin select eventId, moteId, eventTime from TokenEvent order by eventTime*
3	*select a.eventId as aEvId, a.moteId as aId, b.moteId as bId, b.eventId as bEvId from pattern [every a=OrderedUniqueTokenWin -> b=OrderedUniqueTokenWin] where (((b.moteId - a.moteId) != 1))*

VII. DISCUSSION

Our performance indicator for the approach is the ratio of number of detected events in Esper to the number of logged events in Cooja. In this experiment, the token was passed around "randomly" among motes every period, instead of following a certain order. Simulation scenario allows each mote to broadcast a certain time in varying periods at each run of the scenario, which are selected as 3, 5, 10, 15, and 20 seconds, respectively; and each simulation takes 10 minutes to complete. This enabled us to observe the robustness of our solution under varying loads of events (Table-3).

Note that the motes simulated in Cooja for this research are examples of embedded devices. We particularly implemented the experiment on Z1 motes as they appear in Cooja. We preferred Zolertia type motes due to their available RAM capacity for embedding CoAP client and server codes.

TABLE III. PERFORMANCE RESULTS

Pr (s)	**#CjE**	**#EspE**	**#CjF**	**#EspF**	**Prf(%)**
3	1021	1020	993	991	99,79
5	579	579	568	568	100
10	367	367	363	363	100
15	198	197	195	193	98,97
20	128	128	117	117	100

#CjE column shows total number of events produced in Cooja environment, while #EspE indicates number of simple TokenEvent events inserted in Esper. #CjF shows total number of token ring protocol failures occurred in Cooja, and #EspF shows total number of violations detected in Esper. Pr column values shows the broadcast period allowed for each mote, during a scenario. As expected, more events are generated for smaller transmission periods. Performance of our approach is evaluated by the ratio of #EspF / #CjF. The results justify the validity of our approach by a correctness factor of nearly 100% in all cases. We observed that most of the failures generated in Cooja are detected in Esper. We think that undetected errors are missed due to network packet disorders, but those errors deserve further investigation as future work.

The design of CoAPParser supports listening to raw CoAP communication between any numbers of motes, thus our solution can be generalized for different scenarios. This approach can be extended to verify IoT systems that utilize MQTT messaging model and others, provided that those models are expressed with proper Event Calculus as proposed here.

VIII. CONCLUSION

In this study, we devised a novel solution for non-intrusive, non-instrumented, and online runtime verification of IoT systems. Our approach is an event-based solution which exploits RESTful service paradigm employed in CoAP messaging model. IoT systems designed with CoAP model are represented as event-driven systems; and consequently, we demonstrated how to leverage CEP techniques for verification of such a system. Our study not only provides an architectural solution, but also involves integration of several open-source tools.

We believe that deriving ontology of events occurring in IoT domain with respect to communication models is necessary to guide research in this domain. Our event descriptions were extracted by textual representations of simple events and their correlations.

REFERENCES

[1] K. Ashton, (2009, Jun.). Internet of things. RFID J. [Online]. Available: http://www.rfidjournal.com/articles/view?4986

[2] G. Fortino, P. Trunfio, "Internet of Things Based on Smart Objects, Technology, Middleware and Applications". Springer 2014, ISBN 978-3-319-00490-7

[3] G. Fortino, A. Guerrieri, W. Russo, C. Savaglio, "Integration of agent-based and Cloud Computing for the smart objects-oriented IoT". CSCWD 2014: 493-498

[4] T. Teixeira, S. Hachem, V. Issarny, and N. Georgantas, "Service oriented middleware for the Internet of Things: a perspective". Springer LNCS vol. 6994, pp. 220-229, 2011

[5] R. T. Fielding and R. N. Taylor, "Principled design of the modern web architecture". ACM Trans. on Internet Technology, vol.2, issue 2, pp.115-150, 2002

[6] D. Guinard, V. Trifa, S. Karnouskos, P. Spiess, and D. Savio, "Interacting with the SOA-Based Internet of Things: Discovery, Query, Selection, and On-Demand Provisioning of Web Services". IEEE Trans. on Services Computing, 2010 vol.3, pp. 223-235

[7] Z. Shelby, K. Hartke, and C. Bormann, "The constrained application protocol (CoAP)". IETF RFC-7252, June 2014

[8] F. Belqasmi, R. Glitho, and C. Fu, "RESTful web services for service provisioning in next-generation networks: a survey". IEEE Communications Mag., pp.66-73, 2011

[9] W. Qin, Q. Li, L. Sun and H. Zhu, "RestThing: a restful web service infrastructure for mash-up physical and web resources". IEEE/IFIP 9th Int. Conf. on Embedded and Ubiquitous Comp., Melbourne, Australia, October 24-26, 2011

[10] N. Chen, C. Viho, A. Baire, X. Huang, and J. Zha, "Ensuring interoperability for the Internet of Things: experience with CoAP protocol testing". Automatika J. for Control, Measurement, Electronics, Computing and Communications, vol.54 n.4, 2013

[11] D. Luckham, "The Power of Events: An Introduction to Complex Event Processing in Distributed Enterprise Systems". Addison-Wesley, Boston, MA, USA, 2001

[12] http://www.espertech.com/esper/ (Last accessed on 07/05/2016).

[13] T. Takahashi, H. Yamamoto, N. Fukumoto, S. Ano, and K. Yamazaki, "Complex Event Processing to Detect Congestions in Mobile Network". IEEE 16th International Conference on Advanced Communication Technology, Pyeongchang, 2014

[14] C.Y. Chen, J.H. Fu, T. Sung, P.F. Wang, E. Jou, and M. Feng, "Complex Event Processing for the Internet of Things and its Applications". IEEE International Conference on Automation Science and Engineering (CASE). Taipei, 2014

[15] N. Delgado, A.Q. Gates, and S. Roach, "A taxonomy and catalog of runtime software-fault monitoring tools". IEEE Trans. Softw. Eng., vol. 30, issue 12, pp. 859-872, 2004

[16] M. Sahinoglu, K. Incki, M.S. Aktas. "Mobile Application Verification: A Systematic Mapping Study", LNCS Book Chapter, LNCS 9155-9159 for ICCSA 2015.

[17] M.S. Aktas, M. Kapdan, "Structural Code Clone Detection Methodology Using Software Metrics". International Journal of Software Engineering and Knowledge Engineering 26(2): 307-332 (2016)

[18] M. Kapdan, M.S. Aktas, M. Yigit, "On the Structural Code Clone Detection Problem: A Survey and Software Metric Based Approach". LNCS Book Chapter for ICCSA 2014, Vol 8583, pp 492-507

[19] M. Leucker, and C. Schallhart, "A brief account of runtime verification". Elsevier The Jrn. of Logic and Algebraic Programming, vol.78, issue 5, pp.293-303, 2009

[20] D. Bucur, "Temporal monitors for TinyOS". Springer Lecture Notes in Computer Science, vol. 7687, pp.96-109, 2012

[21] ETSI CoAP Interoperability Tests, PlugTests http://www.etsi.org/plugtests/coap/coap.htm, (Last Accessed on 13/05/2016)

[22] J. Cubo, L. González, A. Brogi, E. Pimentel, and R. Ruggia, "Towards Run-Time Verification of Compositions in the Web of Things using Complex Event Processing". In IX Jornadas de Ciencia e Ingeniera de Servicios (JCIS), 2013, Madrid, Spain

[23] C. Watterson, and Heffernan, "Runtime verification and monitoring of embedded systems". IET Software, vol.1, issue 5, 2007

[24] R. Kowalski, and M. Sergot, "A logic-based calculus of events". New Generation Computing, v.4 n.1, pp.67-95, Japan, 1986.

[25] E. T. Mueller, "Event calculus". In Handbook of Knowledge Representation (pp. 671-708). Amsterdam: Elsevier, 2008

[26] R.A. Kowalski, "Database Updates in the Event Calculus". Journal of Logic Programming, vol. 12 (1992), pp. 121–146.

[27] S. Qadeer, and S. Tasiran, "Runtime verification of concurrency-specific correctness criteria". Springer J. Software Tools Technol. Transfer. 2012

[28] F. Wei, X. Zhang, and H. Xiao, "A modified wireless token ring protocol for wireless sensor network". IEEE 2nd Int. Conf. on Consumer Electronics, Communications and Networks (CECNet), China, 2012

[29] D. Lee, R. Attias, A. Puri, R. Sengupta, S. Tripakis, and P. Varaiya, "A wireless token ring protocol for intelligent transportation systems". IEEE Proceedings of 2001Intelligent Transportation Systems. Oakland, CA, 2001

[30] M. Ergen, D. Lee, R. Sengupta, and P. Varaiya, "WTRP - wireless token ring protocol". IEEE Trans. Veh. Technol., vol. 53, no. 6, pp. 1863-1881, 2004

[31] S. Deering and R. Hinden, "Internet Protocol, Version 6 (IPv6) Specification". IETF RFC-2460, 1998

[32] http://jpcap.sourceforge.net/ (Last accessed on 05/05/2016).

[33] M. Kovatsch, M. Lanter, and Z. Shelby, "Californium: scalable cloud services for the Internet of Things with CoAP". In Proceedings of the 4th International Conference on the Internet of Things, 2014

[34] A. Dunkels, B. Gronvall, and T. Voigt, "Contiki - a lightweight and flexible operating system for tiny networked sensors". IEEE Workshop on Embedded Networked Sensors (Emnets-I), Tampa, Florida, USA, November 2004.

[35] F. Osterlind, A. Dunkels, J. Eriksson, N. Finne, and T. Voigt, "Cross-level sensor network simulation with Cooja". In Proceedings of the First IEEE International Workshop on Practical Issues in Building Sensor Network Applications (SenseApp 2006), Tampa, Florida, USA, November 2006.