# RUNTIME VERIFICATION OF INTERNET OF THINGS USING COMPLEX-EVENT PROCESSING (RECEP)

A Dissertation

by

Koray İnçki

Submitted to the
Graduate School of Sciences and Engineering
In Partial Fulfillment of the Requirements for
the Degree of

Doctor of Philosophy

in the
Department of Computer Science

Özyeğin University
June 2018

Copyright © 2018 by Koray İnçki

# RUNTIME VERIFICATION OF INTERNET OF THINGS USING COMPLEX-EVENT PROCESSING (RECEP)

Approved by:

Asst. Prof. Dr. İsmail Arı, Advisor
Department of Computer Science
*Özyeğin University*

Prof. Dr. Şebnem Baydere
Department of Computer Engineering
*Yeditepe University*

Assoc. Prof. Dr. Prof. Hasan Sözer
Department of Computer Science
*Özyeğin University*

Asst. Prof. Dr. Barış Aktemur
Department of Computer Science
*Özyeğin University*

Date Approved: 18 May 2018

Assoc. Prof. Dr. Mehmet S. Aktaş
Department of Computer Engineering
*Yıldız Teknik University*

*To loving memory of my father, Recep İnçki, who is the role model for me that it is never too late to learn...*

# ABSTRACT

Increase in the computing power and memory accompanied with decreasing architectural footprints has enabled conquering new frontiers in proliferation of technology in the next industry revolution. More autonomous systems have been deployed thanks to the advancing capabilities provided by embedded systems with such computing power. Internet of Things (IoT) has emerged as an enabler of many achievements in the industry through presenting a seamless integration of computing units, usually in the form of an embedded system, by allowing interconnection of such embedded systems without requiring human interaction. Engineering a system of systems (SoS) constituted by IoT devices has been the new challenge of designing large scale systems, as the scale of such a system could range from tens of devices in an ambient assisted living (AAL) example to thousands of devices in a smart city application. Therefore, the complexity of software engineering and verification of those SoS's necessitates new approaches that would facilitate those processes. In this thesis, we tackle the problem of verifying IoT SoS's at runtime. We first propose an event calculus that captures the fundamental behavioral model of IoT messaging primitives. The event calculus allows us to specify interaction of IoT devices in terms of events that represent sending and receiving Constrained-Application Protocol (CoAP) messages. Representing the behavior of CoAP endpoints in EC helps us define complex-event processing (CEP) patterns that will later be used as runtime monitors. Existing research on runtime verification (RV) usually presents a solution with heavy formal methods, which hinders the usefulness of method by intimidating the practitioners. We, therefore, propose a model-driven engineering (MDE) approach for RV of IoT systems, which is expected to promote the utilization of RV in industrial scenarios. We propose an extension to

the UML2.5 profile, which enables us to customize a modeling tool so that we can develop a domain-specific model (DSM) for verifying IoT systems. Later, in order to allow automatically generating runtime monitors in the form of CEP statements, we contribute a model-to-text (M2T) transformation utility in the modeling tool. The contributions of the thesis are demonstrated in several case scenarios.

# ÖZETÇE

Bilgisayar teknolojilerindeki artan işlemci gücü ve bellek kapasitesi, bununla birlikte giderek azalan ölçekli mimari boyutları sayesinde endüstride yeni bir devir başladı, Endüstri 4.0. Bu gelişmeler gömülü sistemlerin kapasitesini artırarak, bunların kullanıldığı otonom sistemlerin her geçen gün daha fazla yaygınlaşmasını sağlamıştır. Nesnelerin İnterneti (Internet of Things - IoT) gömülü sistemlerin insan etkileşimine gereksinim duymadan birbiriyle etkileşebilmesini sağlayarak, endüstride pek çok yeniliğin başarılmasına yol açmıştır. IoT cihazlarından oluşan bir sistemlerin sistemi (SoS) geliştirmek büyük ölçekli sistem tasarımında yeni bir zorluk olarak karşımıza çıkmaktadır (örn., ortam destekli yaşama (ambient assisted living - AAL) uygulamasında onlarca cihaz kullanılırken, akıllı şehir uygulamalarında binlerce cihaz kullanılabilir). Bu yüzden, IoT cihazlarından oluşan SoS'lerin yazılımı geliştirilmesi ve doğrulama süreçlerinde karşılaşılan zorlukların üstesinden gelebilmek için yeni yaklaşımlara ihtiyaç vardır. Biz bu tezde IoT cihazlarından oluşan SoS'lerin koşum zamanı doğrulamasının yapılabilmesi problemini ele aldık. Öncelikle, IoT mesajlaşma öğelerinin temel davranış modelini tanımlayan bir olay kalkülüsü (event calculus - EC) öneriyoruz. EC, bize IoT cihazları arasındaki Constrained-Application Protocol (CoAP) mesajı gönderme ve alma şeklinde gerçekleşen haberleşme aksiyonlarını olaylar türünden tanımlayabilmemizi, dolayısıyla CoAP uç noktaları davranışlarını, daha sonra koşum zamanı gözlemcileri olarak kullanacağımız, karmaşık olay işleme (complex-event processing - CEP) şablonları tanımlamamıza imkan sağlıyor. Koşum zamanı doğrulama (runtime verification - RV) alanındaki mevcut çalışmalar genellikle ağır formal yöntemler içeren çözümler önermektedir; bu nedenle, RV endüstride pek yaygın kullanılmamaktadır. Bu problemi de dikkate alarak biz bu araştırmada,

model-güdümlü mühendislik (model-driven engineering - MDE) yaklaşımlarını kullanarak IoT sistemlerinin RV faaliyetleri için formal yöntemlere kıyasla daha kullanılabilir bir çözüm sunduk. Bizim yaklaşımımızda, UML2.5 profilinde IoT alanına özgü değişiklikler yaparak, alana-özgü modelleme (domain-specific modelling - DSM) prensibine dayalı bir MDE çözümü sunulmuştur. Ayrıca, IoT için önerilen DSM kullanılarak davranış modellerinden CEP ifadeleri biçiminde koşum zamanı gözlemcilerini otomatik olarak üretebilmek icin modelden-yazıya dönüşüm (model-to-text - M2T) tekniği ile yeni algoritmalar geliştirilmiştir. Tezde önerilen katkıların gösterimi için MDE ve M2T teknikleri çeşitli durum çalısmalarında kullanılmıştır.

# ACKNOWLEDGEMENTS

Pursuing a PhD degree was a long-lasting goal of my life, because I believe in life-long learning and research. But, I never had the real motivation and time to finish one, until I met with Prof. Ismail Arı. I sincerely thank Prof. Ismail Ari for encouraging me to pursue a PhD degree in Computer Science Department of Özyeğin University. From the very beginning, he guided me towards a thesis subject that has an application-oriented system solution to a common industrial problem. Moreover, the collaborative work ethic fostered in the Computer Science Department had positive impetus on landing a state-of-the-art the thesis subject that is a candidate of having a permanent impact in the literature. Prof. Arı and Prof. Sözer has cooperated in inspiring such a novel research question that has driven my PhD dissertation. Especially during the first semester of my thesis, Prof. Sozer has demonstrated a very keen virtue of selflessness and helped us comprehend the principles of RV domain. I am also thankful to Prof. Mehmet S. Aktaş for his insightful remarks and intriguing questions on the contributions of the thesis at during progress review meetings.

I also would like to thank my wife, Merve. She has always showed compassion and patience towards me, whenever I felt like going nuts because of a software not working properly, or not being able to finalize a paper on time. She had never complained about me stealing time from our marriage, for endless hours some nights. Thank you, Merve!

# TABLE OF CONTENTS

**DEDICATION** . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . **iii**

**ABSTRACT** . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . **iv**

**ÖZETÇE** . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . **vi**

**ACKNOWLEDGEMENTS** . . . . . . . . . . . . . . . . . . . . . . . . . . . **viii**

**LIST OF TABLES** . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . **xii**

**LIST OF FIGURES** . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . **xiii**

**I INTRODUCTION** . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . **1**

1.1 Thesis Scope . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 2

1.2 Motivation . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 4

1.3 Contributions . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 5

1.4 Thesis Overview . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 7

**II BACKGROUND** . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . **9**

2.1 Runtime Verification . . . . . . . . . . . . . . . . . . . . . . . . . . . 9

2.2 CoAP-based IoT Systems . . . . . . . . . . . . . . . . . . . . . . . . 11

2.2.1 A Short Investigation of Lamport's Timestamps on CoAP . . 15

2.3 Verification as a Service: Gaps and Opportunities . . . . . . . . . . 16

2.3.1 Cloud Computing . . . . . . . . . . . . . . . . . . . . . . . . 18

2.3.2 Software Testing and Virtualization . . . . . . . . . . . . . . 21

2.3.3 Research Methodology . . . . . . . . . . . . . . . . . . . . . 21

2.3.4 Evaluation . . . . . . . . . . . . . . . . . . . . . . . . . . . . 23

2.4 Related Work . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 29

2.5 Conclusion . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 29

**III DERIVING AN EVENT CALCULUS FOR IOT** . . . . . . . . . . . **31**

3.1 Event Calculus Revisited . . . . . . . . . . . . . . . . . . . . . . . . . 33

3.2 Representing IoT with Events: Leveraging MSCs . . . . . . . . . . . 36

3.3 Event Calculus for IoT . . . . . . . . . . 40
3.3.1 Event Calculus for CoAP . . . . . . . . . . 41
3.4 Case Study: Wireless Token Ring Protocol . . . . . . . . . . 46
3.5 Related Work . . . . . . . . . . 49
3.6 Conclusion . . . . . . . . . . 51

**IV RUNTIME VERIFICATION OF IOT SYSTEMS USING CEP (RE-CEP) . . . . . . . . . . 53**

4.1 Complex-Event Processing with Esper . . . . . . . . . . 55
4.2 Event Processing for Runtime Verification . . . . . . . . . . 58
4.3 CEP for IoT . . . . . . . . . . 64
4.3.1 CoAP Event Generator . . . . . . . . . . 64
4.4 Case Study: WTRP . . . . . . . . . . 66
4.4.1 Implementation . . . . . . . . . . 69
4.5 Discussion . . . . . . . . . . 71
4.6 Related Work . . . . . . . . . . 73
4.7 Conclusion . . . . . . . . . . 75

**V RUNTIME VERIFICATION AT THE EDGE OF THINGS . . . 76**

5.1 Modeling Runtime Verification for IoT Systems . . . . . . . . . . 78
5.1.1 UML Profile Extension . . . . . . . . . . 79
5.1.2 Model-driven engineering for Interoperability . . . . . . . . . . 80
5.1.3 A model-based RV solution for IoT systems . . . . . . . . . . 82
5.1.4 Implementation . . . . . . . . . . 84
5.2 Verification at the Edge of Things . . . . . . . . . . 87
5.2.1 A Reference Verification Architecture . . . . . . . . . . 90
5.3 Case Study: Ambient-Assisted Living . . . . . . . . . . 92
5.4 Discussion . . . . . . . . . . 96
5.5 Related Work . . . . . . . . . . 98
5.6 Conclusion . . . . . . . . . . 101

**VI CONCLUSION** . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . **103**

**REFERENCES** . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . **107**

**VITA** . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . **119**

# LIST OF TABLES

1 Categorization of Literature Based on Test Level & Type . . . . . . . 24
2 Categorization of Literature Based on Contribution & Delivery Model 25
3 SEC Predicates and Meaning . . . . . . . . . . . . . . . . . . . . . . . 35
4 Context and Event Verdicts for Figure.12 . . . . . . . . . . . . . . . . 46
5 Predicates and Meanings for WTRP . . . . . . . . . . . . . . . . . . . 70
6 Performance Results . . . . . . . . . . . . . . . . . . . . . . . . . . . 72
7 RV@Edge Architecture Components of Figure.28 . . . . . . . . . . . . 91

# LIST OF FIGURES


1 Examples of different scales of IoT deployment[1] . . . . . . . . . . . 2
2 Estimated number of IoT devices installed between 2016-2020 . . . . 3
3 Contributions of the Thesis . . . . . . . . . . . . . . . . . . . . . . 6
4 Verification Techniques . . . . . . . . . . . . . . . . . . . . . . . . 10
5 Runtime Verification Recapped . . . . . . . . . . . . . . . . . . . . . 11
6 CoAP OSI Layering . . . . . . . . . . . . . . . . . . . . . . . . . . 12
7 CoAP Message Format[2] . . . . . . . . . . . . . . . . . . . . . . . . 13
8 Demonstration of CoAP Messaging Model [2] . . . . . . . . . . . . . 14
9 Development and Test Life-Cycle . . . . . . . . . . . . . . . . . . . . 18
10 Cloud Deployment & Delivery Models . . . . . . . . . . . . . . . . . 19
11 How Event Calculus Works . . . . . . . . . . . . . . . . . . . . . . . 34
12 MSC for a CoAP Service $S$ . . . . . . . . . . . . . . . . . . . . . . 37
13 Cooja Simulation of WTRP . . . . . . . . . . . . . . . . . . . . . . . 47
14 Complex-Event Processing Conceptual View . . . . . . . . . . . . . . 56
15 Esper is a container for EPL Statements[81] . . . . . . . . . . . . . . 57
16 RV Process with Complex-Event Processing . . . . . . . . . . . . . . 59
17 CEP Assisted Runtime Verification Reference Architecture . . . . . . 63
18 Event Generation out of CoAP Messages . . . . . . . . . . . . . . . . 65
19 EPL Statement Flow for Figure.13 . . . . . . . . . . . . . . . . . . . 68
20 MDE Process for Interoperability . . . . . . . . . . . . . . . . . . . . 81
21 UML Elements Syntactical Relations (*adapted from* [3]) . . . . . . . . 82
22 CoAP UML Profile Extension . . . . . . . . . . . . . . . . . . . . . . 83
23 Healthcare Interoperability[4] . . . . . . . . . . . . . . . . . . . . . . 85
24 Patient Consent Sequence Diagram . . . . . . . . . . . . . . . . . . . 85
25 Algorithm for Generating EPL Statement of Success Verdict . . . . . 86
26 Algorithm for Generating EPL Statement of Fail Verdict . . . . . . . 87
27 Edge Computing Paradigm . . . . . . . . . . . . . . . . . . . . . . . . 88

28 Conceptual Reference Verification Architecture . . . . . . . . . . . . . . 90
29 IoT-enabled AAL Example . . . . . . . . . . . . . . . . . . . . . . . . 92
30 Sequence of Actions in CareWatch System . . . . . . . . . . . . . . . . 93
31 CareWatch Component View . . . . . . . . . . . . . . . . . . . . . . . 94
32 Sequence Diagram of CareWatch . . . . . . . . . . . . . . . . . . . . . 94
33 Increasing number of EPL statements 23 . . . . . . . . . . . . . . . . . 98
34 Future Work . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 105

# CHAPTER I

# INTRODUCTION

Software and software systems have become so widespread that we cannot think of any aspect in everyday life that does not involve a product with such a system in it. The scale of computing capabilities has penetrated a diverse spectrum of application domains encompassing end user artifacts such as key-chains, all the way up to an industrial factory automation. A technology innovation driven by introduction of IoT phenomenon has increased the momentum towards implementing systems of systems (SoS) with more smartness features [5]. The driver behind such an impetus of IoT is that it allows for engineering such integrated systems that merely require human interaction; thereby, enabling invention of new functionalities that is composed of autonomous endpoints operating in an accord with respect to a predefined service composition definition.

Every new capability built on groundbreaking concepts brings along a challenge in engineering the systems that we used to develop with older conventions. IoT enabled SoS's exemplify a domain of large-scale systems with so intricate detailed design elements that it necessitates innovative engineering solutions to tackle with. That is due to the fact that the size of an IoT system might scale from a system of tens of endpoints up to thousands (Figure.1).

A smart city infrastructure (Figure.1) hosts diverse applications of IoT systems [1]. The elements of such a system might contain smart parking, smart traffic lights, smart metering and similar smart vertical domain applications. That's why, such a system might be composed of thousands of interconnected IoT enabled endpoints. However, a smart house application is usually identified with individually operating

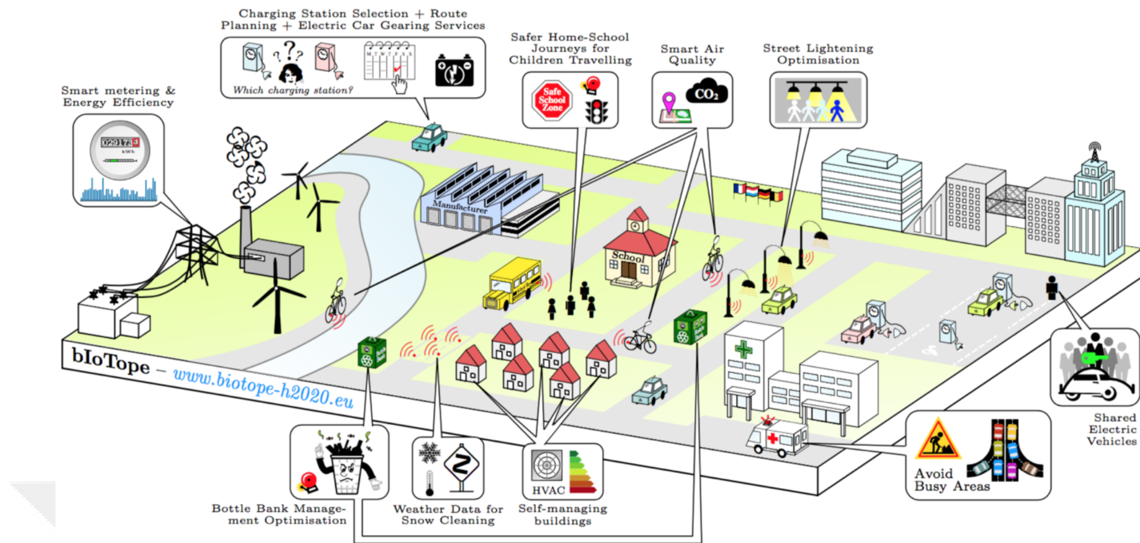


**Figure 1:** Examples of different scales of IoT deployment[1]

collection of IoT endpoints. Those systems generally operate in a stand-alone fashion, allowing end user to conveniently monitor and control various sensors and actuators in a house environment (Figure.1).

In the following sections of this chapter we will first specify the scope of this thesis. Then, a motivational section will be identifying the underlying research questions, which is followed by a section describing the contributions proposed in the thesis. A particular section is dedicated to present an overview of the thesis document.

## 1.1 Thesis Scope

Embedded systems are everywhere; and IoT enabled devices are proliferating the embedded systems in everyday life. According to a recent research by Gartner, Inc. the estimated number of IoT devices installed in year 2020 will be 20,415 millions of units[1] (Figure.2). The total number of IoT devices installed in year 2017 has already passed the total number of human beings on this planet.

IoT systems are enabled by several underlying technologies. One of those technologies that makes those systems easy to design, develop and test are the application

[1]https://www.gartner.com/newsroom/id/3598917

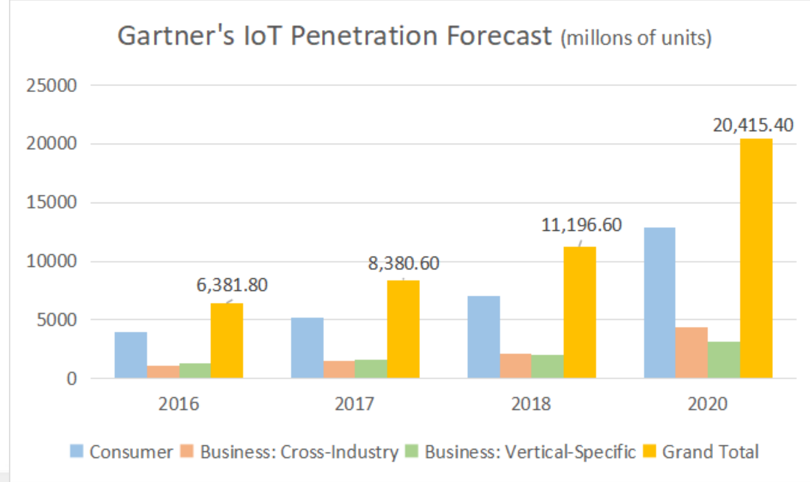


**Figure 2:** Estimated number of IoT devices installed between 2016-2020

layer communication protocols. Even though those systems might be built around hundreds or thousands of devices that are manufactured by different companies, the service-oriented architecture (SOA) principles adopted by some of the protocols facilitate engineering activities. Constrained-Application Protocol (CoAP), an OSI Layer-7 application layer protocol [2], is designed with RESTful API guidelines ([6]) as the building block of the protocol. The IETF Constrained RESTful Environments (CoRE) working group ([7]) has compiled the best practices as CoAP standard for supporting a SOA-based application development experience for resource constrained devices in IoT. RESTful services perspective has a tremendous advantage over other architectural approaches when it comes to discovery, composition, integration and execution of various services procured by different subsystems. Thus, engineering an IoT as a SoS is extremely streamlined with the introduction of such application layer protocols.

This thesis focuses on verification challenge of engineering a SoS with IoT devices, which might be a good example for a large-scale system. There are already various research efforts in testing the IoT ([8]). However, we particularly focus on runtime verification (RV) of such systems, because we argue that proliferation of reliable

IoT would necessitate a feasible RV solution considering the penetration pace of the phenomenon(Figure.2).

## *1.2 Motivation*

Internet of Things (IoT) has drastically modified the industrial services provided through autonomous machine-to-machine (M2M) interactions. Such systems comprise of devices manufactured by various suppliers. Verification of those systems is particularly challenging due to high heterogeneity of deployed devices and the potential scale of the SoS's in IoT domain.

RV has usually been considered as an intimidating verification method by the practitioners ([9]). That's why, it is poorly utilized in the industry. However, large-scale systems such as IoT call for feasible solutions for RV, because of the fact that those systems are deployed with such devices that are line-replaceable-units (LRUs); which means that an IoT device can be easily swapped with another one providing similar or new services. This capability has been introduced owing to the SOA based CoAP protocol. Such a feature necessitates re-assuring the reliability of the overall SoS at runtime, without interfering with the operational behaviors of the system. Those systems with such LRUs necessitate a black-box verification approach.

Providing an online and non-instrumented RV solution for such IoT systems as described above is the main motivation of this thesis. We sought for an online solution because SoSs engineered with LRUs require instant verification of sustained reliability. The driving idea behind providing a non-instrumented approach in such environments as IoT where those devices from various manufacturers can be plugged in and out of the SoS at will, is that we can not inject any instrumentation code to proprietary commercial-off-the-shelf products.

Therefore, this thesis present a model-driven engineering approach for IoT systems to promote automatic RV. We introduce a concept of designing for RV, which

establishes a foundation on representing the messaging model of CoAP in terms of events, then utilizing CEP techniques to yield success/failure verdicts on observed behaviors of a SoS. The modeling support is achieved through deriving an extended UML profile on an open-source modeling tool; and we propose to utilize an open-source CEP engine, which we believe will enable fast adaptation of the contributions of the thesis in the literature and the industry.

The aforementioned motivation elements can be more precisely defined in following research questions, which are main drivers for the thesis:

1. How to propose an RV solution that does not intimidate practitioners, which utilizes CEP tools?

2. How to enable an RV operation to be performed by using a CEP tool?

3. How to provide an intuitive process of RV for promoting its utilization?

## *1.3 Contributions*

Figure.3 summarizes the contributions of this thesis. We first propose an event calculus (EC) for enabling a transformation of message interactions occurring between communicating IoT endpoints into simple events. The event calculus builds on the idea of representing IoT behavior in Message Sequence Charts (MSC) ([10]). The EC primitives specified for IoT help us devise Esper EPL statements for monitoring IoT behavior at runtime.

Afterwards, the EC is utilized in order to develop a domain-specific meta-model (DSM) for specifying IoT behaviors. The DSM is in fact an extension of UML2.5 profile such that it provides a means to describe IoT interactions on a sequence diagram (SD). In order to automatically generate runtime monitors from SDs, we

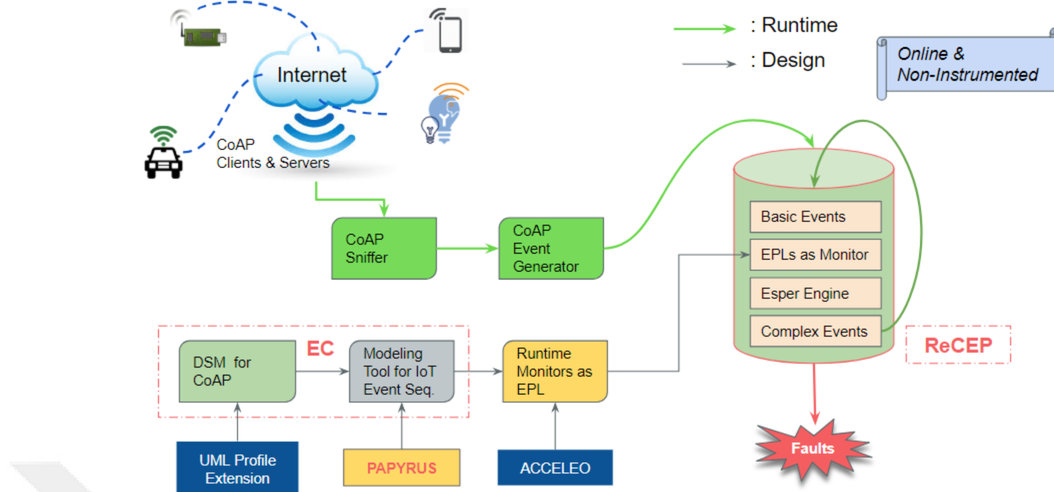


**Figure 3:** Contributions of the Thesis

provide an extension of Papyrus[2] modeling tool, which is an open-source Eclipse-based software. We propose to use Acceleo[3] model-to-text (M2T) transformation algorithms for generating CEP EPL statements from those monitors.

By proposing a complementary MDE solution as an RV approach, we are able to fill the gap between theoretical foundations of RV and practical verification operations. The thesis motivates RV problem for those IoT systems that communicate on Constrained-Application Protocol (CoAP). That's why, we also implement a non-intrusive CoAP Sniffer and a CoAP Event generator that injects CoAP messages captured from an IoT network into Esper CEP engine as simple events.

Note that aforementioned contributions explicitly address the research questions listed in the previous section. To be more specific, using Esper CEP tool for describing runtime monitors provides a seamless process for exercising the correctness properties of SUT, which addresses the first question (Chapter.4). In order to enable using a CEP tool, we first present a domain-specific EC that allows representing occurrences of sending messages in a CoAP system via events; later, those events enable us to use

[2]https://www.eclipse.org/papyrus/
[3]https://www.eclipse.org/acceleo/

a CEP tool for RV (Chapter.3). For promoting an intuitive process of RV for IoT, we democratize the process by providing an MDE-based automatic test generation framework, which addresses the last research questions (Chapter.5).

## *1.4 Thesis Overview*

The thesis is organized as follows.

**Chapter.2** introduces the reader to background information on relevant literature. It discusses the fundamental tenets of runtime verification and software testing as a service. This chapter is an extended version of the work described in [11]. The work conducted justifies the significance of an edge computing based solution framework for RV as a service.

**Chapter.3** presents the event calculus formulated for representing communication actions between IoT endpoints. Event calculus enables abstract description of CoAP messaging events so that the interactions can be specified by a technology-independent fashion. Basic occurrences such as sending and receiving a CoAP message are represented with proposed event calculus. Complex relations between those basic events are captured by specific predicates introduced in this chapter. The chapter is a revised version of the work presented in [12].

**Chapter.4** elaborates on the fundamental information on CEP concepts and how to apply them on event calculus proposed in Chapter.3. It first describes how to represent IoT messaging primitives with as simple events in CEP statements. Later, it goes on to specify template patterns that are utilized as runtime monitors for RV of an IoT behavior. We also introduce a reference design of a passive network listener for sniffing CoAP packages, whereby we promote a black-box verification approach. This chapter is a revised version of the presented in [12] and [13].

**Chapter.5** is a composition of the work we presented in [14] and [15]. Our contributions in MDE approach for providing a seamless method of automatically generating runtime monitors for RV of IoT systems. The contributions are demonstrated on various examples in the chapter.

**Chapter.6** The conclusions and related work regarding each chapter is investigated in the body of corresponding chapter. The last chapter of the thesis is dedicated to explain overall contributions and future work in a coherent way.

# CHAPTER II

# BACKGROUND

This thesis proposes a RV approach for IoT systems with CoAP as the application layer protocol. Therefore, we refer to several building blocks of both domains frequently in the body of the thesis. That's why, this chapter explains the fundamental concepts of RV and CoAP. The chapter also goes on to describe the gaps and opportunities in the verification domain, where more frequently verification services are procured as services over the cloud ([11]).

The details of techniques used to build the contributions are left to the corresponding chapters where we elaborate on each contribution. This chapter is organized as follows: Section.2.1 lays the foundation for the domain of discourse of this thesis, runtime verification. The main characteristic of problem domain, IoT, is described as CoAP protocol in Section.2.2. In the next section (Section.2.3) we survey the literature on testing-as-a-service (TaaS) in order to identify challenges and opportunities for our research. After expressing related work in Section.2.4, we conclude the chapter with a discussion of the gaps that we intend to fill in the remainder of the thesis.

## *2.1 Runtime Verification*

Recent developments in *software-as-a-service* business has made it possible for software systems to become much more prevalent than ever. Those systems operate more autonomously thanks to innovations introduced as agent-based smart objects [16]. Such improvements in engineering software systems necessitates innovative approaches to verification of those in order to enable reliable system operation.

With a broad perspective, verification methods can be listed as (i) *theorem proving*, (ii) *model checking*, (iii) and *testing* [17] (see Figure.4). Theorem proving explores

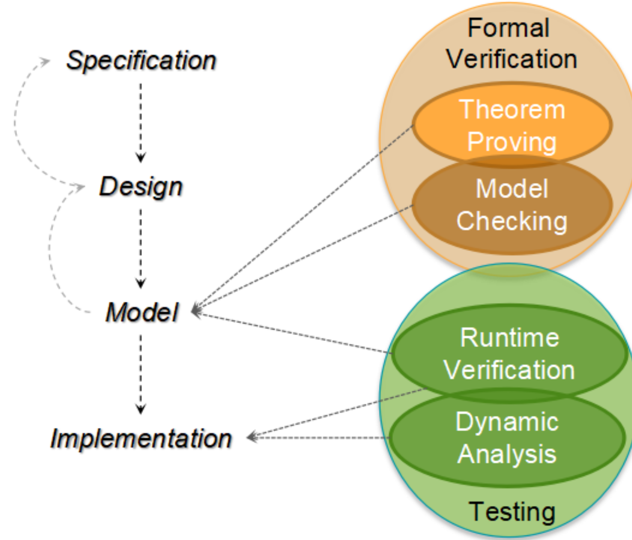


**Figure 4:** Verification Techniques

the correctness of a software by using certain proof techniques. Model checking, a technique that generally requires formal analysis of the system under test (SUT), is more related with verifying a software automatically at design time (i.e., without executing the SUT on production environment), which is usually represented as finite-state machines. On the other hand, testing is practically examining the SUT to demonstrate non-existence of faults in it [17].

Figure.4 summarizes the relation between different verification techniques. Theorem proving and model checking are types of formal verification, which involve mathematical models and proofs of system correctness through formal algebra. Those two types of verification constitute static analysis methods. However, RV, having its roots in formal specification of a SUT and being applied to SUT at runtime, is a complementary technique between functional testing and formal verification. It allows for reacting to unexpected behaviors at runtime [17].

Figure.5 recaps the principles of RV, which is formally defined as examining a systems execution trace against its specification expressed in correctness properties. A *correctness property* formally captures the expected behavior of a particular system

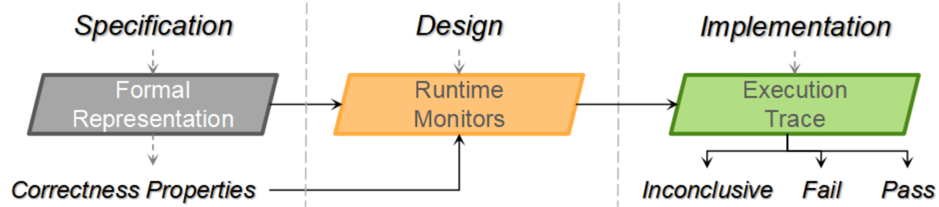


**Figure 5:** Runtime Verification Recapped

specification, which enables checking whether a runtime instance of the system conforms to that specification or not via runtime monitors. A *runtime monitor* is a tool that observes execution trace of a SUT and generates verdicts on the verification of a particular specification against its correctness property. *Execution trace* represents a finite run of a program, usually expressed in SUT's states.

Note that event calculus (EC) can be used for expressing state transitions in a system ([18]), so as to indicate a state change. In order to allow using EC, we define an *execution trace* to consist of a sequence of finite set of events in the SUT. The RV literature has many approaches covering formal methods of representing correctness properties and constructing runtime monitors (e.g., Linear Temporal Logic (LTL)) [17]. We are going to explain how we use EC and CEP techniques for specifying correctness properties and runtime monitors in Chapter.3 and Chapter.4, respectively. SOA-based systems interact through service compositions, that's why we represent each service request and reply message as an event, thereby enabling representation of such systems with proper EC.

## 2.2 CoAP-based IoT Systems

Proliferation of Internet-based utilities and technologies have permanently altered our lives for good. This phenomenon was reinforced by introduction of RESTful API guidelines [19], which have elevated SOA adoption. Many of the Internet utilities are widespread thanks to the utilization of web services that are architected according to RESTful APIs [20]. IoT domain also benefit from RESTful-like application layer

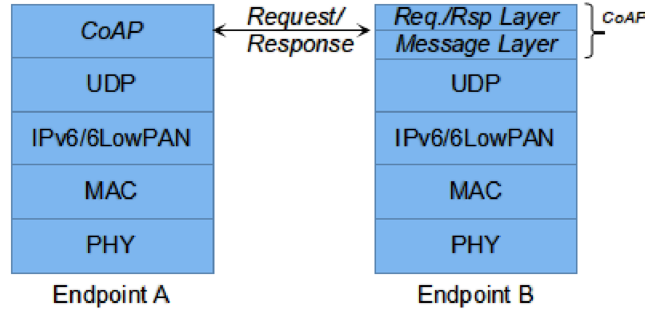


**Figure 6:** CoAP OSI Layering

protocols (i.e., CoAP). CoAP is developed on RESTful guidelines so as to allow developing system architectures through SOA principles. Although CoAP demonstrates HTTP-like interactions, it is specifically devised for resource-constrained devices with limited battery, low memory footprint and limited computation power.

CoAP dictates a *client/server* communication pattern as in other RESTful services (Figure.6), in which parties engage in an interaction by exchanging a series of *RequestandResponse* messages. *RequestandResponse* semantic of CoAP interactions enables us to represent the protocol layer as consisting of two implicit logical layers (Figure.6). 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). A CoAP client sends a *Request* message to a CoAP server, which defines the required action with a special *method code* on a resource of the server. The resources hosted on a server are identified by URIs (unified resource identifier), just as services in HTTP. 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*.

The messaging model of CoAP is an asynchronous interaction model. The messages are exchanged over UDP packets [2] (Figure.6). An entity participating in a CoAP interaction is called an *endpoint*. In a CoAP network, an *endpoint* may engage

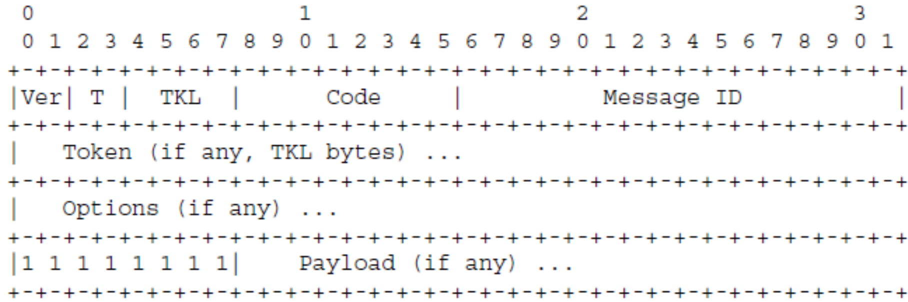


**Figure 7:** CoAP Message Format[2]

in interactions both as a client and a server at the same time. Figure.7 illustrates the packet format for a CoAP message. A CoAP message is classified into four different types: Confirmable (CON), Non-confirmable (NON), Acknowledgment (ACK), Reset (RST). The kind of a message is indicated by a corresponding bit value in the $Type(T)$ field of a CoAP packet determines the type of a particular message. A message is uniquely identified by a $MessageID$ field that is 16 bits long. The $MessageID$ field is not only used for assigning a unique ID to a message, but also for supporting reliability in a UDP-based application layer protocol; whenever an endpoint receives a message with a $MessageID$ that is a duplicate of a previously received message, then it's removed without processing further ([2]). The $MessageID$ value is used also for matching messages of type ACK/RST to messages of type CON/NON.

Even though the CoAP is operated over an un-reliable transport layer protocol, UDP, it is possible to improve reliability of IoT networks with CoAP devices by transmitting the messages with CON type. When a sender sends a CON message, the receiver of that message is expected to reply with a corresponding ACK message. In case an ACK message is not received in a predefined timeout ([2]), a series of re-transmission may be issued according to an exponential back-off mechanism, until a valid ACK is received with the same $MessageID$. However, if an appropriate ACK message is not received in $EXCHANGE_LIFETIME$ time, then the communication between sender and receiver is considered deficient. Note that, an endpoint might

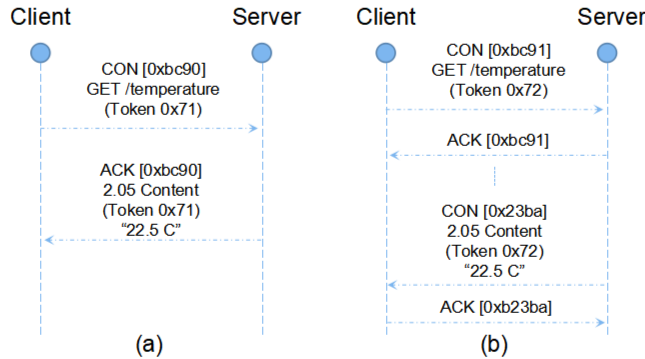


**Figure 8:** Demonstration of CoAP Messaging Model [2]

respond with a RST message, if it is not capable of processing a CON message. A Request can be conveyed via both a CON and NON message; and, the Response to a corresponding Request might be separately sent in a CON/NON message, as well as piggybacked in an ACK message.

Figure.8 illustrates the mechanics of asynchronous exchange of corresponding $CON$ and $ACK$ messages between a pair of client and server *endpoints*. In Figure.8.a, a *Response* to a corresponding $CON$firmable *Request* message is sent piggybacked to an $ACK$ message for that particular $CON$*Request* (i.e., notice how the $MessageID$ and $TokenValue$ fields). Figure.8.b demonstrates another version of $CON$, $ACK$ correspondence, in which the *Response* for the $CON$*Request* message with $MessageID$ $0xbc91$ is sent back later in a separate $CON$*Response* message with $MessageID$ $0x23bc$. That's why an $ACK$ message has to be sent from server to client indicating that the message with $MessageID$ $0xbc91$ has been received. Notice that the $TokenValue$ fields of corresponding *Response* and *Request* messages in Figure.8.b match. The $TokenValue$ field is used to ensure that a *Reply* message matches that of a corresponding *Request* message that has occurred previously in the network.

Considering the fact that CoAP is a communication protocol, the behavior of an IoT system with CoAP installed can be analyzed through Message Sequence Charts

(MSC), which is a formal description technique for communication protocols, developed by ITU-T [10]. It provides a trace language for the specification and description of the communication behavior of system components and their environment by means of message exchanges. Because the Z.120 presents the guidelines for an MSC in such an intuitive and transparent manner that the MSC language is easy to learn, use, and interpret. We are going to cover how to express a CoAP system in terms of events occurring in the form of message exchanges by means of MSCs in Section.3.2.

### 2.2.1 A Short Investigation of Lamport's Timestamps on CoAP

Lamport proposed a logical clock mechanism that enables synchronized communication amongst distributed computing platforms ([21]). The original problem of synchronization has been identified as the challenge of ordering the events occurring in distributed devices. The idea was to use a *timestamp* mechanism that observes a *causality* relation amongst the events in a system. *Causality* relation means that one event leads to another. An event is defined as a *sending* of a message from a process to another. Hereby, if an event $A$ *causally* "happens before" another event $B$, then $timestamp(A) < timestamp(B)$.

Lamport describes a *HappensBefore* logical relation among pairs of events denoted with $\rightarrow$. The relation is built on three rules:

1. on the same endpoint: $a \rightarrow b$, iff $timestamp(a) < timestamp(b)$,
2. if $EP_1$ sends message $m$ to $EP_2$: $send(m) \rightarrow receive(m)$,
3. it is transitive; if $a \rightarrow b$ and $b \rightarrow c$, then $a \rightarrow c$.

Those rules create a "partial order" amongst the events that are causally related to each other.

Hereby, we explain how the *Request/Response* messaging model can be expressed by using Lamport clocks, such that they exhibit a *HappensBefore* relation. Assuming

that the event of an endpoint $EP_i$ sending a *Request* message $m$ to another endpoint $EP_j$ is denoted by $send(i, j, m)$ and the receiving of that message at $EP_j$ is denoted by $receive(i, j, m)$. According to the messaging model of CoAP, a corresponding *Reply* message is assumed to be sent in a message $n$, which is denoted by events $send(j, i, n)$ and $receive(j, i, n)$, respectively. Using the rules of Lamport clocks listed above, we can show that $send(i, j, m) \rightarrow send(j, i, n)$ as follows:

1. $send(i, j, m) \rightarrow receive(i, j, m)$, by Rule 2;
2. $receive(i, j, m) \rightarrow send(j, i, n)$, by Rule 1;
3. then, $send(i, j, m) \rightarrow send(j, i, n)$, by Rule 3.

In following sections, we are going to build on these rules in order to come up with an event calculus that captures temporal ordering amongst events in the system.

## 2.3 Verification as a Service: Gaps and Opportunities

Cloud computing has emerged as a new paradigm that facilitates the development and utilization of highly flexible, elastic services on-demand, and over broadband network access. Those attributes are motivating many organizations to move their businesses to a cloud platform.

Software testing has been one of the best practice areas for migrating to cloud environment. Virtualization, which is an enabling technology of cloud computing, was first used for quickly creating virtual computation resources with different operating systems (OS) to test software applications on various platforms [22]. Testing a new software system often requires costly server, storage and network devices only for a limited time [23]. These computing resources are either not used or underutilized after testing, thus incurring extra cost on budget.

System and software verification activities demand excessive computing and human resources. For instance, to test the performance and scalability of a banking

application, the system must be stressed with requests from millions of users in a short time interval. This is a realistic scenario that should be tested because people rush to their bank accounts regularly on every payday. Reproducing such a scenario would require the provider to set up a test harness (including the user databases) to emulate the actions of millions of users. Similarly, mobile application providers frequently have to deal with maintaining the quality of their services over a plethora of various combinations of platforms [24]. The computing platforms may encompass different browser technologies with diverse back-end support running on various mobile OS. To ensure a reliable service, providers have to test their services on all these platforms.

Yet, another challenge is that each distinct domain of knowledge requires those involved in the verification of the applications in the domain to posses such sufficient expertise that allows them to engage in the verification challenge. This issue is exacerbated on large-scale SoS that may consist of subsystems built on various technologies. IoT is an example of such SoS's, in which endpoints can be either resource-constrained devices or full-fledged server computers; whereby, the practitioners of IoT verification are confronted with mastering a broad spectrum of technologies in order to conduct the tests on those systems. Consequently, it's getting more and more difficult for companies to employ the personnel with all the required skills for all kinds of system attributes. However, we argue that in order to alleviate the issues introduced with broad spectrum of enabling technologies involved in emergent innovations, such as IoT, SOA principles should be adopted in design phases.

Test automation topic is frequently visited when software testing is considered over the cloud. There are many test automation tools in the market, which address different requirements in a testing life-cycle (Figure.9). We believe our review on the subject matter will encourage the migration of those tools to the cloud.

One of the major drivers of cloud computing adoption is economies of scale. It

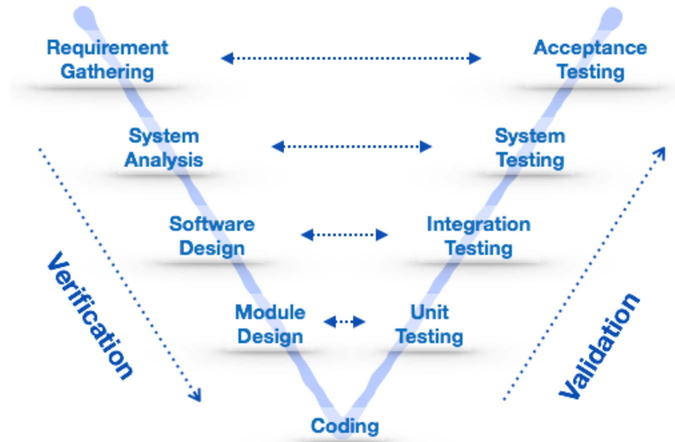


**Figure 9:** Development and Test Life-Cycle

supports a pay-per-use type of service procurement, thus eliminating an upfront investment in many cases. Testing tools and services are no exception. Development teams can benefit from this paradigm for utilizing test tools when they need it and as much as they need it, thus saving license fees.

We will enrich the discussion with current state-of-the-art software testing as a service over the cloud; and the survey will classify related literature according to what type of testing activities these services support for what type of application domains.

### 2.3.1 Cloud Computing

Cloud computing is a relatively recent term, which basically defines a new paradigm for service delivery in every aspect of computing. It enables ubiquitous, convenient, on-demand network access to a shared pool of configurable computing resources (e.g., networks, servers, storage, applications, and services) that can be rapidly provisioned and released with minimal management effort or service provider interaction [25].

Cloud computing has been enabled by the developments in virtualization, distributed computing, utility computing, web and software services technologies [26]. It is especially based on two key concepts. The first one is *SOA*, which is the delivery

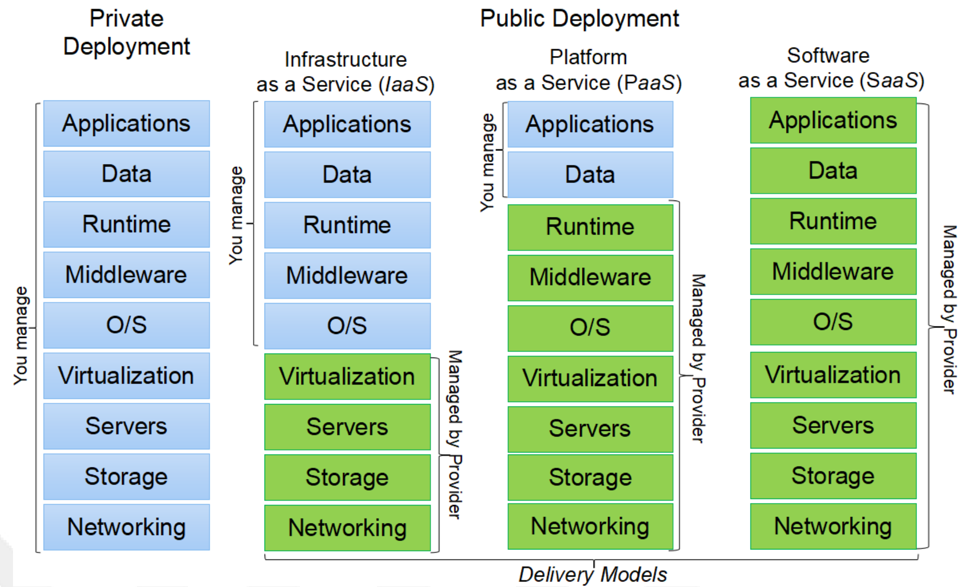


**Figure 10:** Cloud Deployment & Delivery Models

of an integrated and orchestrated suite of functions to an end-user. SOA enables end-users to easily search, use and release services on-demand and at a desired quality level. Workflows allow integration of services to deliver a business-valued application. The second key concept is *virtualization*. Virtualization allows abstraction and isolation of lower level functionalities and hardware, which enables portability of higher level functions and sharing and/or aggregation of the physical resources.

#### *2.3.1.1 Essential characteristics*

Cloud computing exhibits the following essential characteristics. *Rapid elasticity* allows end users to easily and rapidly provision new services and release them, enabling them to pay for what they utilize and how much they use it. *On-demand self-service* is an appealing characteristic for consumers because it provides them the flexibility of provisioning a service exactly when they need it. The services provided over the cloud are *measured services*, which means that consumers only pay for how much service they consume; thus eliminating the need for investing in redundant computing

resources. Cloud computing has benefits at the providers' end as well. A cloud computing provider pools its computing resources in order to serve multiple consumers by means of a *multi-tenant* provisioning model.

##### *2.3.1.2 Delivery models*

Even though there are several definitions for delivery models of cloud computing services, three are widely adopted in the literature. *Software as a Service (SaaS)* delivery model (Figure.10) is described as providing software applications/services over cloud infrastructure for consumers. These applications are accessible from various platforms through an easy-to-use client interface such as a web browser. *Platform as a Service (PaaS)* delivery model enables consumers to deploy their solutions to the cloud by means of platforms such as application servers and database services provided by the Cloud Platform Provider. *Infrastructure as a Service (IaaS)* is the lowest level of service model in cloud delivery models. In IaaS consumers acquire computing services and can deploy their own custom-configured systems in these resources potentially replicating their own existing infrastructures. Therefore, IaaS can also enable legacy system and software compliance.

##### *2.3.1.3 Deployment models*

The deployment model of a cloud platform is also important to consider when delivering or procuring on-line services [25]. Public (Figure.10) cloud infrastructures are provisioned for use by any consumer; infrastructure exists in the premise of the provider. Private cloud infrastructure is provisioned for exclusive use of a single organization and can be owned by a single organization, a third party, or some combination of them. Community cloud infrastructure is provisioned for exclusive use by a particular community of users from organizations that constitute the specific community. Hybrid cloud is a composition of two or more of the models above.

### 2.3.2 Software Testing and Virtualization

Software testing is an integral part of the software development life cycle (Figure.9) that span over all the development phases. One of the main challenges in software testing is deploying and maintaining a real-world test platform at the outset of a project. Virtualization technology has been utilized in testing various software since its inception in 1960's. IBM's CP-40 project might be considered as the pioneer of virtualization technology[22]. Among other goals of the project, CP-40 was mainly used by researchers as a tool to evaluate and test the performance of operating systems.

Developments in network infrastructure triggered a spur in Web-based service delivery. Riungu et.al. discuss the conditions that influence software testing as an on-line service and elicit important research issues [27]. They define on-line software testing as a model of software testing used to test an application provided as a service to customers across the Internet. This model promotes a demand-driven software testing market by enabling organizations and individuals to provide and acquire testing services on-demand. The concepts that affect software testing as an on-line service are domain knowledge, infrastructure, security, pricing, communication and skills [27]. On-line testing reduces costs related to installing and maintaining testing environment. It also introduces a new market where the providers and consumers can reach skilled test engineers on-demand.

### 2.3.3 Research Methodology

The main purpose of this study is to classify research activities performed in cloud-based testing area, clarify the terminology used, identify any gaps or open issues that remain, and address those issues at a high level. There are currently two different perspectives on "cloud testing" and both cases can be considered as valid forms of "Testing as a Service" [28]:

1. Testing the cloud-resident applications,

2. Providing testing software as services in the cloud, and
3. Both of the above, i.e., testing cloud-resident applications by means of cloud-resident testing services.

The former deals with how applications perform in terms of functional correctness and speed when they are migrated to cloud. The latter deals with migration of the testing process itself onto the cloud. This motivation enabled us to distinguish the problem domains of the literature. After thorough review of the selected papers we identified 11 major problem domains (Table.2.3.4.2 and Table.2.3.4.2) depending on the problem/solution domain of the paper. The problem domains that we identified allow us to make a distinction between whether the test service is provisioned for cloud-resident applications or for other platforms (e.g., desktop applications, mobile applications, etc.).

During our search for the related literature, we found the following keywords and phrases to be useful:

- cloud application validation
- cloud application verification
- cloud computing testing
- software testing cloud
- testing cloud applications
- verification cloud

Cloud computing partially relates to, and even depends on prior technologies such as virtualization, web services, utility computing, multi-core and parallel programming and several others. One can go back and analyze how testing processes were affected by these enabling technologies over a long period of time. IEEE, ACM, and

Science Direct are the main search engines we utilized for research. Due to proliferation of recent publications directly on Cloud Computing we decided to focus on publications dating 2009–2012, without limiting the search to any specific venue. Almost all conferences today in the fields of computer systems, data mining, software engineering and even consumer electronics hold special sessions on cloud computing, IoT and correlations of those in the form of edge/fog computing. Therefore, we did not lack any resources. Specifically, in these sessions we looked for papers that mentioned the keywords and phrases listed above. We eliminated any duplicate and incremental report. Our categorization approach resulted four distinct categories: *test level, test type, contribution* and *delivery model.* Based on this analysis we were able to identify trends and gaps.

### 2.3.4 Evaluation

Our survey of software testing literature resulted eleven problem domains, each of which is analyzed according to Test Level, Test Type, Contribution and Delivery Model.

#### *2.3.4.1 Categorization*

*Test level* categorizes the papers against the levels of traditional V-Model (Figure.9) of software testing: unit, integration, system and acceptance testing. *Test type* addresses the type of test that the investigated paper studies: functional, performance, security and interoperability. *Contribution* of papers classified according to the problem description: test execution automation, test case generation, a framework that either defines a tool or a methodology or an evaluation paper. We further investigated and extracted what type of *delivery model* that a specific research builds on.

**Table 1:** Categorization of Literature Based on Test Level & Type

| Problem Domain | Test Level | | | | Test Type | | | |
|---|---|---|---|---|---|---|---|---|
| . | Acceptance Testing | System Testing | Unit Testing | Integration Testing | Functional Testing | Performance Testing | Security Testing | Interoperability Testing |
| Mobile App.s | X | [30], [24] | [31] | X | [24] | [30] | [30] | X |
| Cloud App.s | X | [32], [33], [34], [35], [36], [37], [38] | [33], [34],[31], [35] | [37], [39] | [32], [33], [34], [35], [36], [37] | [32], [38] | X | X |
| Desktop App.s | X | [40], [33], [38] | [31] | X | [40], [33] | [38] | X | X |
| Web Services & App.s | X | [41], [42] | [31] | [26] | [41], [42] | [26], [42] | X | X |
| Distributed & Parallel App.s | X | [43], [44], [45], [46] | [31] | X | [43], [44], [45], [46] | X | X | X |
| Cloud Service Dev. & Deployment | X | [47], [48], [49] | [47], [50], [31], [35] | [48], [47] | [49], [50], [35], [51], [52], [53], [54] | [48], [51], [55], [56] | [51] | [51], [52], [57], [58], [59] |
| Migration to Cloud | X | [60], [61], [62], [63], [27] | [27] | X | [60], [63], [27], [61], [62] | [63] | [63] | X |
| Cloud Infrastructure & Storage | X | [64], [65], [66], [67] | X | X | [66], [67] | [64], [65] | X | X |
| Real-Time Systems | X | [68], [69], [70] | [69], [71], [31] | X | [68], [69], [70] | [72] | X | X |
| Network Config. | X | [73] | X | X | [73] | X | X | X |
| Test Task Mang. | [74] | [75], [28], [74] | [28], [74] | X | [75], [28], [74] | X | X | X |

**Table 2:** Categorization of Literature Based on Contribution & Delivery Model

| Problem Domain | Contribution | | | | Delivery Model | | |
|---|---|---|---|---|---|---|---|
| . | Test Execution Automation | Test Case Generation | Framework | Evaluation | SaaS | PaaS | IaaS |
| Mobile App.s | [30], [24] | [24], [31] | X | X | X | [30], [24] | X |
| Cloud App.s | [32], [36] | [32] | [32], [34], [35], [37] | [33], [38] | [32] | [37] | [37] |
| Desktop App.s | X | [31] | [40] | [33], [38] | X | X | X |
| Web Services & App.s | [42] | [42], [31] | [42] | [26], [41] | X | [41], [42] | X |
| Distributed & Parallel App.s | [44], [45] | [31] | [43] | [46] | X | X | [43], [44], [45] |
| Cloud Service Dev. & Deployment | [47] | [47], [31], [48], [50] | [76], [47], [35], [39] | [49] | [47], [48], [51], [77], [54], [55], [56], [78] | [39] | X |
| Migration to Cloud | [61] | [61] | X | [60], [63], [27], [62] | [61] | X | X |
| Cloud Infrastructure & Storage | [66], [65], [67] | [66], [67] | [66], [64] | X | [66], [65], [67] | X | [64] |
| Real-Time Systems | [69], [70] | [69], [70], [71] | X | [68], [72] | [69], [71] | [69], [70] | X |
| Network Config. | [73] | [73] | [73] | X | [73] | X | [73] |
| Test Task Mang. | [75], [28] | X | [75], [74] | X | [28], [74] | [28] | X |

#### *2.3.4.2 Gaps*

In our review, we could not identify any research that deals with effects of cloud deployment model on providing software testing as a service over the cloud. We believe that the deployment model has a critical role in procuring software testing service as it is for other on-line services. For instance, community cloud model might be further investigated for promoting community testing or crowd-testing (e.g., UTest [29]).

Our categorization emphasizes correlation between testing level, testing type and delivery model. It can be seen from Table.2.3.4.2 that interoperability testing presents opportunities for further research, which refers to interoperability of cloud services. Yet another domain of research for interoperability is large-scale systems such as IoT ([14]). Vast amount of heterogeneity in IoT devices raises new challenges to tackle for providing and assuring a sustained interoperability of those systems. We discuss the interoperability issue of IoT systems in Section.5.1. It's also shown in Table.2.3.4.2 that acceptance testing has not been studied thoroughly.

We noticed that contribution of a literature and its delivery model are mostly interrelated. For instance, it can be seen in Table.2.3.4.2 that [37] introduces a framework for testing and it is classified as an IaaS delivery model; so is [43].

Workload distribution and management over the cloud has being studied by the cloud community. Automated tests might be investigated in terms of their correlation with available task management frameworks or infrastructures. This subject is not studied thoroughly ([75], [28], [74]). We believe that task management issue in distributed and parallel applications has long been studied, and studying those solutions might facilitate task management for testing over the cloud.

As more and more services are migrated to the cloud, verification of legacy applications over cloud will gain more attention by the research community. Especially acceptance testing of those applications needs to be well-structured in order to reap

the benefits of cloud. Ding et al. describe why post-migration testing is necessary when migrating a complex application to cloud in [61]. They introduce a black-box approach for post-migration testing of Web applications without manually creating test cases. They propose to automate those tests and present a software module called Splitter. The tool executes automated functional test by using actual user workload in real-time to compare responses from the migrated application against those from the original production application before cut-over to the new platform. Migration of legacy systems should not only be investigated in terms of system and functional specifications but also studied in terms of performance, security, unit-level verification and integration of composing services (Table.2.3.4.2).

Many cloud services are provisioned through composition of several services.In the near future, several cloud infrastructure service providers may be utilized in providing value-added cloud services. Thus, interoperability testing of cloud infrastructures and real-time systems (e.g., IoT) requires further research.

#### *2.3.4.3 Testing for the Cloud*

Testing for the cloud defines the testing of applications that are specifically developed to run on a cloud platform. This fact entails that the application might be utilizing parallel computing features of cloud computing or it might be a multi-threaded application. Parallel program testing becomes more critical with the proliferation of cloud computing services.

Cloud service development and deployment, test task management, cloud infrastructure and storage, cloud applications domains are good examples of testing for the cloud. For example, Chan et.al. propose a graph-based modeling approach to cloud applications and attempt to support the approach with a testing paradigm for cloud applications [36]. The testing relates to the notion of model-driven engineering.

#### 2.3.4.4 Testing on the Cloud

We distinguish the testing activities for on-premise applications as “testing on the cloud”. In this type of service, the system under test resides either on-premise or on the cloud for testing purposes, but it’s deployed on a platform other than cloud.

Testing for certification is a good example for testing on the cloud. On-demand service delivery attribute of software testing over the cloud paradigm might attract end-users to test the applications which they will install on their PC or mobile devices or check the applications’ conformance to certain standards [33].

Unit testing activities are another area where on-demand software testing service can be utilized. Symbolic execution concept has been migrated to cloud environment, which facilitates automatic test case generation for unit tests ([69], [70], [31]). Symbolic execution presents opportunities for automatic test generation and test execution automation; but it’s not widely studied according to the problem domains we presented (Table.2.3.4.2). Thus it presents further research opportunities.

Testing activities usually mean verification activities. Verification and validation should be considered as a complete service for the quality purposes. Verification and Validation as a Service (VVaaS) should answer both questions: whether the software does the right thing and whether the software is built to do the right thing. Thus acceptance testing should be considered as a new test service to be provided over cloud. VVaaS over the Cloud should be studied and promoted because one of the goals of software testing research is to automate the testing activities as much as possible, thereby

- significantly reducing the cost,
- minimizing human error and
- making regression testing easier.

## 2.4 Related Work

There is a vast amount of literature regarding software testing in the cloud and testing cloud services. However, to the best of our knowledge, there is no comprehensive literature review that categorizes exiting body of work according to problem and solution domains. There have been previous works for identifying research issues for software testing in the cloud [53]. These works are based on a survey conducted with industry practitioners, in which issues are categorized from the application, management, legal and financial perspectives. The analysis of this survey reveals the requirements of a cloud-based testing solution from the viewpoint of industry practitioners [27].

## 2.5 Conclusion

Cloud computing and software testing are likely to be active and popular research fields in the near future. Traditional software testing techniques are being adapted for the cloud. On the other hand, cloud computing itself is under constant evolution, continuously bringing in new opportunities and challenges for software testing research. In this section, we have presented a classification of current research studies, identified gaps in the literature and investigated the correlation of software testing with different deployment models of cloud computing. Researchers in this field can benefit from the results in selecting their research direction and identifying new research opportunities for future work. We have observed that acceptance testing is an open research area for testing over the cloud. Test task management is also among the potential areas for further research.

As we defined in Section.2.1 RV is an intimidating method for practitioners of verification. New computing paradigms such as IoT and edge computing promises new types of utilization models for cloud services. Those systems requiring more robust and scalable verification approaches call for unconventional solutions for RV.

Having interoperability as an ongoing issue in IoT systems, we believe that testing for interoperability of those needs more emphasis as a research area to ensure reliable service composition by means of integrating services from heterogeneous device manufacturers.

# CHAPTER III

# DERIVING AN EVENT CALCULUS FOR IOT

Considering the complexity of modern computers [79], 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. RV [80] is a method in which monitors oversee the execution of a SUT in order to check 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.

IoT systems are heterogeneous systems in 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. Such architecture principles as SOA adopted by IoT protocols (i.e., CoAP) allow expressing an expected behavior of such a system by means of events; so, it encourages to exercise those systems at runtime for checking

correctness properties defined at the design phase. The proposed solution can be tailored for other protocols by following the steps explained in the chapter.

The focal idea behind this research is facilitating the RV process for IoT domain applications by using open-source or commercial-off-the-shelf CEP tools (e.g., [81]). In order to contrive such a purpose, we first need a seamless method of expressing IoT systems in terms of event occurring in those. Because, CEP tools operate on simple events (Chapter.4) for infer complex decisions on the circumstances taking place in the system. We propose to tailor an event calculus (EC) specifically for IoT domain so that we can achieve to express the expected behaviors of it in a human-readable form that is also independent of any available CEP tool.

This chapter is a consolidated and extended version of our work in [12] and [13]. We leverage our argument on the interactions described in a MSC to specify message exchanges of CoAP-based IoT systems in terms of events; a novel event calculus for formally describing IoT system constraints has been specified by means of the linearization of a MSC. Thereby, we enable specification of correctness properties that are used for describing monitors in RV.

The chapter starts with an introduction of basic event calculus (EC) information, and how to use it for new domains. Then, a section is dedicated to identifying principles of MSC's that is used to describe communication protocol behavior. Afterwards, a novel EC is proposed in a particular section, which elaborates how to represent a runtime monitor in terms of complex-events obtained by applying the EC algebra based on simple interactions occurring in an IoT system. The utilization of the proposed approach is demonstrated on a case of wireless token ring protocol (WTRP), which is a frequently utilized protocol for ensuring quality of service in WSNs such as IoT. A separate section specifies the literature review on runtime verification solutions proposed for IoT systems, embedded systems.

## *3.1 Event Calculus Revisited*

In real world, we use natural language assertions to indicate the occurrence of a change in the state of a phenomenon. For example, "The weather turned rainy" sentence implies that the weather has changed state from "not raining" to "raining". Those occurrences that cause a change in the state of a phenomenon are called *events*. Information systems can also be specified in terms of events occurring in the system. Event calculus, first introduced by Kowalski and Sergot [82], is a method of representing occurrences in a domain of discourse with respect to temporal relations. Even though it was initially used for making sense of database transactions, it can also be used for program specifications [82]. Since its proposition, there have been several attempts to extend its representational expressiveness. Kowalski [83] proposed a simplified version of event calculus (SEC) that replaced the event occurrences with event types. Table.3 presents some of the predicates of SEC.

The main goal of our study is to facilitate the research and practice in IoT domain by seamlessly conceptualizing the system under development in terms of event occurrences. If we can automate the process by which we derive conclusions about actions occurring in a system, then we can develop much better engineering approaches for common problems such as RV. EC provides tools for describing those systems in terms of events [84, 85, 86]. In order to formulate an EC for a specific domain we should;

- specify *simple events* in the system;
- specify the algebra that correlate those simple events in order to deduce complex conclusions;
- 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

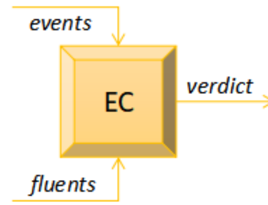


**Figure 11:** How Event Calculus Works

of various event-processing engines for monitoring and verification of system behavior at runtime.

Figure.11 illustrates how EC actually functions. An *event* is a construct of EC that specifies what happens when in a system. In other words, an event is defined as any happening in a context at certain time, that is irrevocable, which causes the system state to change. Another construct of an EC is *time-varying properties* of a system, which is called a *fluent*. A *fluent* allows describing how actions in a system are reacted upon by a system. The *verdicts* determines what's *True*, and when. An EC allows for generating commonsense decisions about actions and corresponding changes of them on a particular domain of discourse [18]. Therefore, we can list fundamental constructs of an EC as;

- events,
- 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 [84]. Fluents (f) allow for representing time-varying properties of a system. There are two concrete happenings that rely on these building blocks;

1. an event can occur in a unique time instance,
2. a system property is true only at a single time-point.

**Table 3:** SEC 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 $True$ after $t$ |
| $Terminates(e, f, t)$ | if $e$ occurs at $t$, then $f$ is $False$ after $t$ |
| $StoppedIn(t_1, f, t_2)$ | $f$ is stopped between $t_1$ and $t_2$ |

As Mueller states [18], event calculus allows to conduct various operations such as design, development and testing in a native computing paradigm on concurrent events, continuous time, events with duration, partially ordered events, and triggered events. In order to make use of commonsense reasoning for EC, we must first identify the domain of interest, and then provide common knowledge on that domain. The resulting situations that arise after an event happens at a particular moment in time are then described.

For instance, a certain event occurring under particular conditions might trigger a predefined system functionality. 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 [84].

EC algebra make use of predicate logic for elaborating time-varying properties of a system, namely *fluents*. Researchers proposed various versions of EC [18]. The EC that we use conforms to Simple Event Calculus (SEC) [18]. The predicate functions and corresponding descriptions are given in Table.3.

In Table.3, $Happens(e, t)$ states that an event e happens at a timepoint $t$; and the $Initiates(e, f, t)$ (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 $t$.

SEC allows us to descriptively specify event-driven requirements of an IoT system. Our approach aims to provide an EC algebra that facilitate expressing IoT message interactions. EC formulas allow to represent those systems in terms of events, in other words, it lets us to transform communication primitives happening in time domain into a discrete event domain. Thereby, EC enables using the logic theory for verification of both design artifacts and runtime system. As pointed out in [86], EC provides a representation that is very similar to interaction models such as RESTful API's. Besides, EC formulas involve a definitive time value; thus, allowing us to distinguish between events occurring at the same time in an event-based system, such as IoT, provided that a system behavior can be described in terms of events, we can develop a domain-specific EC for it.

## 3.2 Representing IoT with Events: Leveraging MSCs

In our endeavor for representing IoT systems by using event calculus, we first need a means to express the interaction between endpoints in terms of simple events. Message Sequence Charts (MSC) are the fundamental means of providing a language for representing execution trace of the specification and description of the communication behavior of system components and their environment by means of message interchange [10]. An MSC specification describes the order of occurrence of message interchange in communication protocols; and, each occurrence is defined as an *event*. Therefore, MSC's are best suited for describing behavior of communication scenarios. 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 ([87], [88], [89]). 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. We restrict ourselves to the *send message* events that are

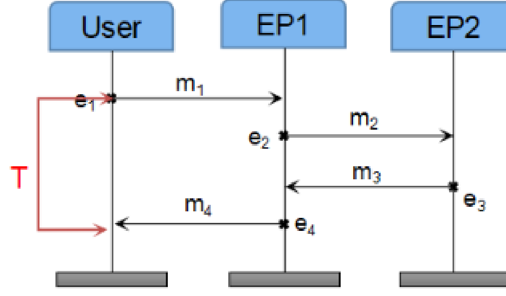


**Figure 12:** MSC for a CoAP Service $S$

observable from the network in a black-box fashion because we promise to provide a non-instrumented RV solution, whereas others elaborate on process level *receive message* events, which are solely observable via a process-level code instrumentation.

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 Figure.12. The vertical lines in the figure are lifelines for each endpoint in a CoAP system, and a lifeline illustrates the time axis for each endpoint. The time increases downwards on a lifeline. 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, every *Reply* message must be issued for a corresponding earlier *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 Figure.12 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 $EP = \{EP_1, EP_2, ..., EP_n\}$ be a set of endpoints in an IoT system, and let $A$ ($A \equiv \{m_1, m_2, m_3, m_4\}$) be a message alphabet for the sequence diagram under

inspection, where $[n]$ denotes $\{1, 2, .., n\}$. We represent each asynchronous message with the label $send(i, j, m)$, which indicates the event of an endpoint $EP_i$ sending a message $m$ to an endpoint $EP_j$. Note that since CoAP exhibits a Request/Response type of messaging model, a message can be either a request message, $m^R$, or a response message $m^r$ (i.e., $m \equiv m^R \cup m^r$). We further define the set $\varepsilon = \{send(i, j, m) \mid i, j \in [n]\ m \in A\}$ 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 $send(i, j, m)$ event can be either a *Request* or a *Reply* event. Therefore, $\varepsilon$ can be partitioned into $\varepsilon^R$ and $\varepsilon^r$ subsets representing set of *Request* events and set of *Reply* events, respectively. $\varepsilon = \varepsilon^R \cup \varepsilon^r$ is the set of all *send* events, where $\varepsilon^R = \{send(i, j, m^R) \mid i, j \in [n]\ m^R \in A\}$ and $\varepsilon^r = \{send(i, j, m^r) \mid i, j \in [n]\ m^r \in A\}$, respectively. The MSC $M$ then can be described as

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

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, $f : E^R \mapsto E^r$

4. a labeling function, $l$ that identifies each event as either *Request* or *Reply*, $l : E \mapsto \varepsilon$, such that $l(E^R) \subseteq \varepsilon^R$ and $l(E^r) \subseteq \varepsilon^r$

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 $\prec \doteq \cup_{i \in [n]} \prec_i \cup \{(e, f(e))) \mid e \in E^R\}$ is a partial order on $E$.

Let's consider the MSC in Figure.12. 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, $f(e_1) = e_4$. 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 $f(e_1) \prec f(e_2)$. A thorough coverage of non-degeneracy condition and MSC formalization can be found in [88] and [87].

Now that we have a definition for an MSC, we can use this to express a specification that an MSC can deliver. We define the specification of an MSC by its *Linearization* [87]. Based on the non-degeneracy condition assured by the reliability option of CoAP protocol, which prevents receiving of duplicate messages sent by an endpoint, a word $w = w_1, w_2, ..., w_{|E|}$ over the alphabet $A$ is a *linearisation* of an MSC $M$ if there exists a total order $e = e_1, e_2, ..., e_{|E|}$ of the events. The word is *well-formed* if each reply event there is a corresponding request event. A *Linearisation* of an MSC $M$, which is represented by a *word* $w$ 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 Figure.12), is attained by a total order of events in $E$; and it is considered as a *string* over $\varepsilon$. In other words, a *Linearisation* is said to exist if a total order of $(e_1 e_2 ... e_n)$ 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, $\Gamma$, that delivers $N$ distinct services, would consists of a disjoint set of $N$ MSC Linearization. That is, specification contains $N$ MSCs $M_1, ..., M_N$ each representing a distinct service implementation, where $E_1, ..., E_N$ are disjoint event sets. Let $\Sigma = \cup_{j=1}^{N} E_j$ be the disjoint sets of events in $\Gamma$; $\Upsilon = \cup_{j=1}^{N} A_j$ be the message alphabet of $\Gamma$, and $\Psi = \cup_{j=1}^{N} EP_j$ be the set of endpoints in $\Gamma$. Then the language of an MSC Specification $\Gamma$ is the union of languages of all MSC's in $\Gamma$. Note that the message alphabets and endpoints in different MSC's can be similar, because an endpoint may engage in several similar interactions in various service compositions.

We have shown that a Linearization 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 $\Psi_j$ is the set of endpoints for $M_j$. MSC guidelines [10] provide various graphical operations

such as *co-region, 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-driven engineering approach we are working on. We will utilize this *event* phenomenon in facilitating an *event calculus* for IoT systems in the next section.

## 3.3 Event Calculus for IoT

From a system (network) viewpoint, sending a request or a response message in a CoAP application constitutes an action. The motes in a CoAP network behaves either as a client or a server [2].

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. 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 $e_2$ ($e_1 \prec e_2$ ) pairs can be described with the following axiom by using SEC:

$$\begin{aligned} Follows(e_1,\ t_1,\ e_2,\ t_2) \ \equiv \ \exists\, e_1,\ e_2,\ t_1,\ t_2\ (Happens(e_1,\ t_1) \wedge \\ Happens(e_2,\ t_2) \ \wedge \ (t_1 \ < \ t_2)) \end{aligned} \qquad (1)$$

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 happening of a terminating event.

$$\begin{aligned}HoldsAt(f,t) \equiv \exists\, e_1,\ e_2,\ t\ (Initiates(e_1,\ f,\ t_1) \wedge \\ Follows(e_1,\ t_1,\ e_2,\ t_2)\ \wedge\ Terminates(e_2,\ f,\ t_2) \wedge \\ (t_1\ <\ t\ <\ t_2))\end{aligned} \qquad (2)$$

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

### 3.3.1 Event Calculus for CoAP

Considering an execution of the MSC $M_i$ in Figure.12, the trace can be monitored in terms of *send message* events in the network. As pointed out in [90], 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 temporal order of events in a trace implicitly exhibit a *follows* relation between each pair of consecutive events (Eq.1). 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* (Figure.5) 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 Figure.12, $e_1$ 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 $t_i < t_j$: 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 $t_i < t_j$: 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 $(t_i < t_m) \wedge (t_m < t_j)$.

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 a 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 correctness properties of a SUT. 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* [2], by using the relations defined above. We can express this requirement as

$$\begin{aligned} Req(CON_i) = f(e_{ACK_i}, e_{CON_i}) \wedge \\ [t(e_{ACK_i}, e_{CON_i}) < EXCHANGE_LIFETIME] \end{aligned} \tag{3}$$

, where $(e_{CON_i}, e_{ACK_i})$ is any pair of send events for a CoAP message $m$ as such $e_{CON_i}$ denotes sending event of the confirmable message with $MessageID\ i$, and $e_{ACK_i}$ denotes its corresponding $ACK$ message with the same $MessageID$, $i$ [2].

Eq.3 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 yield a $Pass$ verdict for a particular $CON$ message $m_i$, the equation must hold $True$ for $(CON_i, ACK_i)$ event pair. However, in order for the correctness property expressed in Eq.3 to $Fail$, we must have;

$$\neg Req(CON_i) = \neg\{f(e_{ACK_i}, e_{CON_i}) \wedge [t(e_{ACK_i}, e_{CON_i}) < EXCHANGE_LIFETIME]\} \tag{4}$$

or

$$\neg Req(CON) = \neg f(e_{ACK}, e_{CON}) \vee [t(e_{ACK}, e_{CON}) >= EXCHANGE_LIFETIME] \tag{5}$$

Eq.5 states that Req(CON) is false either eACK does not occur at all after eCON, or it occurs after EX.. time.

Event linearisation for the sample MSC in Figure.12 constitutes an expected behavior of message interactions between endpoints, such that it represents the specification for the service *S Requested* by event $e_1$:

$$Req(S) = e_1 \prec e_2 \prec e_3 \prec e_4 \tag{6}$$

, 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.6. This linerarization can be interpreted in event relations of MSC as

$$Req(S) = f_i(e_2, e_1) \ \wedge \ f_i(e_3, e_2) \wedge f_i(e_4, e_3) \ \wedge [t(e_4, e_1) < T] \tag{7}$$

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.7, $t(e_4, e_1) < T$ expresses 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.8).

$$
\begin{aligned}
\neg Req(S) = \neg f_i(e_2, e_1) \ \vee \ \neg f_i(e_3, e_2) \ \vee \ \neg f_i(e_4, e_3) \\
\vee \ [t(e_4, e_1) >= T]
\end{aligned}
\tag{8}
$$

A CoAP network can provide communication among hundreds, even thousands of endpoints delivering various services. Therefore, we need to devise a solution for distinguishing repetitive invocations of a same service by different clients. We propose to exert a notion of *context* on processing of events occurring as a result of the invocation of a particular service (e.g., service $S$ in Figure.12). 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 *Requested* service, thus they are out of context. So, only those events that appear on the diagram are context events.

A context $C_{MSC}$ can be described as follows:

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 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.3 we can express the *immediately follows* relation of MSC as

$$\begin{aligned} f_i(e_j, e_i) = Happens(e_j, t_j) \ \wedge Happens(e_i, t_i) \wedge \\ \neg Happens(e_k, t_k) \end{aligned} \tag{9}$$

, 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

$$\neg f_i(e_j, e_i) = f_i(e_k, e_i) \ \vee f(e_i, e_j) \tag{10}$$

, where $e_k \in A_{comp}(e_j, e_i)$. Eq.10 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.10 can be elaborated in event calculus terms by expanding $f_i$'s as in Eq.9

$$\neg f_i(e_j, e_i) = \begin{cases} Happens(e_k, t_k) \wedge \\ Happens(e_i, t_i) \wedge \\ \neg Happens(e_j, t_j), & \text{if } Cond_A \\ Happens(e_j, t_j) \wedge \\ Happens(e_i, t_i), & \text{if } Cond_B \end{cases} \tag{11}$$

Table 4: Context and Event Verdicts for Figure.12

| Verdict | EPL Statement for RV |
|---|---|
| $C_E$ | $\{e_1, e_2, e_3, e_4\}$ |
| $e_0$ | $e_1$ |
| $Pass$ | $f_i(e_2, e_1) \ \wedge \ f_i(e_3, e_2) \ \wedge \ f_i(e_4, e_3)$ |
| $Fail$ | $f_i(e_3, e_1) \ \vee \ f_i(e_4, e_1) \ \vee \ f_i(e_4, e_2) \ \vee \ f_i(e_1, e_2) \ \vee \ f_i(e_3, e_4) \ \vee \ f_i(e_2, e_3)$ |

, 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.11 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 Figure.12 again, we can elicit all the event traces that cause the sample scenario to either succeed or fail as in Table.4. The event relations appearing in $Pass$ and $Fail$ rows of the table represent *runtime monitors* for the MSC in Figure.12 in terms of EC. Thereby, we can exploit event relations any IoT with CoAP for determining correctness properties in event calculus, provided that they are expressed in an MSC with such relations as in Eq.9 and Eq.11. In Chapter.4, we are going to explore how we can translate those basic event calculus predicates and constraints into complex event processing statements.

## *3.4 Case Study: Wireless Token Ring Protocol*

In case when a wireless network needs to be self-healed, self-organized and be deprived of any centralized features, the Wireless Token Ring Protocol (WTRP) is applied [91]. WTRP features high quality provisions for networks that exhibit limited bandwidth and bounded latency characteristics. WTRP is best suited for resource-constrained networks such as IoT, because it supports constructing ad-hoc networks dynamically, provides energy saving measures and efficient transport mechanisms[91]. 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

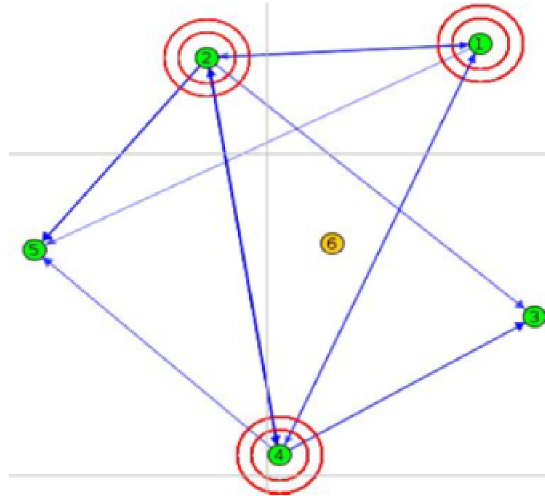


**Figure 13:** Cooja Simulation of WTRP

of communication range. 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 [92], but they might fail due to random system failures such as poor programming skills. However, WTRP is expected to preserve the order of token passing at all occasions. In order to verify the correctness of such a sequential operation of WTRP algorithm, the system developers must monitor its performance by observing the token transitions between endpoints.

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 [91]. 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 [93]. Improvements introduced in [94] promote dynamic ad-hoc network structure expansion, energy saving and transport efficiency enhancements, which are valuable attributes for CoRE (Constrained RESTful Environments) [2] 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.

$$Initially(TO(m,\ 0))\ is\ True\ for\ where\ m\ \equiv 1 \tag{12}$$

Figure.13 demonstrates a simulation of WTRP network in Cooja [95]. We will use the event calculus proposed here 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 $id$s in the network; and mote with $id1$ is assumed to own the token at time 0 (Eq.12).

Assuming that the network in Figure.13 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 increasing mote $id$s, 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.

$Send(m,\ t)$ 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.

$$OO(t)\ \equiv\ 1,\ \ t \tag{13}$$

$$\begin{aligned} HoldsAt(OO(t),\ t) \equiv \forall m_1, m_2\ \exists t_1, t_2\ (Happens(Send(m_1,t_1),\ t_1) \wedge \\ Initiates(Send(m_1,t_1),\ TO(m_2,t_1),\ t_1) \wedge \\ Terminates(Send(m_1,t_1),\ TO(m_1,t_1), t1) \wedge \\ (m_2 - m_1 = 1)\ \wedge\ Happens(Send(m_2,t_2),\ t_2) \wedge\ (t_2 > t_1)) \end{aligned} \tag{14}$$

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.

## 3.5 Related Work

Chen et.al. [8] have proposed a methodology for interoperability testing of CoAP implementations. As for the CoAP specifications [2], 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. [79] 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.

In [86], authors propose an online RV technique that relies on certain mediation techniques, and use Complex-Event Processing (CEP) [81] 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.

In [96], they 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.

In [97], Kane 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.

## *3.6 Conclusion*

The event calculus (EC) for CoAP-based IoT system interactions is provided in this chapter. 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. 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 meta-model that can be used in representing IoT systems with modeling languages such as UML. The MSC approach presented in this chapter lays the foundation for our ongoing and future work on model-driven engineering 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.

# CHAPTER IV

# RUNTIME VERIFICATION OF IOT SYSTEMS USING CEP (RECEP)

do you explain why we need an event calculus for IoT. you should express how you want to facilitate RV by using a complex-event processing engine, which executes simple events occurring in a system, then yields decisions regarding the complex relations of patterns amongst those events.

In 1999, when IoT phrase was first coined by K. Ashton [98] 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” [98]. 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 [5, 99], devices in IoT are considered to be smart-objects. These objects 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) [100, 6, 101] specifications which facilitate collaboration of “things” [100, 2].

The *Request/Response* and asynchronous interaction of CoAP provides a natural support for sequential analysis of events occurring in an IoT network. Even though there is a considerable effort in the literature for verification of such complex and distributed systems [8], a practical solution for IoT systems that supports runtime

verification is still missing. In this chapter, we propose a CEP-based runtime monitoring approach for IoT systems that leverage sequential relations between events as explained in Chapter.3. We also explain the design of a passive network sniffer for capturing CoAP packages without instrumenting the SUT. We further present a simple case scenario to demonstrate the applicability of the approach on WTRP execution.

Entities employed in IoT systems are generally resource-constrained devices; therefore, a new application layer protocol, CoAP [2], has been standardized for enabling IoT system architectures with RESTful services [102, 103] 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 [8], but those studies lack any support for system-level verification.

Verification of heterogeneous 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 that are designed with SOA principles. Our solution approach is designed to yield failures in the form of complex-relations amongst simple events that represent message interactions between endpoints in an IoT system. We discover complex relations among simple events via consolidating them in a CEP [104] engine, namely Esper [81], which allows for exerting various operations on those simple events in order to deduce complex decisions. The solution does not intervene with the operational system and as such it does not incur any overhead in system communication or nor does it alter the system behavior (i.e., this is achieved through passive CoAP sniffer).

CEP techniques have already been utilized for verification purposes such as network congestion control and intrusion detection [104, 105, 106]; 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 chapter are *(i)* CEP statements for the event-calculus proposed in Chapter.3; *(ii)* design of a non-intrusive network sniffer for CoAP packages; and *(iii)* a case study on WTRP.

The organization of this chapter introduces the reader with CEP concepts through an open-source CEP engine, Esper [81]. Then, we define how to utilize some of the basic elements of Esper CEP engine for specifying runtime monitors for simple event relations as defined in Section.3.3, which inherently help us transforming EC formulas specified in 3 into EPL statements. The Section.4.3 further describes the design of a non-intrusive CoAP Sniffer, that is built on open-source libraries. After demonstrating the CEP contributions in Section.4.4, we discuss the performance results of a demonstration on the case scenario WTRP in Section.4.5. Note that, Section.4.2 can be tailored to represent runtime monitors for new domains with a different EC for that particular domain.

## *4.1 Complex-Event Processing with Esper*

CEP is a technique that was initially introduced for deducing complex decisions in business processes [104]. CEP tools enable us to make high-level decisions about event-driven systems by inferring complex meanings out of simple events in various domains [105, 106]. Simple events can be consolidated into complex events through several transformations such as threshold-based filtering, joining, sequencing, and well-known aggregation functions (Figure.14).

CEP provides techniques and tools to reveal complicated reasoning about large-scale domain-specific software systems[81]. CEP is usually deployed with the aim of

**Figure 14:** Complex-Event Processing Conceptual View

decision making on runtime behaviors of systems. Some examples are database systems, network communication, intrusion detection, etc. Even though those complex SoS's produce terabytes of information, only a fraction of those are necessary for coming up with intelligent decision pertaining to certain properties of them. In order to achieve that, CEP engines correlate basic (or so called simple) events through special transformation and aggregation functions.

Decision support systems come in various flavors, one of which is CEP. Deriving complex decisions out of streams of simple events is the fundamental capability provided by CEP tools. Those simple events are defined as the immutable attributes of actions taking place in the system, which trigger a transition the system state [81]. Those simple events are then exercised against predefined event patterns (Figure.14) in order to reveal complex relations, which result in complex-events. According to the definition in [81], new events can only be appended to a stream, but not removed as they are immutable. A stream is the main building block of CEP in Esper. The computation in Esper engine is performed as the evaluation of streaming events.

CEP engines, such as Esper [81], allow us to declare event descriptions and specify patterns of interest by using normative representations. The normative representation in Esper is called Event Processing Language (EPL). EPL statements allow us to define events and operations on those such that we can monitor for complex-relations at runtime by deploy the statements on Esper CEP engine. EPL is a SQL-like declarative language. EPL constructs allow us to build complicated and interrelated query statements that will reveal complex-events implicitly occurring in the system. It is used for aggregating information and deriving knowledge from one or more event streams. EPL statements also enable us to join and merge event streams. Events are

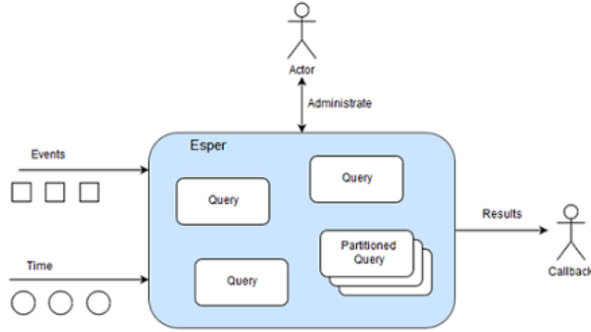


**Figure 15:** Esper is a container for EPL Statements[81]

inserted and processed as continuous streams of information. The basic syntax for EPL in Esper is as follows:

$$\begin{aligned} SELECT &< select_list > \\ FROM &< stream_def > \\ WHERE &< search\ conditions > \end{aligned} \tag{15}$$

The *select* clause in Eq.15 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.

$$select\ timestamp\ from\ TokenRing\ where\ moteId = 1 \tag{16}$$

Esper defines a stream as a temporally ordered sequences of events. First, a stream of simple events, representing fundamental immutable occurrences in a system, are injected into Esper engine; later, the Esper engine performs series of event transformations to determine whether event patterns of interest are satisfied or not (e.g.,

for detecting failures in a system). Essentially, Esper acts as a container for EPL statements Figure.15. EPL statements fundamentally specify queries that analyze events and time, and then detect situations. Esper provides a nest for EPL queries and organizes their lifecycle and execution.

The EPL syntax given in [81] will facilitate our efforts in specifying runtime monitors; thereby, generating those automatically by employing MDE techniques. The memory and processing costs of streams in Esper is negligible (i.e., considerably zero) [81].

## 4.2 Event Processing for Runtime Verification

RV of a system requires representing the specifications of the SUT in terms of monitoring constructs. Then, the RV framework observes the behavior of the SUT to conclude with particular verdicts of *Pass* and *Fail* for certain constraints. In this section, we present a transformation methodology that will guide the process of generating EPL statements from EC relations defined in Chapter.3, 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 suggest a process consisting of consecutive transformation steps. Figure.16 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.

Remember that in Section.3.2 we specified a concept of *context* for a enabling analysis of expected behavior of a particular service composition. The concept of context supports analyzing those simple events that are related to the correctness property at hand, and ignoring any other event occurring in the system. That's why

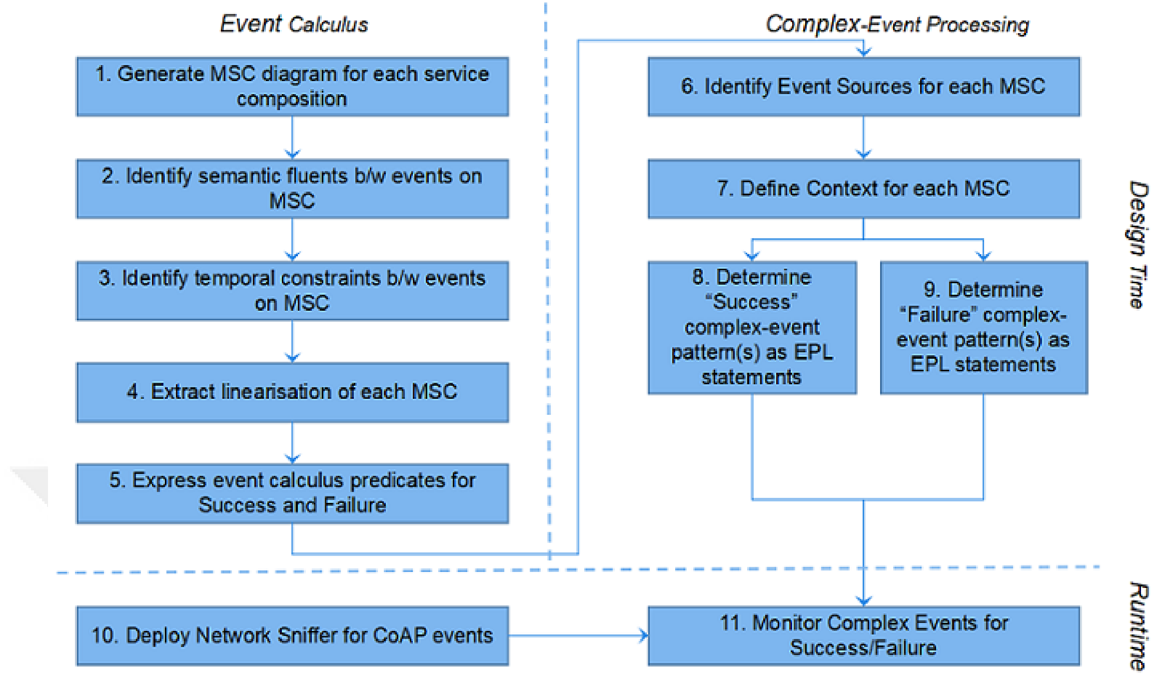


**Figure 16:** RV Process with Complex-Event Processing

CEP engines that we use should allow us to define a context for each service, and help us express complex-event relations for *Pass* and *Fail* verdicts. Esper provides a notion of context, which enables us to define a set of circumstances or facts that surround a particular event [81]. 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.4). 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. We can achieve those goals by employing certain constructs in Esper CEP engine as listed in List.4.1.

**Listing 4.1:** Event Processing Steps

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

Moreover, the *CoAP Events* sniffed from the network should be filtered such that only those events of the context for the particular correctness property, (note that, each unique service composition case constitutes a unique context) are processed at runtime monitors. 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 $Pass$ 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.11) is violated. Thus, we need a runtime monitor for each *not immediately follows* relation ($\neg f_i$).

**Listing 4.2:** EPL Statements for Context-Based RV of Figure.12

```
1 create context CtxSample
2 initiated by pattern
3 [every-distinct(startevent.srcId, startevent.dstId,
4  startevent.mId, startevent.uri) startevent = CoAPEvent(srcId = e1.id)]
       @inclusive
5  terminated by pattern [endevent = CoAPEvent(srcId = e4.id, destId = e4.
       destId, uri = e4.uri) or timer:interval(T)];
6
7 context CtxSample
```

```
8  create variant schema RVSpec as CoAPEvent;
9
10 context CtxSample
11 insert into RVSpec
12 select * from CoAPEvent where srcId = e1.id or  srcId = e2.id
13  or srcId = e3.id or srcId = e4.id;
```

Let us now give an example on how to write the EPL statements for an MSC by writing those for Figure.12. Code listing in List.4.2 summarizes the basic EPL statement that we use for determining a *Pass* 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 $(srcId, destId, msgId)$ for a CoAP message. As you can see from the code listing between lines 10 through 13, we insert only those events that are associated with the context into the variant stream. The code listing in List.4.3 provides an example of how to detect a pattern of events that observe the visual order as in MSC of Figure.12. 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 *Pass* verdict. $FAIL$ statement is the complement of *SUCCESS* statement in List.4.3, so it's not a comprehensive $Fail$ monitoring statement.

**Listing 4.3:** EPL Statements for Pass Verdict of Figure.12

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

```
5
6 @Name('SUCCESS')
7 context CtxSample
8 select count(*) as SuccessVerdict from RVSpec
9 match_recognize (
10  measures A.srcId as aId
11  pattern (A B C)
12  define
13   A as A.srcId = e1.id,
14   B as B.srcId = e2.id,
15   C as C.srcId = e3.id
16  ) output when terminated and context.endevent.srcId = e4.id;
```

The code listing in List.4.4 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 filter those events according to certain properties, such as event id. The $FAIL-1$ statement in List.4.4 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.

**Listing 4.4:** EPL Statements for Fail Verdict of Figure.12

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

Figure.17 shows how the reference architecture was designed in order to reveal failure cases from simple coap events. *CoAP Sniffer* listens to the IPv6 network for

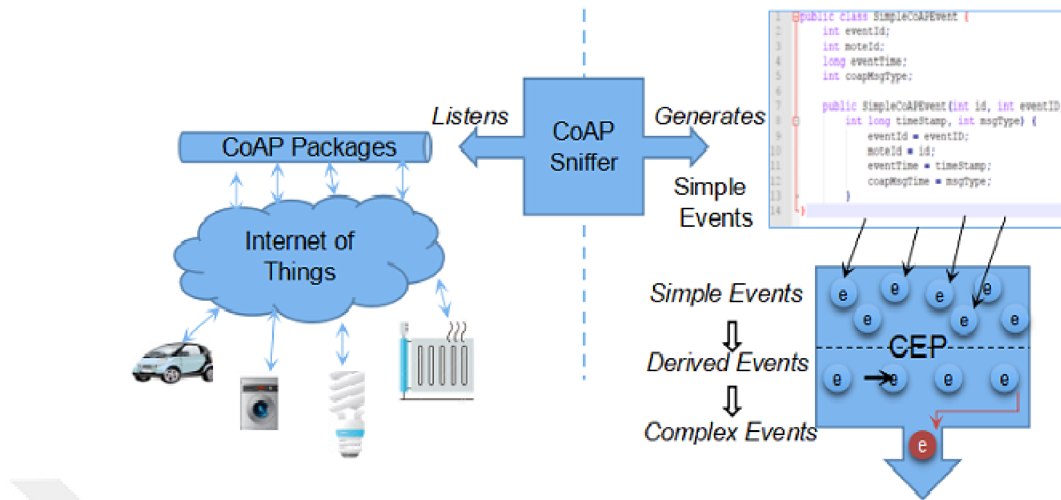


**Figure 17:** CEP Assisted Runtime Verification Reference Architecture

any CoAP communication, non-intrusively at runtime. Whenever 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 (Figure.17) 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).

## 4.3 CEP for IoT

Any software system with distributed function calls can also be described as a collection of events [107]. Interactions between components of an IoT system are also events, as we have demonstrated in Section.3.2. Simple event logs can be synthesized to reveal 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.

Esper engine runs along-side 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 also a built-in support for specifying EPL statements in Java language, thus enabling us to use Plain Old Java Objects (POJO) classes to represent events and event relations.

### 4.3.1 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 Figure.18, 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 Figure.18) to a specified IPv6 network [108] 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 [2]. Such events are then sent to a CEP engine for detection of abnormal situations.

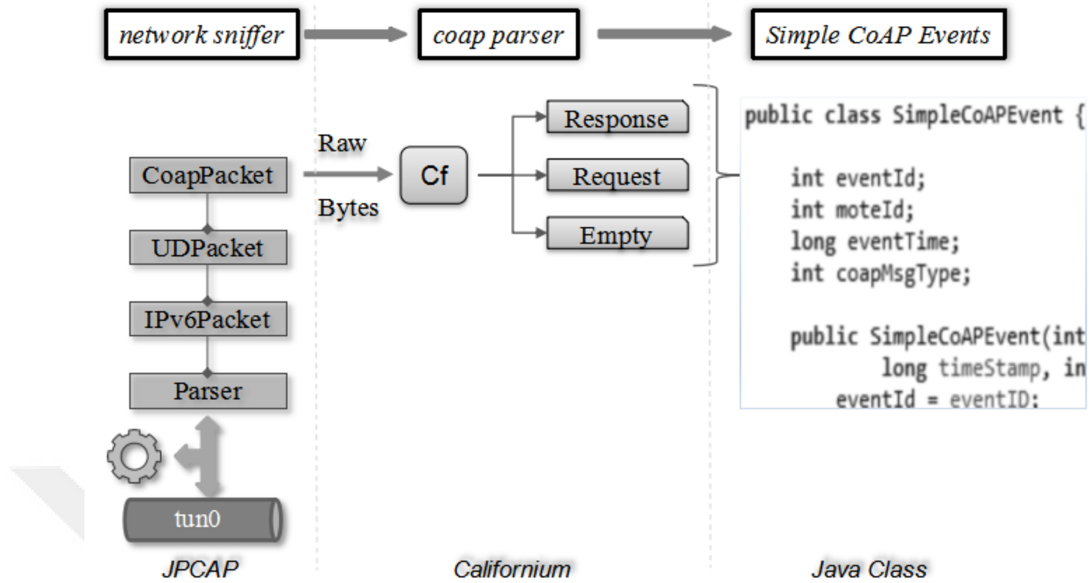


**Figure 18:** Event Generation out of CoAP Messages

Our CoAP parser (Figure.18) utilizes an open-source Java network packet capture library, named JPcap [109], for intercepting packets exchanged on an interface. Basic events are produced by CoAP Parser by utilizing Californium open-source library [110]. 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.

The architecture for event generator in Figure.18 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 [2]. 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.

## 4.4 Case Study: WTRP

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 [111], which comes with implementation of many protocols such as CoAP and a simulation environment, called Cooja [95]. 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 Figure.13. 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 (Eq.14), 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 $m_2 - m_1 = 1$ (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 $m_2 - m_1 \neq 1$ 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 [81] 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.

Figure.19 demonstrates how the EPL statements are organized in order to yield a verdict about the order of ownership relation. Each CoAP Message intercepted by $CoapParser$ is injected into Esper as a simple event of type $TokenEvent$ that is and extension of $CoapEvent$ class. Each simple event is decorated with *event id* and *timestamp*. 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 *timestamp* values in order to avoid false positives. After that, all token events are maintained in an ordered event window, which is an Esper EPL specific element that enables

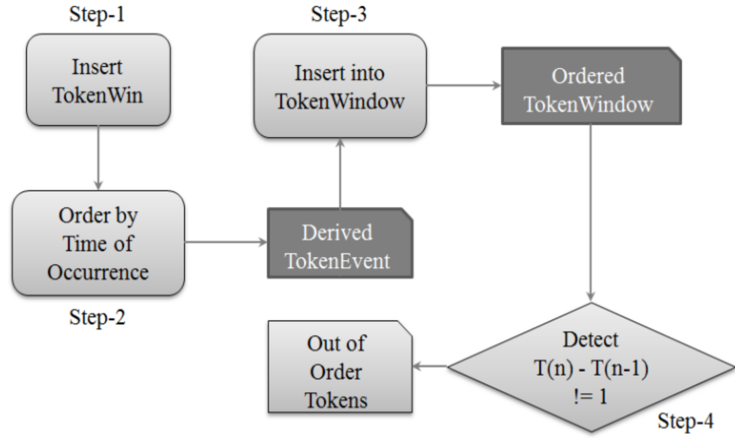


**Figure 19:** EPL Statement Flow for Figure.13

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 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.

Esper maintains events in streams, and EPL statements (Eq.17,18,19) are exerted on stream of events to retrieve special relations or properties in those streams. We did not used the concept of context in these EPL statements, because the experiment was conducted in an isolated simulation environment, where no out of context events was occurring.

$$\begin{aligned} &create\ window\ OrderedUniqueTokenWin.std : unique(eventId).win : keepall() \\ &\qquad as\ select\ eventId,\ moteId,\ eventTime\ from\ TokenEvent \end{aligned} \tag{17}$$

$$\begin{aligned} &insert\ into\ OrderedUniqueTokenWin\ select\ eventId,\ moteId, \\ &\qquad eventTime\ from\ TokenEvent\ order\ by\ eventTime \end{aligned} \tag{18}$$

$$select\ a.eventId\ as\ aEvId,\ a.moteId\ as\ aId,\ b.moteId\ as\ bId,$$
$$b.eventId\ as\ bEvId\ from\ pattern\ [every\ a = OrderedUniqueTokenWin \tag{19}$$
$$\rightarrow\ b = OrderedUniqueTokenWin]\ where\ (((b.moteId\ -\ a.moteId)\ != \ 1))$$

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 $\neg HoldsAt(OO(t), t) = True$.

### 4.4.1 Implementation

We have used a prominent real-time operating system that is particularly designed for resource constrained embedded devices, Contiki [111]. The WTRP is implemented on each mote of type Zolertia-Z1 according to [91]. The simulation in Figure.13 is run on Cooja [95]. 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 Section.4.3. Captured messages are converted to $SimpleCoapEvent$ instances and injected into

**Table 5:** 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 $t$ |

Esper CEP engine. A failure case is randomly generated by adding a random seeded error function in WTRP algorithms of motes, which causes a mote to transmit a message without owning the token. The failure situation is diagnosed by monitoring for a sequence of messages that violate Eq.11. 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.

The predicates and corresponding meanings for WTRP are given in Table.5. $Diff(m_i, m_j)$ 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.3.3. *OO(Diff,t)* is a predicate function that determines whether or not the order of ownership relation is preserved during successive transmissions at any time *t*. Thus, having a $OO(Diff, t) \neq 1$ at time $t$ indicates a $Fail$. As explained in Section.3.3 *OO* relation must satisfy $HoldAt(OO, t)\ \forall\ t$.

**Listing 4.5:** EPL Statement for Success in WTRP

```
1 @Name('SUCCESS')
2 context CtxSample
3 select count(*) as SuccessVerdict from RVSpec
4 match_recognize (
5  measures A.mId as a_Id
6  pattern (A B C D)
7  define
```

```
8     A as A.moteId = m1,
9     B as B.moteId = A.moteId + 1,
10    C as C.moteId = B.moteId + 1,
11    D as D.moteId = C.moteId + 1
12  ) output when terminated and context.endevent.moteId = m5;
```

The EPL statements in List.4.3 can be tailored to reflect event properties specific to the WTRP case, but we can also represent those new predicates (i.e., semantic relations for $Diff$) in EPL statements as shown in List.4.5. As for the $Fail$ cases, we can use the EPL statements shown in List.4.6, as well as those in List.4.4. 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 $t_j > t_i \land \nexists t_k | t_i < t_k < t_j$. Therefore, the difference between mote ids can also be expressed as $m_j = m_i + 1$.

**Listing 4.6:** EPL Statement for Failure in WTRP

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

## 4.5 Discussion

In this experiment, the token was passed around randomly among motes every period, instead of following a certain order. 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; and

each simulation takes 10 minutes to complete. Note that as the period of broadcast decreases the amount of message exchanges increases, thereby increasing the traffic on the network. This enabled us to observe the robustness of our solution under varying loads of events (Table.6).

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 6:** Performance Results

| **Period** | **CoojaEvt** | **EsperEvt** | **CoojaFlt** | **EsperFlt** | **Perf(%)** |
|---|---|---|---|---|---|
| 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 |

The results of simulation are obtained 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. *CoojaEvt* column shows total number of events produced in Cooja environment, while *EsperEvt* indicates number of simple *TokenEvent* events inserted in Esper. *CoojaFlt* shows total number of token ring protocol failures occurred in Cooja, and *EsperFlt* shows total number of violations detected in Esper. *Period* 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 $EsperFlt/CoojaFlt$, which is given in $Perf$ column in the table. As seen on the Table.6, the performance of our verification approach reaches almost 100% success

rate. We observed that most of the failures generated in Cooja are detected in Esper. 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; 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 ported to 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 in Section.3.3.

## *4.6 Related Work*

Software is said to have a failure when the observed behavior deviates from the required behavior [112]. 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 [113, 114, 115]. Run-time verification is an approach that differentiates from classic verification (i.e. theorem proving, model checking, and testing) [17] 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 [17]. 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 [116], 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 [117] were designed to reveal interoperability issues between different implementations of the CoAP draft [2], and consequently unearth the standard specifications. In [8], 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 [8, 105, 118] 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 [8], 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 [105], 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.

## 4.7 Conclusion

In this chapter, we devised a novel solution for non-intrusive, non-instrumented, and online runtime verification of IoT systems; which builds on the EC proposed in Chapter.3. Our approach is an event-based solution which exploits RESTful service paradigm employed in CoAP messaging model, thereby enabling utilization of Esper CEP tool. 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.

# CHAPTER V

# RUNTIME VERIFICATION AT THE EDGE OF THINGS

IoT devices gained more prevalence in ambient assisted living (AAL) systems ([119]). Reliability of AAL systems is critical especially in assuring the safety and well-being of elderly people. Runtime verification (RV) is described as checking whether the observed behavior of a system conforms to its expected behavior (Section.2.1). RV techniques generally involves heavy formal methods; thus, it is poorly utilized in the industry. Therefore, we propose a democratization of RV for IoT systems by presenting a model-driven engineering (MDE) approach. In order to support modeling expected behaviors of an IoT system, we leverage the modeling profiles provided by Unified Modeling Language (UML); we particularly use UML version 2.5 [3], as it was the latest one available at the time of our research. A domain-specific modeling profile has been extended over UML, which later allows us to make use of Sequence Diagrams (SDs) for specifying the expected behaviors of any IoT system. Later, the expected behaviors are translated into runtime monitor statements expressed in Event-Processing Language (EPL), which are proposed to have executed at the edge of the IoT network. We further demonstrate our contributions on a sample AAL system.

The increasing computing capacity of *things* enabled equipping those with full-fledged operating systems and special-purpose software. Such improvements in IoT systems will allow us to deploy certain processing tasks at the edge of an IoT network, thereby providing expedited data processing without compromising data integrity and security. An edge computing solution aims to provide a seamless integration between IoT and cloud computing platform by using the processing power of gateway devices

at the edge of a network to filter, pre-process, aggregate or score IoT data. Edge computing solutions utilize the power and flexibility of cloud services to run complex analytic on those data and, in a feedback loop, support decisions and actions about and on the physical world.

A majority of research on RV utilizes formal methods and logic programming such as Linear Temporal Logic (LTL/T-LTL) [17] for specifying runtime monitors, that's why it has been considered intimidating by the practitioners for more than 50 years. However, RV is a fundamental verification method that increases the user's confidence on a SUT, by exercising certain monitoring techniques on execution traces of SUT. Considering the fact that an IoT SoS may be composed of hundreds or thousands of devices that may be manufactured by a multitude of manufacturers, the reliability of such a large scale system could be further assured by leveraging a *feasible* RV solution. RV complements reliability of IoT systems by black-box testing approach, which is one of the most appropriate testing approaches for such heterogenous large scale systems [17].

IoT systems are increasingly deployed with service-oriented architecture (SOA) principles, due to its ease of implementation and vast community adoption [120]. Thus, RV solutions that favor service-centric [17] nature of those systems are more applicable. In this chapter, we complement our research on RV of event-based IoT systems with a MDE framework that is proposed as a facilitator of proliferation for RV of such large-scale SoS's.

Interoperability is often defined as ability of two or more systems interacting through provided interfaces of collaborating party [121]. In emerging domains such as IoT, special purpose application layer protocols are proposed to govern the interoperability issues. Constrained-application protocol (CoAP) [2] is developed with best practices in the industry, which specifies the interaction model and messaging formats between IoT endpoints. Its fundamental messaging model inherits basic principles of

RESTful API model in [19].

We presented a proof of concept implementation of MDE-based RV for IoT systems in [14]; however, the work fails to define a deployment model that supports practical application. We extended our initial findings on representing basic interoperability issues of IoT systems using CoAP. We improve our research by proposing a meta-model for CoAP by extending the UML profile, and utilizing the meta-model for automatically generating runtime monitors from behavioral models of IoT SUT.

The chapter is structured such that Section.5.1 introduces MDE approaches we have contributed for RV of IoT systems. Then, we describe the design of a reference deployment architecture for RV-as-a-Service (RuVaaS) framework on edge of things in Section.5.2. Section.5.3 demonstrates the proposed architecture on a case of Ambient-Assisted Living system implemented on IoT running CoAP. The merits of the contributions are discussed in Section.5.4, then, we provide a brief recap of related work on MDE.

## 5.1 Modeling Runtime Verification for IoT Systems

IoT presents a new computing phenomenon for such devices that are smart yet resource-constrained. Those *things* involve heterogeneous day-to-day smart objects [5], which aim to seamlessly construct new services and applications through untethered autonomous M2M collaborations. Interoperability is a major challenge in achieving such a goal, as there might occur unprecedented interactions between those objects. It is an issue in such SoS's that are composed of subsystems with various communication protocols, application interfaces. Interoperability challenge at protocol stack of network layers governs the issues related with implementation of protocol specification; for example, CoAP-based devices must be interoperable with respect to the CoAP standard [2, 8].

IoT SoS's are intrinsically hot swap systems, such that an endpoint can be replaced

with another one providing the same services with the same device configurations. However, the new endpoint may be flawed in certain features, inhibiting its expected behaviors. Thereby, it may incur runtime failures even though the overall system design is verified before its deployment. In order to support RV services at such operating environments, we propose the MDE approach that leverage event-based IoT systems, where interoperability can be assured through testing against certain service agreements at runtime. It can be investigated further in findings that we present in Section.2.3 that interoperability is an ongoing issue in such SoS's that heterogeneous subsystems compose a large-scale SoS as in IoT.

IoT systems usually consist of commercial-off-the shelf (COTS) products with nearly no knowledge of internal implementation details, we promote a black-box testing approach for providing interoperability. We propose that a model-driven engineering approach that leverages the RESTful-like application layer interaction model of CoAP-based IoT systems should facilitate interoperability testing efforts. MDE as been utilized in several domains [122]. We demonstrate the applicability of MDE in IoT domain through implementation of a case study with Papyrus [123] modeling tool. Our previous work on runtime verification of IoT systems [12] has demonstrated that an IoT system can be described in terms of simple events occurring in the system. Thereby, we proposed a verification approach that utilized CEP technique. In this thesis, we further that research with a MDE solution that allows automatically generating test cases from sequence diagrams in a UML model.

### 5.1.1 UML Profile Extension

Papyrus is a modeling tool developed on Eclipse Modeling Project [123]. There are a plethora of applications supporting UML standards; however, those tools are commercial applications that are driven by the proprietary companies. Eclipse open-source platform has democratized the modeling experience to masses by incubating a

Model Development Tools (MDT) project. Since its initiation MDT project has been the main modeling choice especially in open-source community.

Papyrus is a graphical model editing tool that builds on Eclipse MDT capabilities, such as GEF, GMF and EMF [123]. The main advantage of using Papyrus over using principle projects of Eclipse (GEF, GMF, EMF) is that it is so flexible that it enables collaboration of different editors. This capability allows extending the Papyrus graphical editing window by introducing pluggable diagram editors. Another advantage is its inherent capability in developing DSM tools with such an ease for those domains that can be extended on UML Profile [3]. Those capabilities encouraged us to choose Papyrus as the basis for developing a DSM for CoAP-based IoT systems.

### 5.1.2 Model-driven engineering for Interoperability

Software intensive systems has increasingly being developed with component-based architectures [124]. IoT systems are no exception to that adoption in the industry. Particularly, application layer protocols such as CoAP have made it possible to treat such systems purely as service-oriented systems. Monitoring of services in SOA systems has been valuable for post mortem analysis [125]. Thus, we will define how model-driven engineering approach can be used for describing the service interactions, and consequently facilitate interoperability testing at runtime.

Interoperability issue might be present at various levels of communication layers. Application level interoperability can be defined to occur between service calls, such that endpoint-A calls a service that exists in endpoint-B with the correct signature and parameters, and also data interoperability. In this research we address solely service call interoperability.

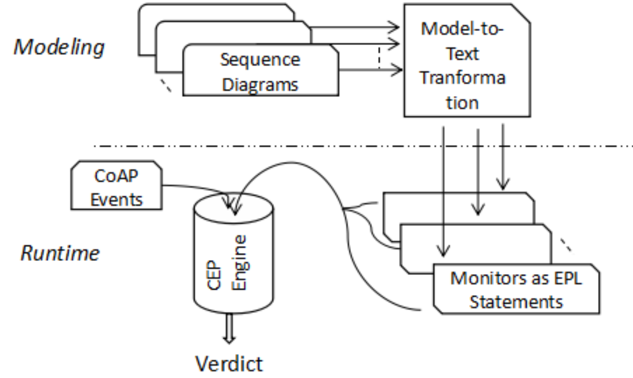


**Figure 20:** MDE Process for Interoperability

Figure.20 illustrates the concept of using MDE in RV of IoT systems. A system integrator first (Step-1) needs to model the interoperability scenarios in sequence diagrams. Each diagram captures expected behavior of an individual service of a particular endpoint in terms of CoAP interactions with other endpoints; thus, there must be as much sequence diagrams as the number of services provided by an endpoint in an IoT system, in order for fully covering all behaviors. The interactions are represented as asynchronous message exchanges in the sequence diagrams. In second step, a model-to-text transformation algorithm is exerted on each sequence diagram to transform event relations into EPL statements. EPL statements act as runtime monitors. EPL statement is an executable special purpose instruction written in Event Processing Language (EPL) of Esper CEP engine [81]. EPL statements are implemented as (see Section.5.1.4) Java classes that can be run on any Java compatible platform. Those are registered (in Step-3) with an Esper engine running either on a stand-alone endpoint acting as an edge computing solution for interoperability testing, or it can be provided as a service over a cloud implementation. In step 4, the CoAP events that are captured from the running network by means of sniffing (Section.4.3) it passively are injected into the Esper engine for monitoring through complex-event processing. The *Verdict* can either be *Pass* or *Fail* depending on the result.

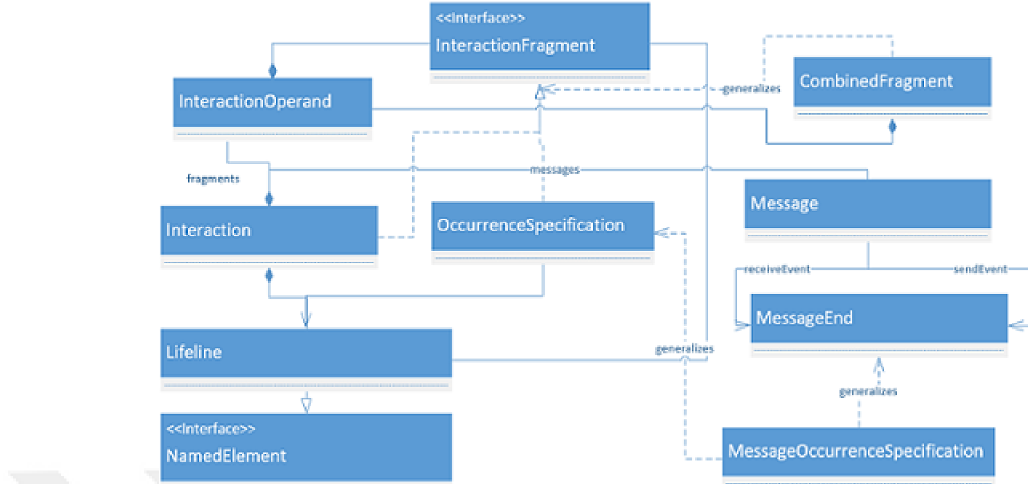


**Figure 21:** UML Elements Syntactical Relations (*adapted from* [3])

### 5.1.3 A model-based RV solution for IoT systems

The modeling of IoT systems as smart objects has already been investigated in [126]. In this section we propose an extension to SD of UML 2.5 profile [3] that will be used in modeling expected behavior of CoAP-based IoT systems. SD is a type of *Interaction Diagram*, which captures the sequence of exchanges of messages between entities. Modeling behavior of a system in terms of messages not only facilitates development activities, but also allows deriving test scenarios for observing the expected behavior.

A SD may be composed of one or more *lifelines*, *messages* exchanged between those lifelines, and *combined fragments*. A *Lifeline* (Figure.21) represent instances of UML *NamedElement*s on a SD. Each lifeline is represented with a vertical line on a SD, and messages are drawn between those vertical lifelines to designate occurrence of message ends at particular entities. A *MessageOccurrenceSpecification* element indicates occurrence of a *sendEvent* or a *receiveEvent* of a particular *Message*. In UML terminology, a SD is a type of *Interaction*, which is fundamentally an *InteractionFragment*. An *InteractionFragment* might contain nested *Fragments* so as to capture more complex interactions. In such scenarios *CombinedFragments* are used to enforce

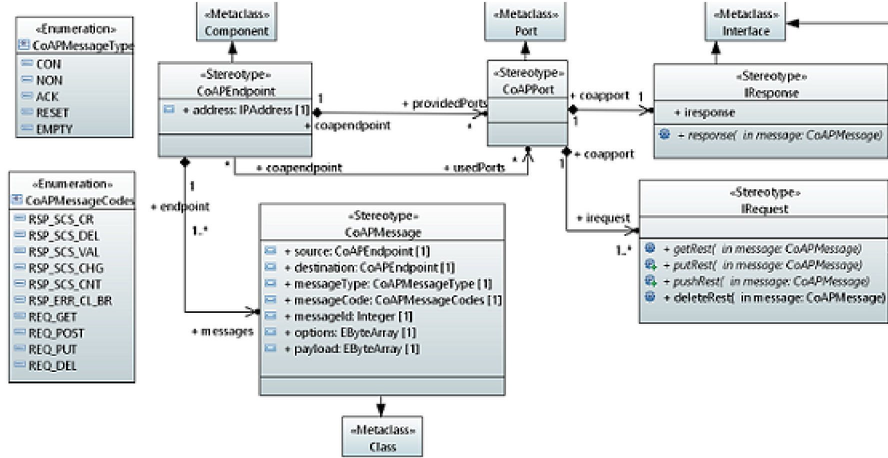


**Figure 22:** CoAP UML Profile Extension

certain constraints or exert different aggregation operations on sets of *MessageOccurrenceSpecification*'s. Events on a *Lifeline* are partially ordered with respect to the visual order of *MessageOccurrenceSpecifications* appearing on the vertical lines. The linearization of *sendEvent*'s on a certain SD gives us an expected behavior of event sequences when the SUT runs. We previously investigated the linearization of SD's [13], which help us in transforming SD's into runtime monitors. Therefore, we will not re-visit that concept.

In order to transform linearization of event occurrences, we need to apply M2T algorithms on each SD. We will assume that a SD represents behavior for a single service composition. A service composition is realized through collaboration of one or more CoAP endpoints.

The CoAP metamodel is implemented as an extension to UML 2.5 profile for SD. The profile captures essential characteristics of a CoAP endpoint, which allows us to use CoAP specific *Stereotypes* in modeling behavior on SD's. UML provides *Components* as a type of *NamedElements* (Figure.21), which enables abstracting away internal structure details of a system, and focusing design on its interfaces. Components interact with other components through functionalities that are interfaced through *Port*'s. We extend *Component* element of UML profile to represent a *CoAPEndpoint*

(Figure.22). A new stereotype *CoAPMessage* is specified for organizing CoAP specific message contents by extending the *Class* metaclass of UML profile. *CoAPMessageType* and *CoAPMessageCodes* enumerations allow describing the attributes of a *CoAPMessage* instance with respect to the network packaged captured by CoAP-Sniffer (Figure.28). *Port* element is extended to represent a *CoAPPort*, which provides interfaces through *IResponse* or *IRequest*. *IRequest* and *IResponse* elements in Figure.22 represent service access points by extending *Interface* element of UML profile.

EPL statements are automatically generated for a particular SD by utilizing the techniques we proposed in this section. We extend those algorithms to incorporate transformation of SD's with *CombineFragments* that are constrained by *assert InteractionOperand*. *assert* is an operand that allows us to exert an assertion constraint on a set of events. The visual order of events occurring on Lifelines covered by an assert *CombinedFragment* are accepted as the only acceptable *valid traces* of a particular SD; any other combination of event sequences would constitute *invalid traces*. Thereby, the M2T methods generate EPL statements that either monitor for occurrence of valid trace or invalid traces. We designed and developed an Eclipse plug-in to incorporate and distribute proposed solution (i.e., extended profile, M2T algorithms). The details of M2T algorithms can be found in Section.5.1, where we succinctly explained how to cover $SUCCESS$ and $FAILURE$ EPL statements (List.4.3, List.4.3).

### 5.1.4 Implementation

The example implementation assumes a healthcare system based on research in [4]. They present a case study on interoperability testing for HL7 systems with a sample hardware reference implementation. The communication model is based on CoAP (Figure.23).

In Section.3.3 we showed that an IoT system with CoAP can be expressed in terms

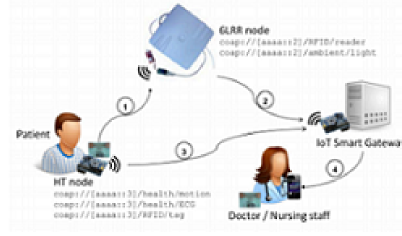


**Figure 23:** Healthcare Interoperability[4]

of *send events* in the system. Thus, we can represent a patient consent scenario with a sequence diagram as in Figure.24. For ensuring privacy, a doctor must first request patient's consent for observing health data (e.g., ECG) ($m_1$). After the patient grants the consent ($m_2$), the doctor can ask to observe the patient data ($m_3$). After that, the sensor on patient can periodically send the measured data ($m_4$). Each $e_i$ represents a *send event* for corresponding message $m_i$.

$$Follows(e_i, t_i, e_j, t_j) \equiv$$
$$\exists e_i, t_i, e_j, t_j Happens(e_i, t_i) \wedge Happens(e_j, t_j) \wedge (t_i < t_j) \quad (20)$$

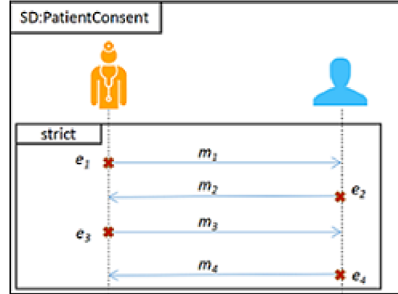


**Figure 24:** Patient Consent Sequence Diagram

The system in Figure.24 can be represented in terms of events (Eq.20) by using *Follows* relations as described in Chapter.3. Eq.20 states that $e_i$ must be followed by $e_j$ if they appear sequentially on the sequence diagram (e.g., $e_2$ follows $e_1$). Thus, by observing if each sequential pair of ($e_i$, $e_j$) at runtime satisfies *Follows* relation we can conclude that interoperability patient consent requirement. In order to conclude

with a *Pass* verdict (Eq.21), we must have $e_1 \prec e_2 \prec e_3 \prec e_4$, where $\prec$ denotes the precedence relation. For a *Fail* result (Eq.22), a $e_i \nprec e_j$ must hold for $(i, j) \in \{(1, 2), (2, 3), (3, 4)\}$, where $\nprec$ represents doesn't precede relation. Eq.22 states that the CEP engine must select all the complex events that occur as a result of $m_1$ is followed by either $m_3$ or $m_4$ message before an $m_2$ event occurs in order to indicate a failure situation. The same rule can be extended for other messages as well.

$$
\begin{aligned}
&select\ 'SUCCESS' from\ HealthEvent\ match_recognize( \\
&\quad measures\ A.mId\ as\ a_id, B.mId\ as\ b_id, C.mId\ as\ c_id, D.mId\ as\ d_id \\
&\quad pattern\ (A\ B\ C\ D)\ define\ A\ as\ A.mId = m_1, B\ as\ B.mId = m_2, \\
&\qquad C\ as\ C.mId = m_3, D' as D.mId = m_4); \qquad (21)
\end{aligned}
$$

```
1  for each CombinedFragment F in InteractionDiagram ID
2      if InteractionOperator equals 'strict'
3          Print EPL Statement of "SUCCESS" monitor by traversing
4          MATCH_RECOGNIZE PATTERN(M1 M2 M3 M4)
5          for each MessageOccurrenceSpecification M of ID
6              Print M
```

**Figure 25:** Algorithm for Generating EPL Statement of Success Verdict

$$
\begin{aligned}
&select\ 'FAIL', m1\ from\ pattern\ [every m1 = HealthEvent(mId = m1) - > \\
&\quad ((m3 = HealthEvent(mId = m3)\ or\ m4 = HealthEvent(mId = m4))\ and \\
&\qquad not\ m2 = HealthEvent(mId = m2))]; \qquad (22)
\end{aligned}
$$

```
1   for each CombinedFragment F in InteractionDiagram ID
2       if InteractionOperator equals 'strict'
3           Print EPL Statement of "FAIL" monitor
4           EVERY
5           for each MessageOccurrenceSpecification MI of ID
6               Print MI
7               for each MessageOccurrenceSpecification MJ of ID
8                  if (MJ - MI) > 1
9                       Print MJ or
10                 else
11                      Print "and not" MJ
```

**Figure 26:** Algorithm for Generating EPL Statement of Fail Verdict

Figure.25 and Figure.26 list algorithms for generating EPL statements for *Pass* and *Fail* cases of interoperability testing in Acceleo. Acceleo runs over XMI[1] definitions of sequence diagrams, and generates Java classes containing corresponding EPL statements (Eq.21 and Eq.22). After executing M2T code in Papyrus [123], a Java class containing a similar EPL statement as in Eq.21 is generated. This EPL statement selects all the matching sequences of messages as described in Figure.24.

The solution framework can be extended to other scenarios by following the procedure and implementation details described in Section.5.1. The event relations logic, how to sniff a network for CoAP packets through a CoAP sniffer, and how to run runtime monitors as EPL statements are explained in [12]. Note that, the solution framework would be applicable to both online and offline testing provided that raw CoAP packets are injected into the CoAP Sniffer. This solution enables for observing interoperability of IoT systems at runtime.

## 5.2 Verification at the Edge of Things

Cloud computing has drastically shifted the information technologies services development and deployment practices; thereby, it facilitated utilization of distributed systems by concealing complexities of resource provisions behind service-oriented delivery and deployment models. IoT systems have tremendously increased demand for

[1]https://www.omg.org/spec/XMI/About-XMI/

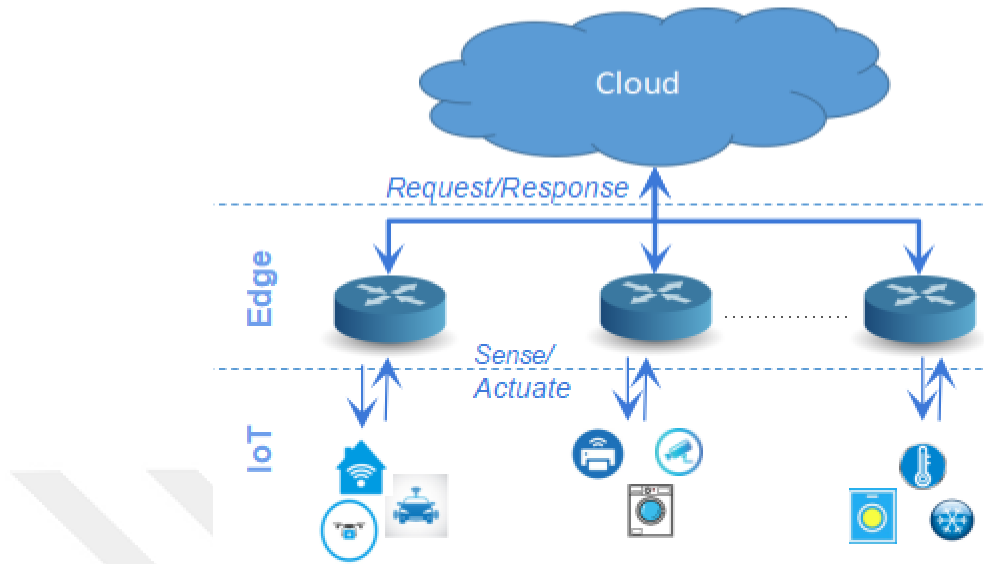


**Figure 27:** Edge Computing Paradigm

computing power, network bandwidth and performance capabilities; because, those systems continuously generate sensor data and engage in network communication more frequently than conventional computing systems. Even though the data analytic capabilities on cloud systems are more advanced than a traditional network gateway, the quantity of service requests and corresponding responds puts unprecedented load on the system capabilities. Besides, IoT systems demand fast results for data analytic as they are promised to seamlessly integrate IP enabled things. Thus, cloud computing falls short of providing services with necessary quality of service agreements.

Since the introduction of IoT paradigm, demand for high-speed data processing approaches has increased. As a result of that demand along with high volume of IoT data to be processed necessitated a more economical approach for data analytics, which recognizes the limitations of network bandwidth. Edge computing paradigm presents itself as a utility at the edge of the network, where data is produced [127]. Edge of things layer as illustrated in *Edge* layer of Figure.27 not only acts as border

routers connecting things to the cloud, but also as computing units that perform various functionalities such as data storage, caching, processing, which facilitate IoT and cloud convergence. Data stream processing can be performed locally, thus enabling faster analysis of sensor data. Speed of analysis becomes more critical in healthcare systems where data stream processing can be utilized for determining medical emergencies.

Edge computing converges IoT and cloud computing by addressing most of the subtle issues at the edge of the network [128]. Those issues are; *(i)bandwidth overhead*; an IoT system might generate gigabytes of data in matters of second (e.g., a Boeing jet generates 5 GB of data per second [127]). Bursting that data onto Internet for data analytic at a cloud facility would deplete the available bandwidth very quickly; *(ii)network latency*; IoT systems are destined to function in a real-time fashion, thus requiring high-speed data processing for sensory data analytic; processing such operations on the cloud incurs extra latency to network traffic; *(iii)privacy*; some application domains requires asserting privacy precautions on processing of sensor data (e.g., especially in healthcare informatics area); and, *(iv)security*; critical IoT data has to be secured against possible attacks both in transit and at rest. Transferring such data over the Internet would expose them to new vulnerabilities.

In following subsections we explain how to leverage border routers/gateways that enable CoAP communication between IoT endpoints and the Internet as as a runtime verification platform at the edge of things. We further explain how we extend the UML profile so as to support automatically generating runtime monitors as EPL statements for CoAP behaviors. Our verification approach differs from others in the literature that are mainly built on instrumenting the SUT, in the cost of memory and processing power [8]. EPL statements generated as RV monitors are deployed on edge devices, thus does not incur any additional cost in terms of memory and processing power to the SUT.

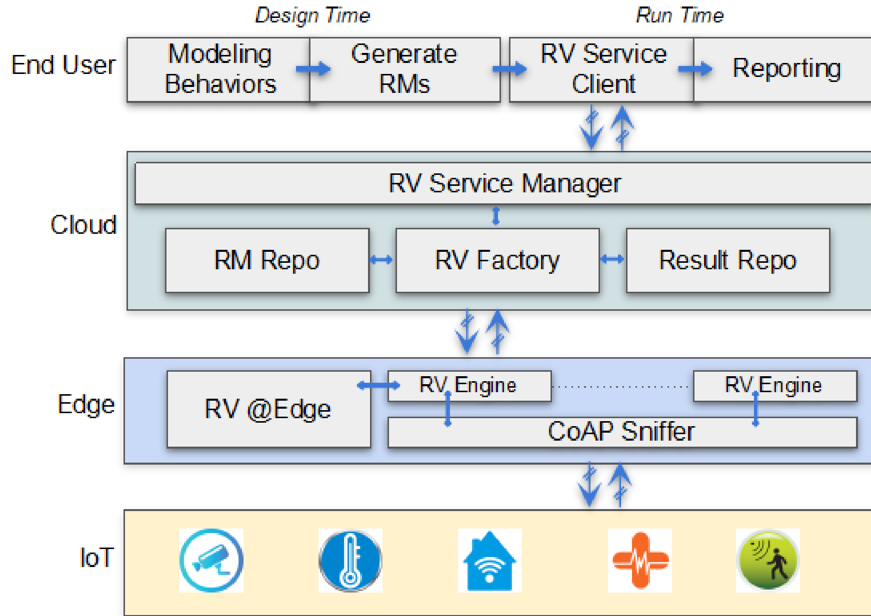


**Figure 28:** Conceptual Reference Verification Architecture

### 5.2.1 A Reference Verification Architecture

Cloud computing providers are motivated to support services that support various use cases, and maintain them in a service pool, which supports on-demand service delivery [129]. Reviewing the literature on cloud computing, we can acknowledge benefits of different deployment and delivery methods (Section.2.3). This section does not explain how to construct a cloud service for RV, but specify the allocation of functionality of a RV solution on the interplay of IoT-Edge-Cloud triplet.

Figure.28 captures general concepts of how to deploy a RV service on edge of things, which is presented to end users via a cloud service manager (*RV Service Manager*). We will briefly describe the layers of functionality allocated to each component of the architecture, then we will elaborate on *Modeling Behaviors*, *Generating RMs* and *RV @Edge* designs in the next subsection. The descriptions for the components of proposed architecture in Figure.28 is given in Table.7.

The user is expected to depict the expected behavior of the SUT via UML at *Design Time*. After the expected behavior of an interaction is modeled on a SD,

**Table 7:** RV@Edge Architecture Components of Figure.28

| Component | Description |
|---|---|
| *RV Service Manager* | Initiates a RV session through *RV Factory*. Uploads generated monitors to the cloud and conducts the RV session |
| *RV Factory* | Enables configuration of a RV session (i.e., IP addresses of endpoints, port numbers for services). Interacts with *RM Repository* and *Result Repository* components for bookkeeping of EPL statements and results. Manages a set of *RV @Edge* components. Each *RV @Edge* component can be started/stopped independent of other instances. |
| *RM Repository* | Enables re-utilization of generated runtime monitors |
| *Result Repository* | Allows recording of a particular RV session |
| *RV @Edge* | Implements a RV management software for a set of IoT endpoints, at the edge of network (i.e., border router or gateway). Enables conducting RV for different SDs independently of each other by allocating each as a session on a different CEP engine. |
| *RV Engine* | Enables conducting RV for a particular SD independently of other behaviors as a RV session. Runtime monitors as EPL statements are uploaded to *RV Engine*, the results of monitoring actions are reported back to *RV @Edge*. |
| *CoAP Sniffer* | Implements a passive network sniffer for CoAP [119] |

model-to-text (M2T) operations are performed on SD so as to derive runtime monitors. *RV Service Client* allows the user to initiate a RV session through *RV Service Manager* at *Run Time*. After the service is initiated, end user needs to upload runtime monitors onto the cloud. The results of RV session are monitored through the *Reporting* component.

Since Esper engine is developed with Java technologies, it is recommended that *RV @Edge*, *RV Engine* and *CoAP Sniffer* components are also implemented with Java. Note that our reference architecture addresses *Online RV*, meaning that CoAP events are generated out of live CoAP network traffic. However, *Edge* layer of this reference architecture can also be implemented on a cloud platform so that it allows performing post-mortem RV on a simulation of recorded/logged IoT network traffic.

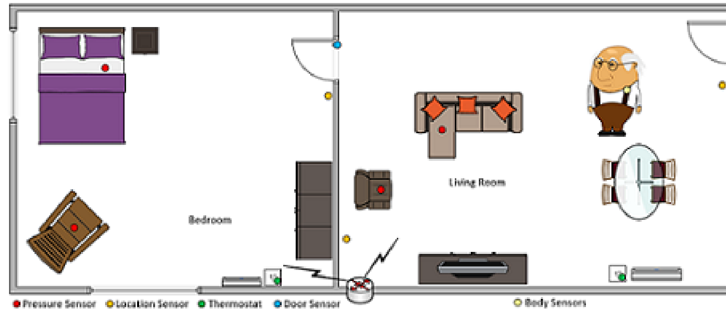


**Figure 29:** IoT-enabled AAL Example

## 5.3 Case Study: Ambient-Assisted Living

IoT-enabled healthcare systems leverage on connectivity of many resources, which involves both in-house and on-person sensors/actuators, to conduct comprehensive data analytics and to deduce timely verdicts on wellbeing of a person. Timeliness is a crucial attribute of such systems, which inherently determines life or death situations. AAL environments are such healthcare informatics systems that are mostly utilized in providing reliability and safety services of elderly people (Figure.29) or people with cognitive impairment (CIP) [130].

AAL systems promise to *(i)* improve the quality of elderly living by accommodating certain habits into people's lives, such as reminder for daily sport routines as walking, medication reminder; *(ii)* non-intrusively monitor vital health data and daily activities of elderly people; *(ii)* facilitate immediate response in case of emergency situations. An AAL system designed to support caring for an elderly person or CIP might be composed of various sensors. Some of the sensors [130] that are most commonly deployed in such systems are *presence sensors*, *motion/location sensors*, *temperature sensors*, *open/close sensors*, and other body sensors.

The design and development of AAL systems must be meticulously carried out, because such systems are safety-critical systems, on which people's lives depend. Particularly, the functionality that handle emergency situations are more critical then others. That's why, it's expected that the designers of such systems observe the

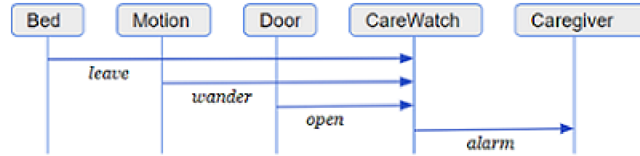


**Figure 30:** Sequence of Actions in CareWatch System

regulations and guidelines embodied for those systems. However, various faults and failures will eventually occur at runtime [130].

Emergency situations can be detected by implementing various analysis techniques that collect and/or aggregate sensor data, then analyze those to yield a decision [119]. The analysis can be carried out by using, for instance, rule-based reasoning techniques as in [119]. Failures conditions on such AAL systems might arise due to two main reasons: (i) Faulty sensor data, (ii) Erroneous analysis.

CareWatch project [130] is an AAL system devised to preserve the safety of CIP by avoiding unsupervised night time leaving of house, and enhancing caregiver sleep quality. The proposed AAL system not only ensures well-being of CIP, but also of caregiver. However, sleep quality of caregiver might be hindered by frequent interruptions generated by AAL system, due to false alarms. The runtime analysis can be carried out (Figure.30) on a set of multimodal events generated by *motion sensor*, *bed presence sensor* and *door sensor*. The system is specified to alert the caregiver in case of an unsupervised stepping out of the house by CIP, and avoid any false alarms.

As the authors express in [130], the system reliability is still an open issue, even though they lowered down the rate of false positive/negatives to 10%. This evaluation intensifies the need of a RV solution for AAL systems. At the time of CareWatch project, IoT paradigm was not contributed to the literature; that's why, the original CareWatch project was not built on IoT. However, it is still a viable example for demonstrating the effectiveness of using edge of things concepts by assuming that it was deployed on an IoT network.

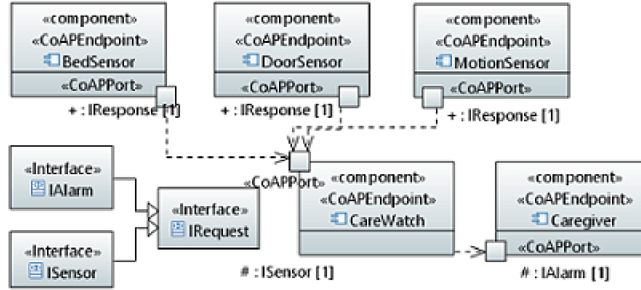


**Figure 31:** CareWatch Component View

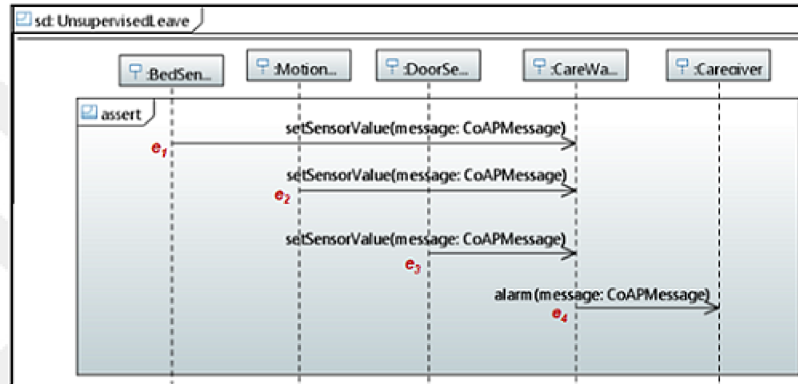


**Figure 32:** Sequence Diagram of CareWatch

This section demonstrates how the MDE approach in Section.5.2 can be demonstrated on the motivating example described below. The SUT, CareWatch, will be monitored for verifying that it doesn't generate neither false negative nor false positive. The interactions of CareWatch system can be designed by first specifying components constituting the system, then designing the expected behavior amongst them on a SD.

The CareWatch SD is displayed in Figure.32. The SD is developed in Papyrus modeling tool [123]. Notice the difference between Figure.32 and Figure.30. The SD demonstrates a single scenario of generating an alarm in case of CIP's unsupervised leaving of the house. *assert* keyword on the upper left corner of the figure states that a *SUCCESS* verdict can yield only if the event sequences in the *assert Combined-Fragment* observe the same visual order as they appear on the diagram. Thus, any deviation of order would result in *FAILURE*.

The sequence of events in Figure.32 specifies an expected behavior of *CareWatch* system for a specific scenario. The scenario is initiated by occurrence of a *sendEvent_setSensorValue* event emitted by the *BedSensor*. This event is registered with *CareWatch* component for further processing. The *sendEvent_alarm* event by *CareWatch* is generated only when *sendEvent_setSensorValue* event by *BedSensor* is immediately followed by *sendEvent_set-SensorValue* of *Motionsensor* and *sendEvent_setSensorValue* of *DoorSensor*. EPLs for this scenario is shown in List.5.1. Notice that each run of the scenario is designated by a contextual representation of the domain. Every time a *BedSensor* event occurs, a new context is generated to verify that each unique *BedSensor* event has been given consideration for detecting unsupervised nighttime leaving house. The statement labeled *FAIL-X* is an example of detecting an event $e_i$ not immediately following another event $e_j$. We needed to add another statement for failure, *FailTOut*, which yields a failure if an *alarm* is not generated after some period of time.

**Listing 5.1:** Generated EPLs for CareWatch System

```
1  context CtxLeaveUnsupervised
2  create variant schema CWRV as CoAPEvent;
3
4  context CtxLeaveUnsupervised
5  insert into CWRV
6  select * from CoAPEvent where name = "BedSensor" or "MotionSensor"
7   or "DoorSensor" or "CareWatch";
8
9  @Name('FailX')
10 context CtxLeaveUnsupervised
11 select count(*) FailOneVerdict from pattern [
12 every
13 rsp1 = CWRV(name="BedSensor")->((rsp2=CWRV(name="DoorSensor")
14  or rsp2=CWRV(name="CareWatch")) and
```

```
15     not rsp3=CWRV(name="MotionSensor"))]
16   output when terminated;
17
18   @Name('FailTOut')
19   context CtxLeaveUnsupervised
20   select count(*) FailOneVerdict from pattern [
21   every
22   rsp1 = CWRV(name="BedSensor")->((rsp2=CWRV(name="DoorSensor")->
23    (rsp3=CWRV(name="MotionSensor"))->rsp4=CWRV(name="CareWatch")))
24     in time.within(THRESHOLD)]
25   output when terminated;
26
27   @Name('FailLast')
28   context CtxLeaveUnsupervised
29   select context.endevent from CWRV.std:lastevent
30   output snapshot when terminated;
31
32   @Name('Success')
33   context CtxLeaveUnsupervised
34   select count(*) as SuccessVerdict from CWRV
35   match_recognize (
36   measures A.name as a_name
37   pattern (A B C)
38   define
39   A as A.name = "BedSensor",
40   B as B.name = "MotionSensor",
41   C as C.name = "DoorSensor"
42   ) output when terminated and context.endevent.name = "CareWatch";
```

## 5.4 Discussion

As shown in Figure.32, there are 4 distinct *sendEvent* instances in the SD, and 3 distinct event pairs $(e_i, e_j)$ that have to follow each other immediately on the SD.

Referring to the event calculus we explored in [12], we can calculate the total number of EPL statements required for monitoring all possible combinations for both *Pass* and *Failure* verdicts. List.5.1 indicates that a single EPL statement suffices to observe a *Pass* verdict over a pair of events $(e_i, e_j)$. However, in order to yield a *Failure* verdict on the SD it takes three different type of EPL statements (i.e., $FailX, FailLast, FailTOut$). A $FailX$ statement is necessary for each consecutive pair of events on the SD (e.g., $(e_1, e_2)$). A single $FailLast$ statement for the entire SD is necessary to cover for the case of reaching end of stream before the last event arrives for the specified context (List.5.1; and a single statement suffices for detecting timeout situation (i.e., $FailTOut$). If we assume that there are $N$ distinct events on a particular SD with $N-1$ event pairs that are required to satisfy *Follows* relation as stated in [12], then the total number of EPL statements required can be calculated as in Eq.23.

$$EPLCount = 2 + 2 * (N - 1) \tag{23}$$

Therefore, the total number of EPLs required to conduct a RV through CEP is linearly proportionate to the number of distinct *sendEvent* instances in the SD (Figure.33). The x-axis in the figure represents the total number of unique events that are denoted in the SD (e.g., $e_1, e_2, e_3, e_4$ in Figure.32); whilst the y-axis represents total number of EPLs required to fully cover all cases of a RV for a particular SD. If we were to write EPL statements manually, then the effectiveness of using CEP techniques would have been hindered. That's why, using M2T algorithms along with an MDE approach for automatically generating those EPLs presents a feasible solution. Our future work could also include query plan optimizations among EPL statements, since they serve shared event streams and continuous queries. Besides, CEP utilities enable us to leverage built-in event patterns and event transformation methods for mutating simple events towards complex-events with such ease.

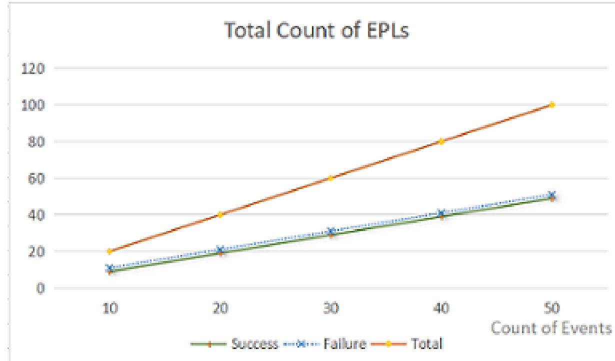


**Figure 33:** Increasing number of EPL statements 23

The validation of using CEP as a RV solution has been investigated in our work in [12], where the performance results indicated that nearly 100% fidelity can be achieved by using CEP.

## *5.5 Related Work*

A black-box testing approach for assuring interoperability assumes that the individual components are thoroughly tested by the manufacturer. But, when it comes to the complexities of the integrated heterogeneous system of systems, the runtime actions of the system might be overlooked with black-box testing. In [122] Wu et al. proposes using Unified Modeling Language (UML) [3] to express the expected behavior of a component-based software. They utilize interaction diagrams to capture functionality expected from the system. They explain how UML interaction diagrams can be used to extract the context-dependence and content-dependence relations so as to use in deciding if test cases are comprehensive or not. The research doesn't provide any guidance through implementation nor the automation of a model-driven engineering approach.

Internet community is majorly built around web services concept, thus interoperability of those distributed and heterogeneous services is an ongoing challenge. Bertolino et al. [131] proposed an audition framework for solving this problem. They

extend the UDDI registries so that the services registered to a directory is audited before it is registered. Thereby, they validate the claimed behavior of the service before such services with the same UDDI registry can collaborate with proclaimed service contracts. This is a solid contribution in service registry coordination, however it lacks to observe the runtime behavior of services.

In [132], Smythe discusses that using a the modeling approach in development of a distributed service oriented system facilitates both implementation and testing efforts. The interoperability tests are automatically generated through a series of XML Metadata Interchange (XMI) transformations over a UML model of the system. Our approach also use XMI transformations in order to facilitate runtime EPL statements for interoperability testing.

In [8] authors proposes a new solution for certification of products, which involves conformance testing of IoT devices with respect to CoAP standards [2]. In their approach, they first record the live network traffic, and save them in files for post processing. When the system test run finishes, they collect those record files and apply post mortem tests on those logs so as to find any deviation in the CoAP communication primitives from the standard specification. The test cases are prerecorded according to the standard specification. Since, they operate on recorded log files, the approach does not scale well to runtime (online) interoperability testing. Moreover, they focus solely on protocol implementation interoperability, so the solution does not scale well for application specific interaction models.

The purpose of RV is detecting monitorable deviations or similarities between expected and observed behaviors of a SUT. MDE operates on abstract models of a SUT, either dynamic or static, and it utilizes the information provided by the environment for the system when generating test case [133]. They present a succint taxonomy of MDE approach in the literature. The authors in [133] conducts a classification on the main concepts of MDE. They lead the classification in three main categories: (i)

*model specification*, (ii) *test generation*, (iii) *test execution*. Our MDE model classifies under *online RV* tool that proposes a *functional* operational paradigm, which is used for *test case specifications* by using *constraints* for runtime monitor generation.

A most recent study in [134] delved into MDE as a service for IoT systems. They aim to provide a general modeling design approach that will proliferate MDE in IoT domain by exploiting the standards (i.e., CoAP, MQTT, HTTP). They propose a verification process through a MDE approach that produces test cases. However, their contribution is limited to the APIs (application programming interfaces) of FI-WARE project, which is a EU funded FP7-ICT project [135]. Their contribution differs from our proposal in the communication paradigm adopted, namely, they build on HTTP, whereas our approach motivates RV of CoAP systems. Yet another difference between the two is in the categories of MDE that those propose to address. Their solution goes under *test case generation and execution framework*, whereas our solution classifies under *runtime monitor generation and RV framework*.

Monitoring is an indispensible element of RV; yet, there have been less consideration of involving capabilities for that in edge computing domain, according to a recent study by Taherizadeh et.al. [136]. The research emphasizes that in order to provide a decent level of quality in service provisions, it is required to have a monitoring solution. In our research, we propose a monitoring solution at the edge of a network so as to alleviate overheads imposed by interconnecting IoT and the cloud for RV and monitoring purposes.

Systems that are designed according to SOA principles are inherently nominee to be tested with MDE, because SOA systems can be easily specified with model elements [132]. Smythe et.al. achieve automatic test generation for interoperability testing. The automation is carried out through a succession of XMI transformations applied on a UML model. We, too, perform a series of M2T transformations based on XMI, but we generate runtime monitors in terms of EPL statements.

Devices that are manufactured by observing certain standards have to be tested for compliance with those standards. A recent study by Chen et.al. [8] investigated conformance test practices for IoT devices, which are employed with an implementation of CoAP standard [2]. They stimulate a SUT with certain input values such that it yields predefined results as defined by the standard. Their approach is an offline verification approach, which runs a system against recorded test cases. Moreover, it aims at verification only the protocol implementation, not a general purpose IoT application. Nevertheless, our proposed solution enables verification of an IoT at runtime (online), and also can be extended to any IoT application with CoAP as its application layer protocol.

## 5.6 Conclusion

Edge computing is an emerging phenomenon that complements distributed computing in IoT domain by facilitating seamless integration of IoT and cloud computing capabilities. Deploying IoT systems in an edge computing architecture mitigates the issues of network latency, computing power and battery life of resource constrained devices. We proposed a novel edge of things solution to RV problem of IoT based AAL systems. Our approach utilizes the EC proposed in Chapter.3 and event-based specification of IoT services by exploiting the CoAP messaging model. The event-based specification is achieved through describing an event calculus approach for CoAP. Then, a MDE framework is presented, which leverages event calculus representation, that enables to design interaction model between IoT endpoints. Those models are then utilized to automatically generate runtime monitors in terms of EPL statements and used in CEP engines. We believe that our solution presents a coherent design, development and execution process for RV, consequently contributing to the democratization of it. Our future work will focus on complementing RV for IoT concept with other application layer protocols.

The chapter presented a framework for facilitating interoperability testing of IoT systems. It promotes interoperability through model-driven engineering techniques. We utilized sequence diagrams in order to describe expected interactions between endpoints. Then, those are extracted from the diagram so as to compose a set of runtime monitors in terms of CEP EPL statements. We demonstrated the applicability of this approach with a case study on a healthcare system. Our future work will address a more comprehensive interoperability approach by involving structural and semantics testing; which will present a domain-specific metamodel for CoAP-based IoT systems, and the framework will be incorporated in a cloud service such that the solution can be used as a service. We believe that we can model the interactions between IoT systems with thorough event relations, which elaborates on the application layer protocol behavior.

# CHAPTER VI

# CONCLUSION

Runtime verification (RV) has usually been considered as a method of formally proving the correctness of a software system, either through model checking or theorem proving techniques. Although, there are several convenient RV solutions that are used for individual program verification, there was not a feasible one that could be easily adapted in industrial scenarios. We envision an IoT SoS as consisting of many subsystems, each of which is an embedded system with limited computing resources. Those IoT systems are assumed to employ CoAP application layer protocol, which enables those devices to engage in interactions with each other through predefined services (provided and required).

A major contribution of our thesis might be presumed to present a streamlined process of runtime verification for IoT systems. Even though, RV can be intimidating to some practitioners as well as researchers, we believe that our solution approach encourage those to practice RV at many different domains that can be expressed in terms of events, especially those designed with SOA principles. We transform a hard-to-analyze problem (RV) into an event domain that help us to express complex relations between interacting devices in terms of simple events representing communication actions taking place in the SUT.

The EC that we proposed for transforming raw CoAP messages into simple events is a novel contribution in using EC for IoT, which addresses the second research question that we defined in Section.1.2. An IoT system consisting of several endpoints actually constitutes a SoS's, with each subsystem being a CoAP-enabled IoT device;

therefore, a particular IoT system performs its expected behavior as a result of composition of various sub-services supplied by constituting endpoints. Thus, the EC proposed in the thesis captures the essentials of a RESTful-like system architecture; which is expected to expedite its proliferation. The complex relations such as *Follows, Timeout, Semantic* are articulated in the corresponding chapters.

Representing complex-relations between IoT endpoints in a human-readable form later enabled us to utilize complex-event processing techniques that already exist in the literature. We adopted an open-source CEP engine, Esper, for running runtime monitors as event-processing agents, which addresses the first research question in Section.1.2. By plugging a passive and non-intrusive CoAP Sniffer into an IoT network, we were able to conduct an online RV experience without incurring any communication overhead. The CoAP Sniffer in fact acted as a mediator of CEP, because it transforms raw CoAP messages into Esper events. We later exert several pattern detection rules on streams of CoAP events in the system by utilizing EPL statements.

The democratization of the RV operation is achieved through contributing a model-driven engineering approach for CoAP-based IoT systems. Our contributions in MDE consist of a DSM for CoAP domain and M2T algorithms for generating runtime monitors in the form of EPL statements from an Eclipse based modeling tool. The representation of a CoAP in a MSC has enabled us to analogically use sequence diagrams in UML profile. We were able to extend the UML profile in order to incorporate the DSM we've defined for CoAP. The contribution on MDE has addressed the third research question in Section.1.2.

To the best of our knowledge, this thesis contributes utilization of Event Calculus, as well as Complex-Event Processing techniques in runtime verification of IoT systems for the very first time. We believe that using such a human-readable formal specification approach will alleviate the intimidation factor of RV as perceived by

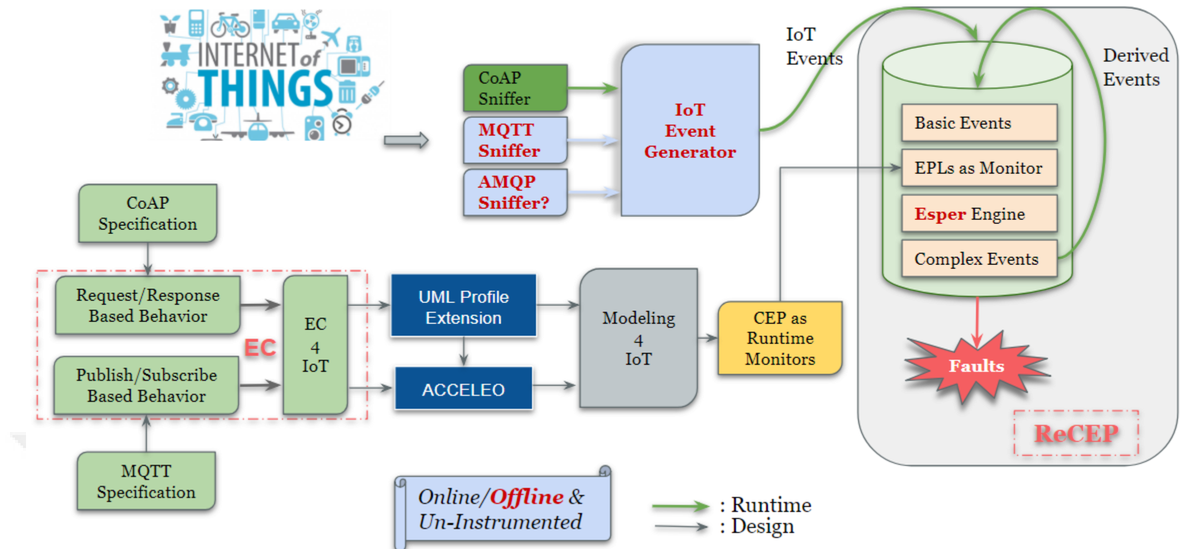


**Figure 34:** Future Work

the industry, and whereby will promote its adoption. However, further research is required to investigate how EC addresses the state space explosion problem that is frequently experienced when conducting RV.

On a broader perspective, the skeleton of our contributions is democratization of RV for RESTful-like systems by using MDE techniques integrated with CEP methods. That single contribution is a candidate of paradigm shift in reliability of IoT systems. As the Internet claims its major role in the next industry revolution, SOA-based systems are going to be employed much more than ever. Because, having a RESTful-like interface simplifies the engineering of such large-scale systems. That's why, we seek to extend our work in providing similar capabilities to other domains, as well as IoT systems with other application-layer protocols such as MQTT.

Even though the theoretical foundation of the paper has been proven to be valid, the contributions of the thesis in terms of case studies and implementation can be argued to be limited to certain tools that we have used, such as Papyrus and Esper. Moreover, the messaging behavior that exhibit only *Request/Response* model limits

the proliferation of the contributions to those domains that adopt the same interaction pattern. However, there are other modeling and CEP tools, as well as various application layer protocols to cover for. That's why we foresee to extend our research to reflect upon those limitations and provide feasible solutions as in Figure.34. We continue our research such that the framework will incorporate a more generic event calculus that captures both *Request/Response* and *Push/Pull* behaviors as in CoAP and MQTT protocols, respectively. In order to achieve that we will modify the network event generator into an IoT event generator, and the UML profile extension will be improved so as to involve the M2T algorithms for *Push/Pull* behavior, as well.

# Bibliography

[1] S. Kubler, "Building an IoT OPen innovation Ecosystem for connected smart objects," 2015.

[2] Z. Shelby, K. Hartke, and C. Bormann, "The constrained application protocol (CoAP)," 2014.

[3] "UML - Unified Modeling Language. version 2.5," may 2015.

[4] R. D. Snelick, L. E. Gebase, and M. Skall, "Conformance Testing and Interoperability: A Case Study in Healthcare Data Exchange," in *International Conference on Software Engineering Research and Practice, SERP'08*, (Las Vegas, NV), 2008.

[5] G. Fortino and P. Trunfio, *Internet of Things Based on Smart Objects, Technology, Middleware and Applications.* Italy: Springer, 2014.

[6] R. T. Fielding and R. N. Taylor, "Principled Design of the Modern Web Architecture," *ACM Trans. Internet Technol.*, vol. 2, pp. 115–150, may 2002.

[7] Core, "IETF Constrained RESTful Environments (core) Working Group," 2018.

[8] 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, 2013.

[9] J. P. Bowen and M. G. Hinchey, "Seven More Myths of Formal Methods," *IEEE Software*, vol. 12, pp. 34–41, jul 1995.

[10] "ITU Z.120: Message Sequence Charts," tech. rep., 2011.

[11] K. Incki, I. Ari, and H. Sozer, "A Survey of Software Testing in the Cloud," in *2012 IEEE Sixth International Conference on Software Security and Reliability Companion*, pp. 18–23, 2012.

[12] K. Incki, I. Ari, and H. Sozer, "Runtime verification of IoT systems using Complex Event Processing," in *Proceedings of the 2017 IEEE 14th International Conference on Networking, Sensing and Control, ICNSC 2017*, 2017.

[13] K. Incki and I. Ari, "A Novel Runtime Verification Solution for IoT Systems," *IEEE Access*, 2018.

[14] K. Incki and I. Ari, "Observing Interoperability of IoT Systems Through Model-Based Testing," in *3rd EAI International Conference on Interoperability in IoT*, 2017.

[15] K. Incki and I. Ari, "Democratization of Runtime Verication for IoT Systems: An Edge Computing Approach (accepted)," *Elsevier Computers and Electrical Engineering*, no. Special, 2018.

[16] G. Fortino, W. Russo, C. Savaglio, W. Shen, and M. Zhou, "Agent-Oriented Cooperative Smart Objects: From IoT System Design to Implementation," *IEEE Transactions on Systems, Man, and Cybernetics: Systems*, pp. 1–18, 2017.

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

[18] E. T. Mueller, "Event Calculus," 2008.

[19] R. Fielding, *Architectural Styles and the Design of Network-based Software Architectures*. PhD thesis, University of Irvine, 2000.

[20] L. Richardson and S. Ruby, *Restful Web Services*. O'Reilly, first ed., 2007.

[21] L. Lamport, "Time, Clocks, and the Ordering of Events in a Distributed System," *Communications of the ACM*, vol. 21, no. 7, pp. 558–565, 1978.

[22] A. F. V. Frederic P. Miller and M. John, *IBM Cp-40*. VDM Publishing, 2010.

[23] W. Kim, S. D. Kim, E. Lee, and S. Lee, "Adoption issues for cloud computing," in *Proceedings of the 7th International Conference on Advances in Mobile Computing and Multimedia*, MoMM '09, (New York, NY, USA), pp. 2–5, ACM, 2009.

[24] Y. Ridene and F. Barbier, "A model-driven approach for automating mobile applications testing," in *Proceedings of the 5th European Conference on Software Architecture: Companion Volume*, ECSA '11, (New York, NY, USA), pp. 9:1—-9:7, ACM, 2011.

[25] P. Mell and T. Grance, "The NIST Definition of Cloud Computing ( Draft ) Recommendations of the National Institute of Standards and Technology," *Nist Special Publication*, vol. 145, p. 7, 2011.

[26] M. A. Vouk, "Cloud Computing Issues , Research and Implementations," *Components*, pp. 31–40, 2008.

[27] L. M. Riungu, O. Taipale, and K. Smolander, "Software Testing as an Online Service: Observations from Practice," *2010 Third International Conference on Software Testing Verification and Validation Workshops*, pp. 418–423, 2010.

[28] L. Yu, W.-T. Tsai, X. Chen, L. Liu, Y. Zhao, L. Tang, and W. Zhao, "Testing as a Service over Cloud," in *Service Oriented System Engineering (SOSE), 2010 Fifth IEEE International Symposium on*, pp. 181–188, jun 2010.

[29] "UTest - Online Software Testing Services Community," 2012.

[30] S. Baride and K. Dutta, "A cloud based software testing paradigm for mobile applications," *ACM SIGSOFT Software Engineering Notes*, vol. 36, pp. 1–4, 2011.

[31] M. Staats and C. P\vas\vareanu, "Parallel symbolic execution for structural test generation," in *Proceedings of the 19th international symposium on Software testing and analysis*, ISSTA '10, (New York, NY, USA), pp. 183–194, ACM, 2010.

[32] Z. M. Jiang, "Automated analysis of load testing results," in *Proceedings of the 19th international symposium on Software testing and analysis*, ISSTA '10, (New York, NY, USA), pp. 143–146, ACM, 2010.

[33] G. Candea, S. Bucur, and C. Zamfir, "Automated software testing as a service," in *Proceedings of the 1st ACM symposium on Cloud computing*, SoCC '10, (New York, NY, USA), pp. 155–160, ACM, 2010.

[34] J. S. Rellermeyer, M. Duller, and G. Alonso, "Engineering the cloud from software modules," in *Proceedings of the 2009 ICSE Workshop on Software Engineering Challenges of Cloud Computing*, CLOUD '09, (Washington, DC, USA), pp. 32–37, IEEE Computer Society, 2009.

[35] L. Zhao, A. Liu, and J. Keung, "Evaluating Cloud Platform Architecture with the CARE Framework," in *Proceedings of the 2010 Asia Pacific Software Engineering Conference*, APSEC '10, (Washington, DC, USA), pp. 60–69, IEEE Computer Society, 2010.

[36] W. K. Chan, L. Mei, and Z. Zhang, "Modeling and testing of cloud applications," in *Services Computing Conference, 2009. APSCC 2009. IEEE Asia-Pacific*, pp. 111–118, 2009.

[37] A. I. Avetisyan, R. Campbell, I. Gupta, M. Heath, S. Ko, G. Ganger, M. Kozuch, D. O'Hallaron, M. Kunze, T. Kwan, K. Lai, M. Lyons, D. Milojicic, H. Y. Lee, Y. C. Soh, N. K. Ming, J.-Y. Luke, and H. Namgoong, "Open Cirrus: A Global Cloud Computing Testbed," *Computer*, vol. 43, no. 4, pp. 35–43, 2010.

[38] L. Riungu-kalliosaari, O. Taipale, and K. Smolander, "Testing in the Cloud : Exploring the Practice," *IEEE Software*, vol. PP, p. 1, 2011.

[39] N. Zhou, D. P. An, L.-J. Zhang, and C.-H. Wong, "Leveraging Cloud Platform for Custom Application Development," in *Proceedings of the 2011 IEEE International Conference on Services Computing*, SCC '11, (Washington, DC, USA), pp. 584–591, IEEE Computer Society, 2011.

[40] T. Vengattaraman, P. Dhavachelvan, and R. Baskaran, "A Model of Cloud Based Application Environment for Software Testing," *CoRR*, vol. abs/1004.1, p. XX, 2010.

[41] W. Jun and F. Meng, "Software Testing Based on Cloud Computing," in *Proceedings of the 2011 International Conference on Internet Computing and Information Services*, ICICIS '11, (Washington, DC, USA), pp. 176–178, IEEE Computer Society, 2011.

[42] N. Snellman, A. Ashraf, and I. Porres, "Towards Automatic Performance and Scalability Testing of Rich Internet Applications in the Cloud," in *Proceedings of the 2011 37th EUROMICRO Conference on Software Engineering and Advanced Applications*, SEAA '11, (Washington, DC, USA), pp. 161–169, IEEE Computer Society, 2011.

[43] T. Hanawa, H. Koizumi, T. Banzai, M. Sato, S. Miura, T. Ishii, and H. Takamizawa, "Customizing Virtual Machine with Fault Injector by Integrating with SpecC Device Model for a Software Testing Environment D-Cloud," in *Proceedings of the 2010 IEEE 16th Pacific Rim International Symposium on Dependable Computing*, PRDC '10, (Washington, DC, USA), pp. 47–54, IEEE Computer Society, 2010.

[44] T. Banzai, H. Koizumi, R. Kanbayashi, T. Imada, T. Hanawa, and M. Sato, "D-Cloud: Design of a Software Testing Environment for Reliable Distributed Systems Using Cloud Computing Technology," in *Proceedings of the 2010 10th IEEE/ACM International Conference on Cluster, Cloud and Grid Computing*, CCGRID '10, (Washington, DC, USA), pp. 631–636, IEEE Computer Society, 2010.

[45] T. Hanawa, T. Banzai, H. Koizumi, R. Kanbayashi, T. Imada, and M. Sato, "Large-Scale Software Testing Environment Using Cloud Computing Technology for Dependable Parallel and Distributed Systems," in *Proceedings of the 2010 Third International Conference on Software Testing, Verification, and Validation Workshops*, ICSTW '10, (Washington, DC, USA), pp. 428–433, IEEE Computer Society, 2010.

[46] S. R. S. Souza, M. A. S. Brito, R. A. Silva, P. S. L. Souza, and E. Zaluska, "Research in concurrent software testing: a systematic review," in *Proceedings of the Workshop on Parallel and Distributed Systems: Testing, Analysis, and Debugging*, PADTAD '11, (New York, NY, USA), pp. 1–5, ACM, 2011.

[47] W.-T. Tsai, P. Zhong, J. Balasooriya, Y. Chen, X. Bai, and J. Elston, "An Approach for Service Composition and Testing for Cloud Computing," in *Autonomous Decentralized Systems (ISADS), 2011 10th International Symposium on*, pp. 631–636, mar 2011.

[48] C. R. Senna, L. F. Bittencourt, and E. R. M. Madeira, “An environment for evaluation and testing of service workflow schedulers in clouds,” in *High Performance Computing and Simulation (HPCS), 2011 International Conference on*, pp. 301–307, jul 2011.

[49] A. F. Mohammad and H. Mcheick, “Cloud Services Testing: An Understanding,” *Procedia CS*, vol. 5, pp. 513–520, 2011.

[50] L. Zhang, T. Xie, N. Tillmann, J. de Halleux, X. Ma, and J. Lu, “Environment Modeling for Automated Testing of Cloud Applications,” *IEEE Software, Special Issue on Software Engineering for Cloud Computing*, vol. 1, p. xx, 2012.

[51] T. M. King and A. S. Ganti, “Migrating Autonomic Self-Testing to the Cloud,” in *Proceedings of the 2010 Third International Conference on Software Testing, Verification, and Validation Workshops*, ICSTW ’10, (Washington, DC, USA), pp. 438–443, IEEE Computer Society, 2010.

[52] T. Rings, J. Grabowski, and S. Schulz, “On the Standardization of a Testing Framework for Application Deploymenton Grid and Cloud Infrastructures,” in *Proceedings of the 2010 Second International Conference on Advances in System Testing and Validation Lifecycle*, VALID ’10, (Washington, DC, USA), pp. 99–107, IEEE Computer Society, 2010.

[53] L. M. Riungu, O. Taipale, and K. Smolander, “Research Issues for Software Testing in the Cloud,” in *Cloud Computing Technology and Science (CloudCom), 2010 IEEE Second International Conference on*, pp. 557–564, 2010.

[54] W.-T. Tsai, Q. Shao, Y. Huang, and X. Bai, “Towards a scalable and robust multi-tenancy SaaS,” in *Proceedings of the Second Asia-Pacific Symposium on Internetware*, Internetware ’10, (New York, NY, USA), pp. 8:1—-8:15, ACM, 2010.

[55] G. Wang, A. R. Butt, P. Pandey, and K. Gupta, “Using realistic simulation for performance analysis of mapreduce setups,” in *Proceedings of the 1st ACM workshop on Large-Scale system and application performance*, LSAP ’09, (New York, NY, USA), pp. 19–26, ACM, 2009.

[56] S. Gaisbauer, J. Kirschnick, N. Edwards, and J. Rolia, “VATS: Virtualized-Aware Automated Test Service,” in *Proceedings of the 2008 Fifth International Conference on Quantitative Evaluation of Systems*, (Washington, DC, USA), pp. 93–102, IEEE Computer Society, 2008.

[57] Y. Wang and J. Wei, “VIAF: Verification-Based Integrity Assurance Framework for MapReduce,” *2011 IEEE 4th International Conference on Cloud Computing*, pp. 300–307, 2011.

[58] P. Zech, “Risk-Based Security Testing in Cloud Computing Environments,” *2011 Fourth IEEE International Conference on Software Testing Verification and Validation*, pp. 411–414, 2011.

[59] X. Zhang, H. Liu, B. Li, X. Wang, H. Chen, and S. Wu, "Application-Oriented Remote Verification Trust Model in Cloud Computing," *2010 IEEE Second International Conference on Cloud Computing Technology and Science*, pp. 405–408, 2010.

[60] V. Tran, J. Keung, A. Liu, and A. Fekete, "Application migration to cloud: a taxonomy of critical factors," in *Proceedings of the 2nd International Workshop on Software Engineering for Cloud Computing*, SECLOUD '11, (New York, NY, USA), pp. 22–28, ACM, 2011.

[61] X. Ding, H. Huang, Y. Ruan, A. Shaikh, B. Peterson, and X. Zhang, "Splitter: a proxy-based approach for post-migration testing of web applications," in *Proceedings of the 5th European conference on Computer systems*, EuroSys '10, (New York, NY, USA), pp. 97–110, ACM, 2010.

[62] T. Parveen and S. R. Tilley, "When to Migrate Software Testing to the Cloud?," in *ICST Workshops*, pp. 424–427, IEEE Computer Society, 2010.

[63] P. Mohagheghi and T. Saether, "Software Engineering Challenges for Migration to the Service Cloud Paradigm: Ongoing Work in the REMICS Project," in *Proceedings of the 2011 IEEE World Congress on Services*, SERVICES '11, (Washington, DC, USA), pp. 507–514, IEEE Computer Society, 2011.

[64] J. H. Kim, S. M. Lee, D. S. Kim, and J. S. Park, "Performability Analysis of IaaS Cloud," in *Proceedings of the 2011 Fifth International Conference on Innovative Mobile and Internet Services in Ubiquitous Computing*, IMIS '11, (Washington, DC, USA), pp. 36–43, IEEE Computer Society, 2011.

[65] P. Joshi, H. S. Gunawi, and K. Sen, "PREFAIL: a programmable tool for multiple-failure injection," *SIGPLAN Not.*, vol. 46, pp. 171–188, oct 2011.

[66] C. Pham, D. Chen, Z. Kalbarczyk, and R. K. Iyer, "CloudVal: A framework for validation of virtualization environment in cloud infrastructure," in *Proceedings of the 2011 IEEE/IFIP 41st International Conference on Dependable Systems&Networks*, DSN '11, (Washington, DC, USA), pp. 189–196, IEEE Computer Society, 2011.

[67] L. Martignoni, R. Paleari, G. Fresi Roglia, and D. Bruschi, "Testing system virtual machines," in *Proceedings of the 19th international symposium on Software testing and analysis*, ISSTA '10, (New York, NY, USA), pp. 171–182, ACM, 2010.

[68] J. S. Bolin, J. B. Michael, and M.-T. Shing, "Cloud Computing Support for Collaboration and Communication in Enterprise-Wide Workflow Processes," *2011 IEEE World Congress on Services*, pp. 589–593, jul 2011.

[69] L. Ciortea, C. Zamfir, S. Bucur, V. Chipounov, and G. Candea, "Cloud9: a software testing service," *SIGOPS Oper. Syst. Rev.*, vol. 43, pp. 5–10, jan 2010.

[70] S. Bucur, V. Ureche, C. Zamfir, and G. Candea, "Parallel symbolic execution for automated real-world software testing," in *Proceedings of the sixth conference on Computer systems*, EuroSys '11, (New York, NY, USA), pp. 183–198, ACM, 2011.

[71] Y. Kim and M. Kim, "SCORE: a scalable concolic testing tool for reliable embedded software," in *Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering*, ESEC/FSE '11, (New York, NY, USA), pp. 420–423, ACM, 2011.

[72] G. Chang, E. Law, and S. Malhotra, "Demonstration of LMMP automated performance testing using cloud computing architecture," in *Proceedings of the 2nd International Workshop on Software Engineering for Cloud Computing*, SECLOUD '11, (New York, NY, USA), p. 71, ACM, 2011.

[73] H. Liu and D. Orban, "Remote network labs: an on-demand network cloud for configuration testing," *SIGCOMM Comput. Commun. Rev.*, vol. 40, pp. 83–91, jan 2010.

[74] L. Yu, X. Li, and Z. Li, "Testing Tasks Management in Testing Cloud Environment," in *Proceedings of the 2011 IEEE 35th Annual Computer Software and Applications Conference*, COMPSAC '11, (Washington, DC, USA), pp. 76–85, IEEE Computer Society, 2011.

[75] J. M. Ferris, "Systems and Methods for Software Test Management in Cloud-Based Network," 2009.

[76] S. Hosono, H. Huang, T. Hara, Y. Shimomura, and T. Arai, "A Lifetime Supporting Framework for Cloud Applications," in *Proceedings of the 2010 IEEE 3rd International Conference on Cloud Computing*, CLOUD '10, (Washington, DC, USA), pp. 362–369, IEEE Computer Society, 2010.

[77] Y. Brun, R. Holmes, M. D. Ernst, and D. Notkin, "Speculative analysis: exploring future development states of software," in *Proceedings of the FSE/SDP workshop on Future of software engineering research*, FoSER '10, (New York, NY, USA), pp. 59–64, ACM, 2010.

[78] M. Oriol and F. Ullah, "YETI on the Cloud," in *Software Testing, Verification, and Validation Workshops (ICSTW), 2010 Third International Conference on*, pp. 434–437, apr 2010.

[79] R. Medhat, B. Bonakdarpour, D. Kumar, and S. Fischmeister, "Runtime Monitoring of Cyber-Physical Systems Under Timing and Memory Constraints," *ACM Trans. Embed. Comput. Syst.*, vol. 14, pp. 79:1—-79:29, oct 2015.

[80] S. Colin and L. Mariani, "Run-Time Verification," in *Model-Based Testing of Reactive Systems*, pp. 525–555, Springer, Berlin, Heidelberg, 2005.

[81] "Esper: Complex-Event Processing Engine," 2018.

[82] R. Kowalski and M. Sergot, "A logic-based calculus of events," *New Gener Comput*, vol. 4, no. 67, 1986.

[83] R. Kowalski, "Database updates in the event calculus," *The Journal of Logic Programming*, vol. 12, pp. 121–146, jan 1992.

[84] E. T. Mueller, "Automating Commonsense Reasoning Using the Event Calculus," *Commun. ACM*, vol. 52, pp. 113–117, jan 2009.

[85] G. Spanoudakis, C. Kloukinas, and K. Androutsopoulos, "Towards Security Monitoring Patterns," in *Proceedings of the 2007 ACM Symposium on Applied Computing*, SAC '07, (New York, NY, USA), pp. 1518–1525, ACM, 2007.

[86] W. Gaaloul, S. Bhiri, and M. Rouached, "Event-Based Design and Runtime Verification of Composite Service Transactional Behavior," *IEEE Transactions on Services Computing*, vol. 3, pp. 32–45, jan 2010.

[87] B. Mitchell, "Resolving race conditions in asynchronous partial order scenarios," *IEEE Transactions on Software Engineering*, vol. 31, pp. 767–784, sep 2005.

[88] R. Alur, K. Etessami, and M. Yannakakis, "Inference of message sequence charts," in *Proceedings of the 2000 International Conference on Software Engineering. ICSE 2000 the New Millennium*, pp. 304–313, jun 2000.

[89] H. Dan and R. M. Hierons, "Conformance Testing from Message Sequence Charts," in *2011 Fourth IEEE International Conference on Software Testing, Verification and Validation*, pp. 279–288, mar 2011.

[90] F. Belli, M. Beyazit, and A. Memon, "Testing is an Event-Centric Activity," in *2012 IEEE Sixth International Conference on Software Security and Reliability Companion*, pp. 198–206, jun 2012.

[91] F. Wei, X. Zhang, H. Xiao, and A. Men, "A modified wireless token ring protocol for wireless sensor network," in *2012 2nd International Conference on Consumer Electronics, Communications and Networks (CECNet)*, pp. 795–799, apr 2012.

[92] L. Lamport, R. Shostak, and M. Pease, "The Byzantine Generals Problem," *ACM Trans. Program. Lang. Syst.*, vol. 4, pp. 382–401, jul 1982.

[93] D. Lee, R. Attias, A. Puri, R. Sengupta, S. Tripakis, and P. Varaiya, "A wireless token ring protocol for intelligent transportation systems," in *ITSC 2001. 2001 IEEE Intelligent Transportation Systems. Proceedings (Cat. No.01TH8585)*, pp. 1152–1157, 2001.

[94] M. Ergen, D. Lee, R. Sengupta, and P. Varaiya, "WTRP - wireless token ring protocol," *IEEE Transactions on Vehicular Technology*, vol. 53, pp. 1863–1881, nov 2004.

[95] F. Osterlind, A. Dunkels, J. Eriksson, N. Finne, and T. Voigt, "Cross-Level Sensor Network Simulation with COOJA," in *Proceedings. 2006 31st IEEE Conference on Local Computer Networks*, pp. 641–648, nov 2006.

[96] K. Yu, Z. Chen, and W. Dong, "A Predictive Runtime Verification Framework for Cyber-Physical Systems," in *2014 IEEE Eighth International Conference on Software Security and Reliability-Companion*, pp. 223–227, jun 2014.

[97] A. Kane, "Runtime Monitoring for Safety Critical Embedded Systems," oct 2015.

[98] K. Ashton, "That 'Internet of Things' Thing," 2009.

[99] G. Fortino, A. Guerrieri, W. Russo, and C. Savaglio, "Integration of agent-based and Cloud Computing for the smart objects-oriented IoT," in *Proceedings of the 2014 IEEE 18th International Conference on Computer Supported Cooperative Work in Design (CSCWD)*, pp. 493–498, may 2014.

[100] T. Teixeria, S. Hachem, V. Issarny, and N. Georgantas, "Service Oriented Middleware for the Internet of Things: A Perspective," 2011.

[101] 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 Transactions on Services Computing*, vol. 3, pp. 223–235, jul 2010.

[102] F. Belqasmi, R. Glitho, and C. Fu, "RESTful web services for service provisioning in next-generation networks: a survey," *IEEE Communications Magazine*, vol. 49, pp. 66–73, dec 2011.

[103] W. Qin, Q. Li, L. Sun, H. Zhu, and Y. Liu, "RestThing: A Restful Web Service Infrastructure for Mash-Up Physical and Web Resources," in *2011 IFIP 9th International Conference on Embedded and Ubiquitous Computing*, pp. 197–204, oct 2011.

[104] D. C. Luckham, *The Power of Events: An Introduction to Complex Event Processing in Distributed Enterprise Systems*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 2001.

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

[106] C. Y. Chen, J. H. Fu, T. Sung, P. F. Wang, E. Jou, and M. W. Feng, "Complex event processing for the Internet of Things and its applications," in *2014 IEEE International Conference on Automation Science and Engineering (CASE)*, pp. 1144–1149, aug 2014.

[107] S. Qadeer and S. Tasiran, "Runtime verification of concurrency-specific correctness criteria," *International Journal on Software Tools for Technology Transfer*, vol. 14, no. 3, pp. 291–305, 2012.

[108] S. Deering and R. Hinden, "Internet Protocol, Version 6 (IPv6) Specification," RFC 2460, RFC Editor, dec 1998.

[109] "jpcap - a network packet capture library," 2017.

[110] M. Kovatsch, M. Lanter, and Z. Shelby, "Californium: Scalable cloud services for the Internet of Things with CoAP," in *2014 International Conference on the Internet of Things (IOT)*, pp. 1–6, oct 2014.

[111] A. Dunkels, B. Gronvall, and T. Voigt, "Contiki - a lightweight and flexible operating system for tiny networked sensors," in *29th Annual IEEE International Conference on Local Computer Networks*, pp. 455–462, nov 2004.

[112] N. Delgado, A. Q. Gates, and S. Roach, "A taxonomy and catalog of runtime software-fault monitoring tools," *IEEE Transactions on Software Engineering*, vol. 30, pp. 859–872, dec 2004.

[113] M. Sahinoglu, K. Incki, and M. Aktas, *Mobile application verification: A systematic mapping study*, vol. 9159. 2015.

[114] M. S. Aktas and M. Kapdan, "Structural Code Clone Detection Methodology Using Software Metrics," *International Journal of Software Engineering and Knowledge Engineering*, vol. 26, no. 02, pp. 307–332, 2016.

[115] M. Kapdan, M. S. Aktas, and M. Yigit, "On the Structural Code Clone Detection Problem: A Survey and Software Metric Based Approach," 2015.

[116] D. Bucur, "Temporal monitors for TinyOS," 2012.

[117] "ETSI: CoAP 4 Plugtests," 2014.

[118] C. Watterson and D. Heffernan, "Runtime verification and monitoring of embedded systems," *IET Software*, vol. 1, pp. 172–179, oct 2007.

[119] K. Incki and M. S. Aktas, "Improving Awareness in Ambient-Assisted Living Systems: Consolidated Data Stream Processing," in *International Conference on IoT Technologies for HealthCare. HealthyIoT 2016*, vol. 187, pp. 89–94, Springer, 2016.

[120] L. D. Xu, W. He, and S. Li, "Internet of Things in Industries: A Survey," *Transactions on Industrial Informatics*, vol. 10, no. 4, pp. 2233–2243, 2014.

[121] G. Aloi, G. Caliciuri, G. Fortino, R. Gravina, P. Pace, W. Russo, and C. Savaglio, "Enabling IoT interoperability through opportunistic smartphone-based mobile gateways," *Journal of Network and Computer Applications*, vol. 81, pp. 74–84, 2017.

[122] Y. Wu, M.-H. Chen, and J. Offutt, "UML-Based Integration Testing for Component-Based Software," in *Proceedings of the Second International Conference on COTS-Based Software Systems*, ICCBSS '03, (London, UK, UK), pp. 251–260, Springer-Verlag, 2003.

[123] S. Gerard, C. Dumoulin, P. Tessier, and B. Selic, "Papyrus: A UML2 Tool for Domain-Specific Language Modeling," in *Model-Based Engineering of Embedded Real-Time Systems*, vol. 6100, pp. 361–368, Springer, 2010.

[124] P. Allen and S. Frost, *Component-based Development for Enterprise Systems: Applying the SELECT Perspective*. New York, NY, USA: Cambridge University Press, 1998.

[125] G. Canfora and M. D. Penta, "Testing services and service-centric systems: challenges and opportunities," *IT Professional*, vol. 8, pp. 10–17, mar 2006.

[126] G. Fortino, R. Gravina, W. Russo, and C. Savaglio, "Modeling and Simulating Internet-of-Things Systems: A Hybrid Agent-Oriented Approach," *Computing in Science Engineering*, vol. 19, no. 5, pp. 68–76, 2017.

[127] W. Shi, J. Cao, Q. Zhang, Y. Li, and L. Xu, "Edge Computing: Vision and Challenges," *Internet of Things*, vol. 3, pp. 637–646, oct 2016.

[128] "Fog Computing and the Internet of Things: Extend the Cloud to Where the Things Are," tech. rep., Cisco, 2015.

[129] A. Jula, E. Sundararajan, and Z. Othman, "Cloud computing service composition: A systematic literature review," in *Expert Systems with Applications*, 2014.

[130] M. Rowe, S. Lane, and C. Phipps, "CareWatch: A Home Monitoring System for Use in Homes of Persons With Cognitive Impairment," *Top Geriatr Rehabil.*, vol. 23, no. 1, pp. 3–8, 2007.

[131] A. Bertolino and A. Polini, "The audition framework for testing Web services interoperability," in *31st EUROMICRO Conference on Software Engineering and Advanced Applications*, pp. 134–142, aug 2005.

[132] C. Smythe, "Initial Investigations into Interoperability Testing of Web Services from their Specification Using the Unified Modelling Language," in *Proc. of International Workshop on Web Services Modeling and Testing (WS-MaTe 2006)*, 2006.

[133] M. Utting, A. Pretschner, and B. Legeard, "A taxonomy of model-based testing approaches," *Software Testing, Verification & Reliability*, vol. 22, 2012.

[134] A. Ahmad, F. Bouquet, E. Fourneret, F. Le Gall, and L. B., "Model-Based Testing as a Service for IoT Platforms," in *Leveraging Applications of Formal Methods, Verification and Validation: Discussion, Dissemination, Applications.*

*ISoLA 2016* (M. T. and S. B., eds.), vol. 9953 of *Lecture Notes in Computer Science*, Springer, 2016.

[135] "FI-WARE: Future Internet Core Platform Project. EU FP7-ICT Project," 2014.

[136] S. Taherizadeh, A. C. Jones, I. Taylor, Z. Zhao, and V. Stankovskia, "Monitoring self-adaptive applications within edge computing frameworks: A state-of-the-art review," *Journal of Systems and Software*, vol. 136, pp. 19–38, 2018.

[137] T. Dillon, C. Wu, and E. Chang, "Cloud Computing: Issues and Challenges," in *Advanced Information Networking and Applications (AINA), 2010 24th IEEE International Conference on*, pp. 27–33, apr 2010.

[138] D. E. Vega, I. Schieferdecker, and G. Din, "Design of a Test Framework for Automated Interoperability Testing of Healthcare Information Systems," in *2010 Second International Conference on eHealth, Telemedicine, and Social Medicine*, pp. 134–140, feb 2010.

[139] M. Spichkova, H. S. Heinz, and I. Peake, "From abstract modelling to remote cyber-physical integration/interoperability testing," vol. 1403, 2014.

[140] "Papyrus Modeling Environment," 2018.

[141] D. Budgen and P. Brereton, "Performing systematic literature reviews in software engineering," *Int. Conf. Soft. Engin.*, p. 1051, 2006.

[142] J. Moreno, "A Testing Framework for Cloud Storage Systems," *ETH Zurich*, no. March, p. 66, 2010.

# VITA

## EDUCATION

**Doctor of Philosophy**, *Computer Science*, Özyeğin University,
İstanbul, 2018 (*expected*)
Thesis: Runtime Verification of IoT Systems Using Complex-Event Processing
Advisor: Asst. Prof. Dr. İsmail Arı, GPA: 3.60

**Master of Science**, *Electrical Engineering*, University of Southern California,
Los Angeles, CA/USA, 2000
Focus: Computer Networks, GPA: 3.53

**Bachelor of Science**, *Electrical & Electronics Engineering*, Çukurova University,
Adana, 1997
Senior Project: Investigation of EEG Signals with Neural Network Techniques
Advisor: Asst. Prof. Dr. Sokol Saliu, GPA.: 79.9 / 100

## EXPERIENCE

***Freelance Consultant and Occupational Trainer***, Adana
05/01/2017 - present
Compiling and conducting occupational training courses on *Software Testing*, *Test Automation*, *Mobile Application Testing*, and *Web Application Testing.*

***Senior Instructor***, Adana Science and Technology University, Adana
01/01/2015 - 04/29/2017
Prepare and conduct lectures to Bachelor of Science students in Computer Engineering, Electrical Engineering and Management Information Systems departments: *Introduction to C Programming*, *Algorithms and Programming*, *Discrete Mathematics and Its Applications*

***Institute Deputy Director***, TÜBİTAK BİLGEM Information Technologies Inst. (BTE), Kocaeli 08/2012 - 05/2014
• Executing research activities, projects and personnel on IED, Avionics, Real-Time Systems and Maritime Defense Systems
• Strategic Business Development Experience with Turkish Undersecretary of Defense (UoD), and Armed Forces Command High Ranking Officers. Secured contracts with DoD and UoD.
• Conducting International R&D Collaboration with Fraunhofer
• Reorganization of the Institute with $\sim$ 330 researchers
• Management and Coordination of $\sim$ 60 research engineers working on concurrent projects under my directorate, majority of whom had M.Sc. and Ph.D. degrees.
• Technology Transfer of Selected Products

***Program Manager (Turkish tx/fx jet)***, TÜBİTAK, Kocaeli & Ankara
04/2013 - 05/2014
• Engaging TÜBİTAK with the UoD, DoD and Private Sector defense industry to partake in the Programme by developing relations
• Building and management of a team of 20 high-caliber engineers/scientist from 7 different research institutes

***Project Manager***, TÜBİTAK BİLGEM BTE, Kocaeli
12/2011 - 02/2013
• Turkeys first indigeneous DO-178B Level-A Certifiable, ARINC-653 and POSIX-1003 compliant Safety-Critical real-time operating system was successfully delievered to the customer ($\sim$ 1000 KLOC, deployed on several mil-spec platforms)
• Exceptional leadership in crisis management and risk mitigation, re-scheduling of tasks and deadlines, re-planning, project management, and securing a 12 months deadline extension by negotiating with all stakeholders, for a project that was otherwise doomed to fail. Restructuring of teams and reassignment of roles and responsibilities over a team of approximately 40 software & firmware engineers
• Extensive use and generating know-how on full SDLC management at CMMI - Level 3 maturity

***Software Team Leader***, TÜBİTAK BİLGEM BTE, Kocaeli
09/2006 - 12/2011
• The first indigenous ARINC-653 and POSIX-1003 compliant, DO-178B Level-A certifiable real-time operating system (RTOS), Extensive programming experience at system level with C, and partly C++
• Product and Software Engineering Process Ownership from Requirements to Delivery for a team of 25 senior software engineers
• Resource planning, team building, role assignment, know-how development over particular subject matters, enabling the team to train themselves on RTOS essentials by engaging them in collaborative research and development teams
• URL: http://bilgem.tubitak.gov.tr/en/content/rtos/gis-real-time-operating-system

***Assistant Project Manager***, TÜBİTAK BİLGEM BTE, Kocaeli
01/2009 - 02/2012 (URL: http://ytkdl.bilgem.tubitak.gov.tr/)
• Founding executive member of Center for Software Testing and Quality Assessment by securing a fund from Department of Development of Turkish Republic.
• Building Business Relations with National Software companies and Public Sector
• Established and trained a team of Software Quality Engineers and Management of Software Quality Process and Team for the Institute
• Established TSE (Turkish NIST) accredited certification laboratories for TS 13298 and TS ISO/IEC 25051

***Head of Software Technologies Department***, TÜBİTAK BİLGEM BTE, Gebze
09/2007 - 08/2012
• Software Process Engineering/Improvement Transition Process Owner (IEEE 12207, CMMI - Level 3)
• Consult the Inst. Director on Software Engineering & Resource Planning Issues when embarking on new projects
• Established coding standards for Java, C++, and software engineering processes that were to be applied by all projects in the Institute. Observed the conformance of projects to the standards through periodical reviews.

***Senior Software Engineer***, TÜBİTAK BİLGEM BTE, Kocaeli
10/2002 - 09/2006
• Design, develop with C++ and verify integration of MXF-484 radio subsystem to attack helicopter avionics system through MIL-STD-1553B bus communication lines.
• Design, develop with C++ and verify modernization of Link-11 tactical data communication subsystem on GENESIS, G-Class frigate modernization project.

***Software Engineer***, PRI/BROOKS Automation, Mountain View, California
02/2001 - 06/2002
Design, develop with Java and verify a distributed real-time shop-floor dispatcher system for LfS (Leverage for Scheduling) software. Achieved a modular GUI system development through use of MVC, Java Swing and AWT. Tested on customer premises (www.tsmc.com)