Journal of Universal Computer Science, vol. 20, no. 8 (2014), 1049-1070
submitted: 4/4/14, accepted: 23/7/14, appeared: 1/8/14 © J.UCS

Dynamic Verification of Mashups of Service-Oriented Things through a Mediation Platform

Antonio Brogi
(Department of Computer Science, University of Pisa, Italy
brogi@di.unipi.it)

Javier Cubo
(Universidad de Málaga, Dept. de Lenguajes y Ciencias de la Computación, Spain
cubo@lcc.uma.es)

Laura González
(Instituto de Computación, Facultad de Ingeniería, Universidad de la República
Montevideo, Uruguay
lauragon@fing.edu.uy)

Ernesto Pimentel
(Universidad de Málaga, Dept. de Lenguajes y Ciencias de la Computación, Spain
ernesto@lcc.uma.es)

Raúl Ruggia
(Instituto de Computación, Facultad de Ingeniería, Universidad de la República
Montevideo, Uruguay
ruggia@fing.edu.uy)

Abstract: The new Internet is evolving into the vision of the Internet of Things, where physical world entities are integrated into virtual world things. Things are expected to become active participants in business, information and social processes. Then, the Internet of Things could benefit from the Web Service architecture like today's Web does; so Future service-oriented Internet things will offer their functionality via service-enabled interfaces. As demonstrated in previous work, there is a need of considering the behaviour of things to develop applications in a more rigorous way. We proposed a lightweight model for representing such behaviour based on the service-oriented paradigm and extending the standard DPWS profile to specify the (partial) order with which things can receive messages. To check whether a mashup of things respects the behaviour, specified at design-time, of composed things, we proposed a static verification. However, at run-time a thing may change its behaviour or receive requests from instances of different mashups. Then, it is required to check and detect dynamically possible invalid invocations provoked by the behaviour's changes. In this work, we extend our static verification with an approach based on mediation techniques and complex event processing to detect and inhibit invalid invocations, checking that things only receive requests compatible with their behaviour. The solution automatically generates the required elements to perform run-time validation of invocations, and it may be extended to validate other issues. Here, we have also dealt with quality of service and temporal restrictions.

Keywords: Run-Time Verification, Composition, Mashup, Behaviour, Service-Oriented Things, Internet of Things, Web of Things, Complex Event Processing, Mediation Patterns
Categories: D.2.10, D.2.11, D.2.12

1 Introduction

The new Internet is evolving into the vision of the Internet of Things (IoT) where physical world entities are integrated into virtual world things. Future Internet has emerged as a new initiative to pave a novel and dynamic global network infrastructure, with self-configuring capabilities, to meet the changing global needs of business and society. Future service-oriented Internet devices will offer their functionality via service-enabled interfaces adopting the vision of the Web of Things (WoT) (inspired from the IoT), e.g., via SOAP-based Web Services or RESTful APIs [Guinard et al. 2012, Pautasso et al. 2008].

The IoT, including the mass of resource-constrained devices, could benefit from the Web Service architecture like today's Web does. Then, the IoT vision will allow the interconnection among heterogeneous devices as services, which is not done currently within the common Internet scenarios. Recent work [De Souza et al. 2008, Jammes et al. 2005] has focused on applying the paradigm of Service-Oriented Architecture (SOA) [Erl 2005], in particular Web Services standards (SOAP, WSDL, etc.), directly on devices. In general, applying SOA to networked systems is a crucial solution to achieve reusability and interoperability of heterogeneous and distributed things. Specifically, implementing Web Service standards on devices presents several advantages in terms of integration by reducing the needs for gateways and translation between the components. This would enable the direct orchestration of services running on devices with high-level enterprise services. Hence, the goal is to provide the functionality of each thing as a Web Service in an interoperable way that can be used by other entities such as enterprise applications or even other devices.

However, adapting a given device to SOA is not a trivial problem. It is required to implement efficiently the things, and many efforts are still needed to handle the composition and interaction of things coming from diverse sources, as well as to standardize and manage their data. Several SOA initiatives, such as OSGi, UPnP, or Jini, have evolved to interconnect heterogeneous devices and services. But not all of them can equally adapt to the others using the same hood. Furthermore, the lack of standardization makes programming for devices an arduous task. Then, it is required a standard way for device manufacturers to expose devices to software developers and consumers, while providing developers with a standardized API.

To address this issue, the new emergent OASIS standard Devices Profile for Web Services [OASIS 2009] has been designed as a set of guidelines based on WS-* specifications to provide interoperability among different devices and services in a networked environment, e.g., a printer, a smartphone, a sensor or other new devices can detect DPWS-enabled devices on a network. Some convincing points in favour of DPWS are that it is an OASIS standard, it employs a Web Service mode being built on the standard W3C Web Service architecture (SOAP+WSDL+XML-Schema), and it is natively integrated into Windows Vista (WSDAPI implementation of DPWS). In DPWS, every device is abstracted as a service where features of the device are exhibited as hosted services. DPWS is lightweight, supports dynamic discovery in local networks, and can be used by orchestration or choreography standards, such as (Web Services) Business Process Execution Language (WSBPEL) or Web Services Choreography Description Language (WS-CDL).

However, the comparison between the important properties of reuse and research challenges of Web Services shows a gap in the use of DPWS in the future focused on reusability [Zinn et al. 2010]. Thus, DPWS shows, for example, those topics like business processes, context dependencies or quality factors have to get more focus in order to increase the reuse of DPWS devices. Hence, some open points of research for the future in this sense have been detected in order for DPWS to be used more easily in the area of software engineering and to become more accepted.

To develop Future Internet service-oriented applications and exploit correctly the composition among things, it is crucial to define rigorous methodologies. These methodologies should not only consider features as signature, eventing mechanisms, security and discovery as it is currently done by DPWS, but also complex real world integration, such as those involving complex business processes.

In our previous work [Cubo et al. 2012], we detected the need to explicitly represent the (implicit) behaviours of things in order to develop applications in a more rigorous way. Specifically, we promoted the usage of WS-* technologies to specify service interfaces of things by extending the standard DPWS with behavioural descriptions. The main purpose of this is to facilitate developers the implementation of DPWS-compliant things (or devices) that host services by considering their behaviour in terms of the (partial) order in which the actions visible at the interface level are performed. We consider this challenge is crucial to control the behaviour of heterogeneous things during their compositions in highly dynamic environments of the Future Internet. These compositions will allow the creation of new applications generated as mashups of things where some concerns have to be handled, such as that the composition may violate the behaviour of the things (provoking lock situations) and some of their features may change at run-time. We proposed a static verification technique to check whether or not a mashup of things respects the behaviour of the composed things specified at design-time. However, at run-time a thing may change its behaviour or receive requests from instances of different mashups. Then, it is required to check and detect dynamically possible invalid invocations provoked by the behaviour's changes. To this end, we extended our static verification with an approach based on mediation techniques and Complex Event Processing (CEP) [Luckham 2007] to detect and inhibit invalid invocations, checking that things only receive requests compatible with their behaviour [González et al. 2013].

Our proposal consists in processing invocations of services hosted in devices through a mediation platform, in order to detect and block the invalid ones using CEP techniques. As main contribution, the solution automatically generates the required elements to perform the run-time validation of invocations, and it may be easily extended to validate other issues. Here, we have also dealt with quality of service (QoS) and temporal restrictions.

This paper is organized as follows. In Section 2, we motivate and list the contributions of our proposal. Section 3 describes our proposal to perform run-time verification of mashups of things. In Section 4, we discuss advantages, deployment alternatives and we compare this solution with our previous one. In Section 5, we compare our approach to related work. Section 6 draws conclusions and future work.

2 Overview of our Proposal

Throughout this section, in Section 2.1, we describe the model we have proposed in our previous work based on extending DPWS to support the behaviour of things. In Section 2.2, we illustrate the need of extending our proposal, describing the current problem statement to be solved, the dynamic verification during things' composition at run-time. In Section 2.3, we introduce CEP, mediation patterns and discovery proxy, which we will use to tackle the run-time verification of the behaviour-aware composition of things. In Section 2.4, we give an overview of our approach to detect and manage invalid invocations at run-time during the composition.

2.1 Behaviour-Aware Model based on Device Profile for Web Services

As is depicted in Figure 1, DPWS uses the primitives of the Web Services Architecture to create a framework for interoperable and standardized communication between embedded devices. In DPWS, every device is abstracted as a service where features of the device are exhibited as Web Services. Some of the Web Services specifications in which DPWS is based are: (i) WSDL for describing the messages each hosted service is capable of sending and receiving, (ii) SOAP for transporting all the messages, (iii) WS-Discovery and SOAP-over-UDP for device discovery, and (iv) WS-Transfer / WS-MetadataExchange for device and service description.

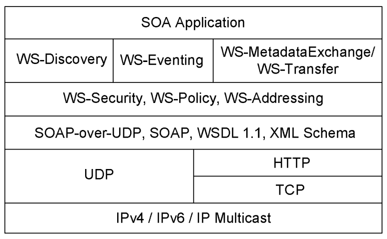

SOA Application
WS-Discovery
WS-Eventing
WS-MetadataExchange/ WS-Transfer
WS-Security, WS-Policy, WS-Addressing
SOAP-over-UDP, SOAP, WSDL 1.1, XML Schema
UDP
HTTP
TCP
IPv4 / IPv6 / IP Multicast

Figure 1: DPWS Protocol Stack

In [Cubo et al. 2012] we motivated the necessity of extending DPWS to facilitate the implementation of a device (or thing) as a full-service considering that its WSDL description should specify not only signature, but also the behaviour with the order in which input and output actions are performed while the networked system interacts with its environment. Input actions model methods that can be called, or the end of receiving messages from communication channels, as well as the return values from such calls. Output actions model method calls, message transmission via communication channels, or exceptions that may occur during methods execution. A formalisation of our model is already presented in this previous work.

Figure 2 depicts the interaction messages among clients and services hosted in devices, according to the DPWS architecture. Messages include discovery, description, control and eventing.

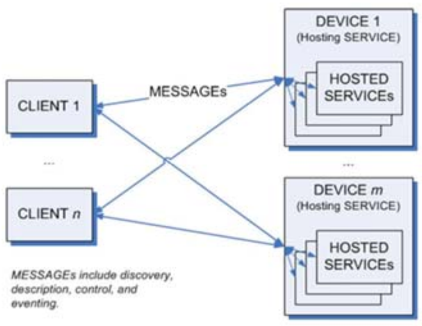


Figure 2: DPWS Messages Interaction among Clients and Services Hosted into Devices

In order to include this extension in the DPWS profile, in our previous proposal we applied rigorous and lightweight methodologies to develop things, by promoting WS-* technologies, to specify interfaces of things, and adding the behaviour of things to the DPWS profile. This extended DPWS specification will help the manufacturers and developers the implementation of DPWS-compliant things (or devices) that host services by taking into account their behaviour (using *constraints* or *finite state machines*) in terms of the order in which actions, visible at the interface level, are performed while things are composed. In fact, we have developed a prototype as a GUI to facilitate the specification of constraints and FSMs in real complex examples.

- ***Constraints***. When only a partial order of the behaviour of things is required, we propose to use three types of behavioural constraints to be added to the guidelines (statements) exposed by DPWS:

  $$\{b_i\}\texttt{afterAll}\{a_i\}, \{b_i\}\texttt{afterSome}\{a_i\}, \texttt{onlyOneOf}\{a_i\}$$

  where $\{b_i\}$ and $\{a_i\}$ are actions of a service hosted in a device. The `afterAll` constraint is used to specify that any action $\{b_i\}$ only can be executed if all the actions $\{a_i\}$ have been previously executed. The `afterSome` constraint is less restrictive than `afterAll`, since any action $\{b_i\}$ can be executed whether some action $\{a_i\}$ have been already executed. The `onlyOneOf` constraint means that only one of the sets of actions $\{a_i\}$ can be executed in an interaction session.
- ***Finite State Machines***. In those cases where it is required to specify not only the partial order, but also the ordered full-sequence among operations with the corresponding states changes according to the messages execution, we propose to use Finite State Machines (FSMs) [Brand et al. 1983] as a simple and user-friendly graphic solution to represent the complex relationships between messages.

2.2 Motivating Example: An Airport Surveillance System

In order to illustrate our previous model (presented in Section 2.1), and the need of our new proposal, we considered in [Cubo et al. 2012] a complex real-world example:

an airport surveillance system composed by heterogeneous devices hosting services; a motion detector, and a surveillance camera – hosting a record control service – located in a specific area in the airport, and a video device – hosting a video streaming service and a media ejection service – located in a control center, and people (using other devices), everything interconnected.

We focus on a scenario where a security guard connects, by means of an application installed within a mobile device to a new motion sensor and a new camera, both installed in a specific area of the airport. The behaviour of the system is the following: once Bob finds the new devices, when a non-expected motion is detected, Bob is notified with the exact position of the movement detected. He logins to access the camera, and after he can perform three actions: (i) move the camera to the desired position, (ii) start to record, and/or (iii) make a zoom. If Bob considers there is an emergency situation, then he sends a command to warn the control center to start the streaming at real-time of the video being recorded by the camera. After this, the control center staff may reproduce and analyze the video, while it is recorded concurrently, and act accordingly. When monitoring and surveillance of the concrete situation is complete, then Bob can finish recording.

In this scenario the handling of the behaviour of the hosted services into the heterogeneous and distributed devices is required, not only to achieve a correct interaction, but also to get appropriate specifications of every behaviour-aware service and application.

- ***Constraints***. The behaviour of the service record control hosted in the device camera, with actions such as *auth*, *move*, *record*, or *halt*, can be specified by means of the following constraints:

  C1: {*move*, *record*} `afterAll` {*auth*}
  C2: {*halt*} `afterSome` {*move*, *record*}

- ***Finite State Machines***. The behaviour of the service video streaming, hosted in the device video, requires a considerable number of exchanged messages (*on*, *play*, *pause*, *stop*, *rewind*, *fast-forward*, and *off*) in a concrete order (as detailed in [Cubo et al. 2012]), so its handling may need a more complicated model as provided by the FSMs. Figure 3 depicts the control of the message full-sequence of this service using FSM representation.

The explicit specification of the behaviour of things by means of constraints or full-sequences with FSMs is the foundation to develop behaviour-aware compositions of things. These compositions will create applications generated in form of mashups with new functionalities to be remotely accessed (e.g., as Software-as-a-Service - SaaS, or Mashups-as-a-Service). But it is required to check whether a composition of things fulfills or violates their behaviour, so we proposed a simple and efficient verification technique at design-time.

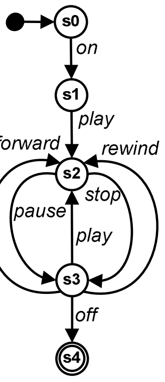


Figure 3: FSM for Complex Behaviour of the Service Video Streaming hosted in Device Vide

Therefore, we defined a checker function in order to perform the static verification, analysing traces and actions executed of the orchestration specified by the user, according to a sets of constraints and/or finite state machines, both determining the behaviour of the things.

However, as aforementioned, a thing may change its behaviour at run-time. Then, a change in the behaviour of a thing may cause that various compositions do not fulfill its behaviour anymore. Although compositions could be redesigned to comply with the new behaviour, it would be appropriate to design run-time verifications techniques to react when this situation occurs. Moreover, given that a thing can receive at run-time requests from instances of different mashups, these requests could violate the behaviour of that thing, even though each mashup fulfills such behaviour, because of the state's change of the thing. This kind of situations cannot be detected at design-time, so run-time mechanisms are required to become aware of it and act accordingly. Therefore, we propose to use complex event processing and meditation patterns to address these new open issues.

2.3 Complex Event Processing and Mediation Patterns

This section presents background information on some technologies that are relevant for this work: (i) complex event processing, (ii) integration and mediation patterns, and (iii) WS-Discovery.

Complex Event Processing (CEP). CEP refers to methods, techniques, and tools for processing events while they occur. It allows deriving relevant higher-level events (i.e. complex events) from a combination of lower-level events, in a timely fashion and permanently [Eckert et al. 2011]. To this end, event queries are continuously monitoring incoming streams of simple events. These queries are used to specify situations as a combination of simple events occurring, or not occurring, over time.

Various products (e.g. Drools Fusion) rely on the production rules approach to implement event queries [Eckert et al. 2011]. In this case, whenever an event occurs a corresponding fact must be created in the so-called working memory and rules specify actions to be executed when certain states are entered. These states are detected through event queries expressed as conditions over these facts.

CEP platforms provide support for various types of event patterns, which allow specifying combinations of events. Some types of supported patterns are logical operator patterns, subset selection patterns, temporal patterns and spatial patterns [Etzion et al. 2010].

Integration and Mediation Patterns. Integration and mediation solutions are usually based on probed and well-known patterns, which have also been documented in the literature [Wylie et al. 2009, Chappell 2004, Hohpe et al. 2003] . This section reviews the relevant patterns for our proposal.

Service virtualization patterns take an existing service and deploy a new virtual service in a mediation platform. These patterns introduce a point of mediation which can be used to validate, route, transform or normalize requests, among others [Wylie et al. 2009].

The VETO pattern [Chappell 2004], which consists in applying a sequence of mediation mechanisms: validate, enrich, transform and operate, is a frequently applied mediation pattern that can be used in conjunction with the previous one. The validate mechanism checks incoming requests and blocks invalid ones, that is, it does not allow that the request reach the target service or application. The enrich mechanism adds additional data to a request. The transform mechanism converts the request to the required target format. The operate mechanism invokes the target service.

Event-driven integration patterns deal with distribution of events in real time and integration with CEP engines. The event extractor pattern monitors interactions across a mediation platform and passes relevant events to a CEP engine. The event reactor pattern extends the previous one by supporting a synchronous interaction with a CEP engine to check if the latest event has triggered a complex event [Wylie et al. 2009].

WS-Discovery. DPWS leverages WS-Discovery for discovering devices [Jeyaraman et al. 2008]. WS-Discovery supports both an ad-hoc discovery and a managed discovery mechanism. Using the ad-hoc mechanism the client sends multicast Probe messages to discover devices on the local network. This mechanism has some limitations: the network range of multicast messages is limited and multicast messages increase network traffic [Jeyaraman et al. 2008]. To deal with these limitations, WS-Discovery also supports a managed discovery mechanism where a specialized component, a discovery proxy, is used. This proxy usually stores the network address of services that are present on the local subnet and on a wider network. In such a way, clients may directly communicate with the discovery proxy to discover devices avoiding the generation of multicast traffic [Jeyaraman et al. 2008].

2.4 Contributions of our Approach

The main idea of our proposal is to process, through a mediation platform, the interactions between clients (e.g. mashups) and devices in order to validate the requests that are sent to hosted services. Our platform handles the invocations of services hosted in devices to detect and block the invalid invocations using CEP techniques. In this way, devices only receive requests which are compatible with their behaviour.

The platform can be placed as a specialized device within the same network where devices are located. Figure 4 presents the high level architecture of the proposal and the interactions that take place between the different components.

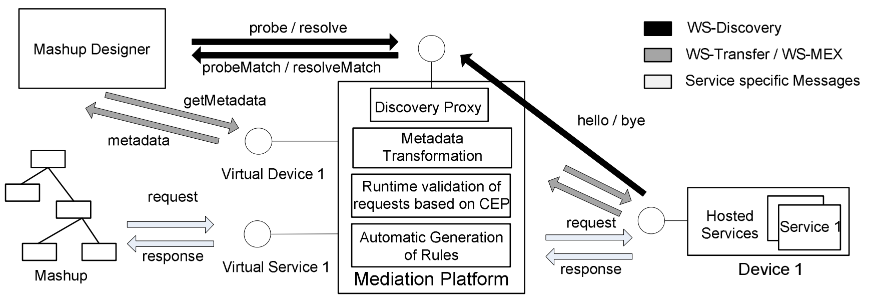


Figure 4: High Level Architecture of the Proposal

Following the Virtual Services mediation pattern, devices and hosted services are exposed through the mediation platform via Virtual Devices and Virtual Services, respectively. These virtual elements are automatically configured when devices advertise themselves in the platform (via WS-Discovery) or when the platform receives the metadata of hosted services (via WS-MetadataExchange).

In order to intercept the interactions that take place during the discovery process (WS-Discovery interactions), the platform includes a Discovery Proxy. This component sends to clients the proper information (i.e. network addresses) so that the subsequent WS-Transfer/WS-MetadataExchange interactions, between clients and devices, are processed through the platform via Virtual Devices. This allows that the metadata obtained from the devices can be modified in a way that clients receive the network addresses of Virtual Services instead of the ones of Hosted Services.

Figure 5 a) presents the mediation flow that is executed when the Discovery Proxy receives WS-Discovery messages: it stores or removes, from a repository, devices' metadata when receiving a "hello" or "bye" message, respectively, and it gets, transforms and returns these metadata when receiving a "probe" or "resolve" message. In this case, the transformation consists on returning the Virtual Devices's network addresses instead of the real addresses of the devices. In addition, Figure 5 b) presents the mediation flow that is executed when a Virtual Device receives WS-Transfer or WS-MetadataExchange messages: it invokes the corresponding device and, as mentioned above, it transforms the response to return the networks addresses of Virtual Services instead of the ones of Hosted Services. This way, service requests are processed by the platform and they can be validated, and even blocked, before reaching the services.

Finally, run-time validation of invocations is performed leveraging CEP: production rules are automatically generated and deployed in the platform, according to the behaviour of the devices, so that the requests can be validated against them.

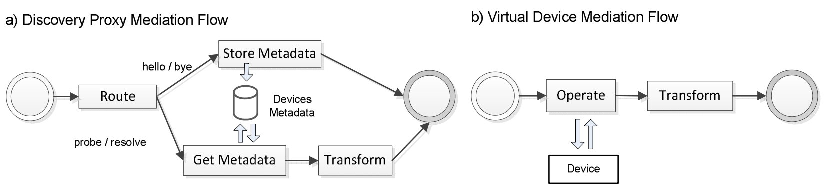


Figure 5: Discovery Proxy and Virtual Devices' Mediation Flow

The main contributions of the platform we propose are the following:

1. Detecting and blocking invalid invocation at run-time.
2. Validating behaviour with rules generated automatically.
3. Dealing with temporal constraints and quality of service.

3 Run-Time Verification of Mashups of Things as Services

In this section, we describe actions of our approach to perform run-time verification of mashups of things as services. First, in Section 3.1, it is detailed how our approach works to detect and block invalid invocations while things are composed. Then, Section 3.2 presents the validation of the behaviour of things by means of rules automatically generated, considering the invalid invocations detected.

3.1 Strategy to Detect Invalid Invocations at Run-Time

To detect and block invalid invocations, the platform uses a combination of mediation and CEP techniques. Figure 6 shows how a service invocation is processed by the platform and the components that allow the runtime verification of the invocations.

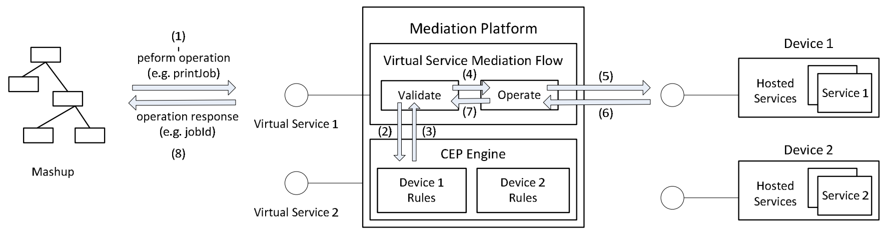


Figure 6: Detecting and Blocking Invalid Invocations

As stated before, hosted services are invoked through virtual services deployed in the platform, tag (1) in Figure 6, following the Virtual Services pattern. Virtual services consist of a mediation flow, which is a simplification of the VETO pattern, and comprises two mediation steps: validate and operate. The validate step synchronously interacts with a CEP Engine (2), following the event reactor pattern, to check if the invoked operation is invalid. If so, a complex event is triggered (3) and the invocation is blocked. Otherwise, the operate step is executed (4) which invokes the target operation in hosted service (5). Lastly, the response is returned to the client (6), (7) and (8). In order to specify when a given invocation is invalid for a device, a set of production rules has to be deployed on the CEP Engine. These rules can be automatically generated based on the specified behaviour of each device, which is obtained by the platform from the metadata of the hosted services (see details in Section 3.2). Also, when the platform receives an invocation a new event is generated and sent to the engine. These events and the deployed rules constitute the basic elements to trigger a complex event when an invalid invocation for a service is received, allowing the platform to detect this situation and act accordingly. For instance, the platform can block the invocation (i.e. prevent that it reaches the target service) and return an error message to the client or try to apply a compensation action (e.g. invoking another service).

3.2 Behavioural Validation through Rules

In order to validate the interactions between clients and devices, the platform includes a set of modules which generates rules automatically based on the specified behaviour of devices, specified using constraints or finite state machines. This is depicted in Figure 7. Also, the platform can be extended with other modules to generate rules based on other information (e.g. regarding QoS).

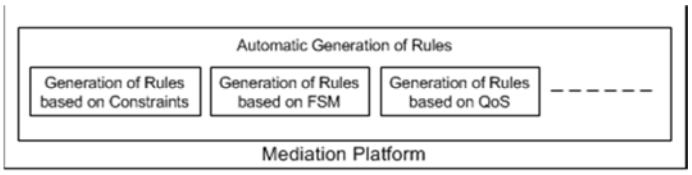
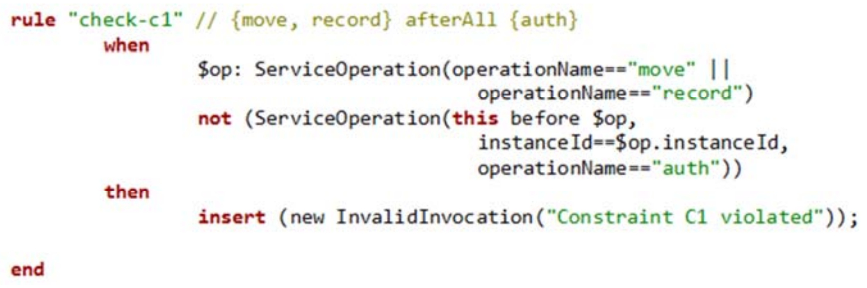

Automatic Generation of Rules
Generation of Rules based on Constraints
Generation of Rules based on FSM
Generation of Rules based on QoS
Mediation Platform

Figure 7: Automatic Generation of Rules within the Platform

This section describes how production rules can be used to detect invalid invocations according to the behaviour of devices and how these rules can be automatically generated based on those behaviours, which can be specified through constraints or FSMs. In particular, the Drools Rule Language [Drools 2013] is used to exemplify our proposal over the running example. However, note that any other language that allows specifying event queries [Eckert et al. 2011], even the ones not based in production rules, can be used for the same purpose. For instance, if the Esper CEP engine is used in the solution, the Event Processing Language is the one to be leveraged in order to detect these invalid invocations.

3.2.1 Behaviour specified with Constraints

When the behaviour of a device is specified with constraints, one or more rules are required to perform the run-time validation. Giving the temporal support that CEP technologies provide, in most of the cases rules can be specified in a very intuitive way. For instance, the rule depicted in Figure 8 detects invalid invocations for the constraint C1 presented in Section 2.2. More precisely, it detects when a *"move"* or *"record"* operation is received and an *"auth"* operation was not received before, in the context of the same mashup's instance.

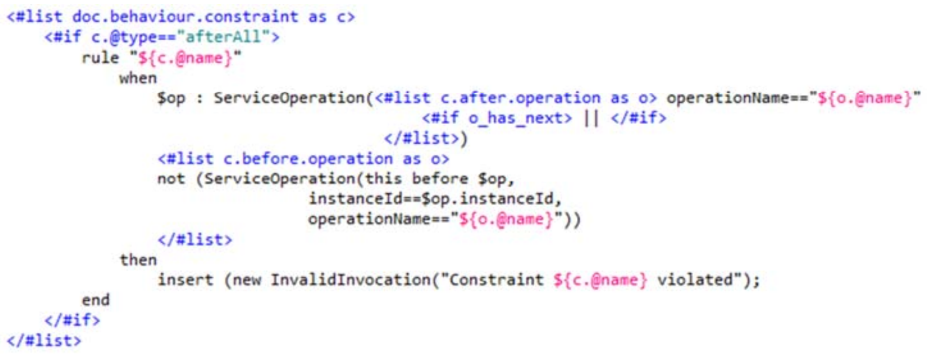


```
rule "check-c1" // {move, record} afterAll {auth}
        when
                $op: ServiceOperation(operationName=="move" ||
                                        operationName=="record")
                not (ServiceOperation(this before $op,
                                        instanceId==$op.instanceId,
                                        operationName=="auth"))
        then
                insert (new InvalidInvocation("Constraint C1 violated"));

end
```

Figure 8: A DRL Rule to Check the Constraint C1

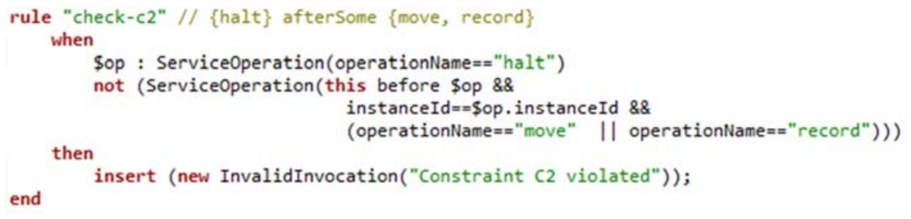
Each `afterAll` constraint leads to a rule with the overall structure of the previous one. Indeed, given a specific constraint, the corresponding rules can be automatically generated. As an example, Figure 9 presents how rules for `afterAll` constraints can be generated with the Freemarker [Freemaker 2013] template engine, assuming as an input the XML representation of `afterAll` constraints.



```
<#list doc.behaviour.constraint as c>
    <#if c.@type=="afterAll">
        rule "${c.@name}"
            when
                $op : ServiceOperation(<#list c.after.operation as o> operationName=="${o.@name}"
                                            <#if o_has_next> || </#if>
                                        </#list>)
                <#list c.before.operation as o>
                not (ServiceOperation(this before $op,
                                instanceId==$op.instanceId,
                                operationName=="${o.@name}"))
                </#list>
            then
                insert (new InvalidInvocation("Constraint ${c.@name} violated");
        end
    </#if>
</#list>
```

Figure 9: Automatic Generation of Rules based on Behaviour

Invalid invocations for the other types of constraints (i.e. `afterSome` and `onlyOneOf`) can also be detected with this type of rules. For example, Figure 10 detects invalid invocations for the constraint C2 presented in Section 2.2. More precisely, it detects when a *"halt"* operation is received and *"move"* or *"record"* operations were not received before, in the context of the same mashup's instance.



```
rule "check-c2" // {halt} afterSome {move, record}
    when
        $op : ServiceOperation(operationName=="halt")
        not (ServiceOperation(this before $op &&
                              instanceId==$op.instanceId &&
                              (operationName=="move"  || operationName=="record")))
    then
        insert (new InvalidInvocation("Constraint C2 violated"));
end
```

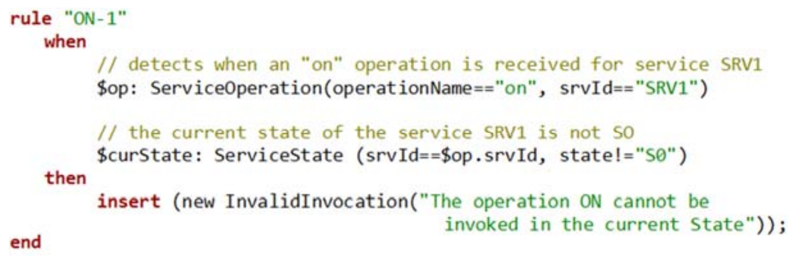
Figure 10: A DRL Rule to Check the Constraint C2

Similarly, these rules can be automatically generated for each specific constraint.

3.2.2 Behaviour specified with Finite State Machines

When the behaviour of a device is specified by means of FSMs, the run-time interactions can also be validated with rules. In this sense, our approach builds rules which handle the state of a device and detect when invalid operations are invoked for the current state.

Concretely, for each operation of a hosted service two rules have to be created. One of them detects when the operation is received in an invalid state and trigger a complex event to inform this situation. The other one detects when the operation is received in a valid state and, if needed, updates the current state of the device. Figure 11 presents the first one of these rules, specified with DRL, for the operation *"on"* of the example presented in Section 2.2.



```
rule "ON-1"
   when
        // detects when an "on" operation is received for service SRV1
        $op: ServiceOperation(operationName=="on", srvId=="SRV1")

        // the current state of the service SRV1 is not SO
        $curState: ServiceState (srvId==$op.srvId, state!="S0")
   then
        insert (new InvalidInvocation("The operation ON cannot be
                                        invoked in the current State"));
end
```

Figure 11: A DRL Rule for the "on" Operation

As it happens with constraints, the overall structure of these rules is the same for any behaviour specified using FSMs. Thus, they can be automatically generated given a specific FSM as presented in the pseudo-code included in Figure 12, which generates the rules that detects when operations are received in an invalid state.

```
for each operation o
   rule "o.getName()"-1
        when
                $op: ServiceOperation(operationName=="o.getName()", ....

                $curState: ServiceState (srvId==$op.srvId,
                                         for each valid state s for o
                                                state!="s.getName()",
                                         next
                                  )
        then
                insert (new InvalidInvocation("...
   end
next operation
```

Figure 12: Automatic Generation of Rules based on Behaviour

3.3 Temporal Constraints and Quality of Service

Given that the proposed approach is based on mediation and CEP techniques, it provides a suitable infrastructure to handle temporal constraints and deal with quality of service (QoS) issues. This section presents how the proposed solution can natively support this kind of requirements.

Quality of Service. Some QoS issues can also be handled by the platform. For instance, if a hosted service specifies the maximum number of invocations it can handle in a specific period of time, a rule can detect when this maximum is reached. Figure 13 shows a DRL rule which detects if the last invocation for a hosted service exceeded the maximum number of invocations it can handle (three) in ten seconds.

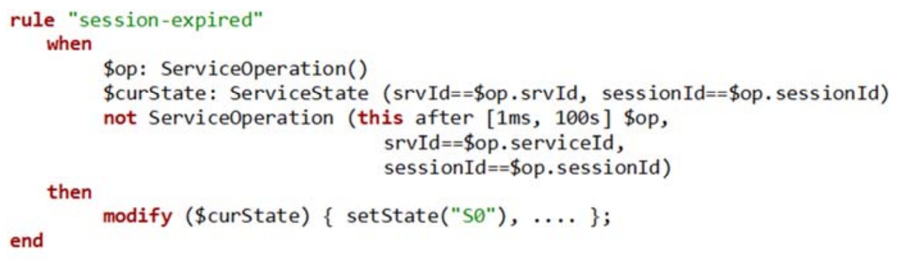
```
rule "check-throughput"
   when
        $op: ServiceOperation(srvId=="SRV1")
        $op: ArrayList(size >  3) from collect (ServiceOperation(srvId == "SRV1")
                                            over window:time(10s)
   then
        insert (new InvalidInvocation("Maximum throughput reached."));

end
```

Figure 13: A DRL Rule to check the Maximum Throughput

When the maximum number of invocations is reached, the platform can block the invocation or differ it, so hosted service receives the invocation when may process it.

Temporal Constraints. Given the built-in support of temporal patterns in CEP solutions, the platform is able to handle temporal constraints in different ways. For instance, if the behaviour of a device, specified by means of FSMs, allows specifying a maximum period of time between operations in the same session, a rule can be created so that if an operation is not received within this period of time the device returns to the initial state. As an example, Figure 14 presents a DRL rule which detects when an operation for a given service was not received in the last 100 seconds, in which case updates the current state of the device to the initial one.



```
rule "session-expired"
   when
        $op: ServiceOperation()
        $curState: ServiceState (srvId==$op.srvId, sessionId==$op.sessionId)
        not ServiceOperation (this after [1ms, 100s] $op,
                              srvId==$op.serviceId,
                              sessionId==$op.sessionId)
   then
        modify ($curState) { setState("S0"), .... };
end
```

Figure 14: A DRL Rule to detect when a Session expired

4 Further Discussion

This section discusses deployment and implementation alternatives for the solution, advantages, disadvantages, limitations, and comparison with the static verification.

Deployment Alternatives. We envision four main deployment alternatives for the platform: intra-organization, gateway, "as a service" and embedded. The intra-organization alternative can be used to validate the interactions between devices and clients located within an organization. In this case, the platform is a specialized hardware/software infrastructure located in the same network where the other devices are. The gateway approach can be used to deliver the functionalities of the devices to other organizations. In this case, the platform is also placed in the same network where devices are and it acts as a gateway which processes and validates requests from other organizations. The "as a service" alternative can be used to deliver the functionalities of the devices to other organizations but delegating the validation of invocations to an external entity (e.g. due to infrastructure limitations). In this case, the platform can be located within a cloud infrastructure leveraging its computational resources. Finally, the embedded alternative consists in using only two components of the platform: the runtime validation component and the automatic generator of rules component. These components have to be placed either on the client side (e.g. as part of a mashup execution environment) or within the device, if it has the required computational resources.

Implementation Alternatives. The implementation of the proposed mediation platform can be based in well-known and probed enterprise software solutions like Enterprise Service Buses and CEP Engines.

First, an Enterprise Service Bus (ESB) product can be used as the core component of the mediation platform. An ESB is a standard-based integration platform that combines messaging, Web Services, data transformation, and intelligent routing to reliably connect and coordinate the interaction of heterogeneous applications [Chappell 2004]. It provides a middle integration layer, with reusable integration and communication logic, through which services and applications communicate by sending messages. These messages can be processed by mediations flows which may apply to them different mediation operations (e.g., transformation, routing).

ESB products natively support most of the mediation patterns and mechanisms (e.g. service virtualization, message transformation, event-driven integration) described in Section 2.3 and which are the base of the proposed solution. Also, there are various types of ESB products in the market which can be leveraged to address specific requirements and to implement the different deployment alternatives we envisioned. For example, an Internet Service Bus [Ferguson 2010] can be used for the “as a service” deployment alternative and hardware-based ESBs [Roshen 2009] can be used when a more efficient message processing is required. This way, ESBs provide a suitable infrastructure over which the implementation of the mediation platform can be based.

Regarding CEP Engines, there is also a wide range of products in the market as well as academic proposals [Eckert et al. 2011]. They can be leveraged to implement the runtime validation component according to the specific requirements of the solution regarding, for example, performance [Bizarro 2007] or resource constraints [Pietrzak et al. 2012]. In this sense, we have prototyped the runtime validation component using the Drools Fusion product. In particular, we have implemented the required DRL rules for the presented example and, by emulating service invocations, we were able to test that these rules indeed detect when a request violates the specified behaviour of a thing.

Advantages. One of the advantages of the solution is the agility to respond to changes in the behaviour of things. That is, rules are automatically generated when a device advertises itself or when a change is detected, so the verification of the new behaviour and blocking of invalid invocations can be done almost immediately. Given that invocations are verified before reaching the devices, another advantage of the proposed solution is that it prevents devices from being saturated with invalid invocations which can cause, for example, an overall degradation in the performance of the device affecting also valid requests. Also, it reduces network traffic arriving to devices which, depending on the specific context, can be an important issue to consider. Furthermore, if an invalid invocation is received, the platform provides a suitable place to perform compensation actions (e.g. retry, differ) in order to maintain the correct execution of mashups. Finally, this solution is well suited for resource-restricted devices which may not be able to execute complex verification operations.

Disadvantages and Limitations. The main disadvantage of the proposed solution is the time overhead in the invocations: they have to pass through the mediation platform where there is always a synchronous interaction with the CEP engine. Thus, although CEP engines are well known by supporting high throughputs per second with low latency [Taher et al. 2011], the proposed solution has an impact in performance. However, it is important to note that devices may choose not to advertise themself to the platform and do it directly to their potential clients. This will lead to direct interactions between mashups and devices, but without run-time verifications. Therefore, the decision to use, or not, the platform for the run-time validation of invocations has to be a trade-off between the performance requirements and the problems that may arise by allowing invocations to reach a device without complying with its behaviour.

The main limitation of the proposed solution is that it detects the invalid invocations when mashups are already being executed, not before. That is, a mashup can be designed and deployed, but only until an invalid combination of operations is executed, the solution detects that there is a problem. Also, the solution still does not provide mechanisms to handled invalid invocations according to different situations.

For example, if a mashup instance is being executed and the behaviour of one of the devices that it uses changes in an incompatible way, the platform validates the invocations according to this new behaviour. This may lead to stopping the execution of the mashup even though it was compliant with the behaviour of the device when its execution started. A further analysis of this kind of situations is required.

Compensation Actions. As stated before, the platform provides a suitable place to perform compensation actions when an invalid invocation is received. In this sense, other common mediation patterns can be leveraged for this task. For instance, if a device is receiving more requests that the ones it can handle, the “differ” mediation pattern [González et al. 2011], which delays an invocation for a given time period, can be used to deal with that. Also, if messages are received in a different order that the one expected, the “resequencer” mediation pattern [Hohpe et al. 2003] can be used to reorder the received messages and send them in the right order to the device. In order to perform these compensation actions, a new component is required in the platform to manage the suitable compensation actions for each type of device and to apply them in case it is required.

Comparison with the Static Verification Approach. The proposed run-time verification solution constitutes a complementary approach to the static verification mechanisms presented in previous work. On one side, the run-time verification mechanisms are required to be able to deal with changes at run-time in the behaviour of the devices once mashups have already been designed and deployed. There are also different situations that cannot be detected with static verification techniques like temporal related issues (e.g. temporal restrictions), QoS aspects and the concurrent execution of mashups. On the other side, the run-time verification solution detects problems as they occur, that is, only until an invalid combination of operations is executed by a mashup. However, the static verification mechanisms allow detecting that a mashup does not comply with the behaviour of devices at design time. In this sense, we believe that both approaches have to be used in conjunction to provide an integral verification solution for the behaviour-aware mashups in the Web of Things paradigm.

Regarding the use of CEP for run-time verification, we also consider re-executing the checker function with all the mashups which use the device whose behaviour has changed, and block the invocations from mashups which do not comply with this new behaviour. Even though a rigorous evaluation has not been performed, it is clear that the required time to perform these checks increases as the number of mashups grows. Hence, this alternative can be time consuming if there is an important number of mashups affecting the agility to respond to a change. Also, this solution only verifies the invocations coming from mashups and does not consider other types of clients which also have to comply with the specified behaviour of devices. However, we understand that this solution can also complement the CEP-based verification, for example, by alerting mashups developers that a mashup does not comply with the behaviour of a device anymore and that it has to be re-designed.

5 Related Work

In this section, we analyze different approaches focused on the run-time verification of things in the context of service-oriented and event-based solutions. Mainly, we compare our proposal to works using CEP and mediation techniques for the WoT.

With the rise of the Future Internet as a reality, there exists the necessity of considering contents, devices, sensors, and things included in the new challenges of the service-oriented computing. Thus, some works have proposed service-oriented solutions for Home Network System [Nakamura et al. 2011] or Smart Home [Parra et al. 2011]. The former presents a sensor mashup platform that allows the dynamic composition of the existing sensor services. They mainly focus on helping non-expert developers to create context-aware services within the home network system, but their framework does not offer a guide to control the behaviour of the system, only messages are exchanged by using WSDL and REST/SOAP. The latter is closer to our approach. Authors propose an application logic distribution where devices in a smart home incorporate a set of rules than can govern their behaviour, following ECA (Event-Condition-Action) rules: they listen to external messages (notifications coming from other services) and, according to some conditions defined in these rules, they decide to perform their own actions. In comparison to our approach, this mechanism is not lightweight and rules may be not enough to determine the correct order among operations of a service. Therefore, we propose not only to define rules from scratch, but also to generate them to check the invocations at run-time, based on the specified behavior of each device.

In [Bertolino et al. 2009] authors working in FET European project CONNECT propose derive from the WSDL of a Web Service a partial ordering relation among the invocations of the different WSDL operations, represented as an automaton modeling the interaction protocol. The behaviour protocol is obtained through synthesis and testing stages (to verify conformance). Compared to our approach, they first assume the behaviour protocol is already specified by means LTSs, while we are proposing to specify service interfaces of things by adding a set of single constraints to the DPWS guidelines to determine the links in the interactions. Their proposal to derive a partial order of the message sequence of a service is complex and does not easily maintain the compromise between expressiveness and scalability in a word composed by billions of resource-constrained devices, since both synthesis and testing processes are required. We apply rigorous and lightweight methodologies to develop things, by helping with the implementation of DPWS-compliant devices.

CEP and mediation techniques are being increasingly used for run-time monitoring, verification and adaptation in service-oriented and event-based solutions. Firstly, in [Inverardi and Tivoli 2013], authors define a method for the automatic synthesis of modular connectors, each of them expressed as the composition of independent mediators. A modular connector, as synthesized by their method, supports connector evolution and performs correct mediation. Instead, our approach uses also CEP to detect and inhibit invalid invocations, checking that things only receive requests compatible with their behaviour. In [Taher et al. 2011] the authors describe an approach to deal with differences between Web Services protocols, by using CEP to adapt their interactions and resolve conflicts. Compared to our approach, message consumption and transmission are modeled as events, and adaptation is specified using automata and deployed as CEP adapters. In [Baouab et al. 2011] an event-based approach to verify the compliance of the overall sequence of inter-organizational choreography operations is presented. Each message received or sent by an organization is associated to an event and CEP is used to verify whether the participating parties have performed their tasks according to the choreography. In [Birukou et al. 2010] the authors propose an integrated solution for run-time compliance governance in SOA, focusing on QoS, security and licensing issues. In a similar way to our proposal, this solution uses CEP to monitor the compliance of business processes during their execution. However, although these proposals leverage CEP and mediation techniques for run-time verification, none of them focus neither in the field of the WoT nor in verifying the compliance of invocations according services' behaviour.

Nevertheless, recently, CEP techniques and mediation solutions have been applied and considered relevant in the field of the WoT in a separate way. On the one hand, as regards CEP applied to the things world, in [Fengjuan et al. 2013] the authors propose a solution to deal with imprecise timestamps and events order in this highly distributed context. Also, in [Wang et al. 2012] a solution to solve the integration of heterogeneous event information resources is proposed. However, none of these solutions uses CEP techniques for the run-time verification of invocations. On the other hand, mediation solutions have been also proposed in the field of the Web of Things. In [De Souza et al. 2008] a middleware infrastructure focused on enabling an efficient collaboration between device-level services and enterprise applications is presented. To this end, the infrastructure includes mediation capabilities to provide connectivity with non-DPWS enabled devices. In turn, DPWS enabled devices can directly interact with enterprise applications or they can be accessed through the infrastructure to get more advanced features (e.g. asynchronous invocations). In [Wu et al. 2012] the authors propose the concept of Gateway as a Service: a Cloud Computing framework for the WoT, focused on integrating devices into service compositions and business processes. Also, in [Hamida et al. 2012] an integrated development and run-time environment for the future Internet is proposed, which include a Light Service Bus to address the access to things considering their resource constraints and leveraging DPWS. All these proposals focus on using mediation capabilities to enable the connectivity to heterogeneous things; but unlike our proposal, they do not provide mechanisms to detect invalid invocations to things according their behaviour. Therefore, to the best of our knowledge there is not any effort in the field of the Web of Things that uses both CEP and mediation techniques jointly to address the run-time verification of the behaviour of things.

6 Conclusion and Further Work

This work has proposed an approach to verify at run-time compositions of service-oriented things specified as services. From our previous efforts, in which we proposed a static verification technique to check whether or not a mashup of things respects the behaviour of the composed things specified at design-time, we have motivated the need of checking, detecting and blocking dynamically possible invalid invocations provoked by the behaviour's changes. This is required because at run-time a thing may change its behaviour or receive requests from instances of different mashups.

Then, along the paper, we have presented a platform to detect and inhibit invalid invocations while things are dynamically composed, by using complex event processing and mediation techniques. Also, our platform takes advantages of the usage of the DPWS discovery proxy. With this proposed platform, we complement our previous static verification mechanism by automatically generating required elements to perform run-time validation of invocations of things, in order to check things only receive requests compatible with their behaviour. This new approach may be easily extended to validate other issues, such as quality of services and temporal restrictions. To illustrate our proposal and the platform in action, we have applied it to a service-based system, an airport surveillance system. Using our platform, production rules have been generated with both behaviour of things or devices specified with constraints (simple behaviours) and FSMs (in case of devices with more complex behaviours). The rules have been deployed in a particular CEP engine.

Currently, we are developing an IDE as Mashup Editor to specify the orchestration corresponding to mashups and a Mashups Execution Environment to deploy the generated mashups; which will be integrated in the architecture of the platform. We have been starting to perform the evaluation of potential performance issues in scenarios with a big number of things, and we hope to publish these results.

As future work, we plan to study the inclusion of concrete recovery strategies in the event an invalid invocation occurs.

Acknowledgements

This work has been partially supported by projects TIN2012-35669, funded by Spanish Ministry MINECO and FEDER; P11-TIC-7659 funded by Andalusian Government; FP7-610531 SeaClouds funded by European Union; and Universidad de Málaga, Campus de Excelencia Internacional Andalucía Tech.

References

[Baouab et al. 2011] Baouab, A., et al.: An Event-Driven Approach for Runtime Verification of Inter-Organizational Choreographies. In Proc. of SCC’11, pages 640–647. IEEE CS, 2011.

[Bertolino et al. 2009] A. Bertolino, P. Inverardi, P. Pelliccione, and M. Tivoli. Automatic Synthesis of Behavior Protocols for Composable Web-Services. In Proc. of ESEC/FSE’09, pages 141–150. ACM, 2009.

[Birukou et al. 2010] Birukou, A., et al. An Integrated Solution for Runtime Compliance Governance in SOA. Service-Oriented Computing, 122–136. Springer, 2010.

[Bizarro 2007] Bizarro, P. BiCEP-Benchmarking Complex Event Processing Systems. Event Processing, 2007.

[Brand et al. 1983] Brand, D., et al.: On Communicating Finite-State Machines. J.ACM, 30(2):323–342, 1983.

[Chappell 2004] Chappell, D. Enterprise Service Bus: Theory in Practice. O’Reilly, 2004.

[Cubo et al. 2012] Cubo, J., Brogi, A., Pimentel, E.: Behaviour-Aware Compositions of Things. In Proc. of iThings'12 in conjunction with GreenCom'12, pages 1-8. IEEE CS, 2012.

[De Souza et al. 2008] De Souza, L.M.S., et al.: Socrades: A Web Service Based Shop Floor Integration Infrastructure. In Proc. of IoT’08, vol. 4952 of LNCS, pages 50–67. Springer, 2008.

[Drools 2013] Drools Rule Language, version 6.0, JBOSS, 2013. Available at: http://www.jboss.org/drools/

[Eckert et al. 2011] Eckert, M., et al.: A CEP Babelfish: Languages for Complex Event Processing and Querying Surveyed.Studies Comp Intellig,vol.347,pages 47-70. Springer, 2011.

[Erl 2005] Erl, T. Service-Oriented Arch.: Concepts, Technology, and Design.Prent.Hall, 2005.

[Etzion et al. 2010] Etzion, O., et al. Event Processing in Action. Manning Publications, 2010.

[Fengjuan et al. 2013] Fengjuan, W., et al. The Research on Complex Event Processing Method of Internet of Things. In Proc. ICMTMA’13, pages 1219–1222, 2013.

[Ferguson 2010] Ferguson, D. The internet service bus. On the Move to Meaningful Internet Systems 2007: CoopIS, DOA, ODBASE, GADA, and IS, 5–5, 2010.

[Freemaker 2013] Freemaker project, version 2.3.20, 2013. Available at: http://freemarker.org

[González et al. 2013] González, L., Cubo, J., Brogi, A. Pimentel, E., Ruggia, R. Run-Time Verification of Behaviour-Aware Mashups in the Internet of Things. In Advances in Service-Oriented and Cloud Computing, pages 318–330. Springer Berlin Heidelberg, 2013.

[González et al. 2011]. González, L., and Ruggia, R. Addressing QoS Issues in Service based Systems through an Adaptive ESB Infrastructure, pages 1–7. ACM Press, 2011.

[Guinard et al. 2012] Guinard, D., Ion, I., Mayer, S.: In Search of an Internet of Things Service Architecture: REST or WS-*? A Developers’ Perspective. In Proc. of MobiQuitous’11, volume 104 of LNICST, pages 326–337. Springer, 2012.

[Hamida et al. 2012] Hamida, A.B., et al.: An Integrated Development and Runtime Environment for the Future Internet. The Future Internet. pp. 81–92. Springer, 2012.

[Hohpe et al. 2003] Hohpe, G., Woolf, B. *Enterprise Integration Patterns: Designing, Building, and Deploying Messaging Solutions*. Addison-Wesley Professional, 2003.

[Jammes et al. 2005] F. Jammes and H. Smit. Service-Oriented Paradigms in Industrial Automation. IEEE Trans. Ind. Informatics, 1(1):62–70, 2005.

[Jeyaraman et al. 2008] Jeyaraman, R., et al. Understanding Devices Profile for Web Services, WS-Discovery, and SOAP-over-UDP. Microsoft, 2008.

[Luckham 2007] Luckham, D.: The Power of Events: An Introduction to Complex Event Processing in Distributed Enterprise Systems. Addison-Wesley, 2002.

[Nakamura et al. 2011] Nakamura, M., et al.: Application Framework for Efficient Development of Sensor as a Service for Home Network System. In Proc. of SCC’11, pages 576–583. IEEE CS, 2011.

[OASIS 2009] Devices Profile for Web Services, version 1.1, OASIS, 2009. Available at: http://docs.oasis-open.org/ws-dd/ns/dpws/2009/01

[Inverardi and Tivoli 2013] Inverardi, P., and Tivoli, M. Automatic Synthesis of Modular Connectors via Composition of Protocol Mediation Patterns. In Proc. of ICSE 2013, pages 1-12. IEEE Press, 2013.

[Parra et al. 2011] Parra, J. et al.: Flexible Smart Home Architecture using Device Profile for Web Services: A Peer-to-Peer Approach. International Journal of Smart Home, 3:39–55, 2009.

[Pautasso et al. 2008] Pautasso, C., et al.: RESTful Web Services vs. ”big” Web Services: Making the Right Architectural Decision. In Proc. of WWW’08, pages 805–814. ACM, 2008.

[Pietrzak et al. 2012] Pietrzak, P., et al. Towards a lightweight CEP engine for embedded systems. In Proc. of IECON 2012, pages 5805–5810, 2012.

[Roshen 2009] Roshen, W. SOA-Based Enterprise Integration: A Step-By-Step Guide to Services-Based Application Integration. McGraw-Hill Professional, 2009.

[Taher et al. 2011] Taher, Y., et al.: Adaptation of Web Service Interactions using Complex Event Processing patterns.In Proc.of ICSOC’11,vol.7084 LNCS,pages 601–609.Springer, 2011.

[Wang et al. 2012] Wang, W., Guo, D.: Towards Unified Heterogeneous Event Processing for the Internet of Things. In Proc. of IOT’12, pages 84–91, 2012.

[Wu et al. 2012] Wu, Z., et al.: Gateway as a Service: A Cloud Computing Framework for Web of Things. In Proc. of ICT’12, pages 1–6, 2012.

[Wylie et al. 2009] Wylie, H., Lambros, P.: Enterprise Connectivity Patterns: Implementing integration solutions with IBM’s Enterprise Service Bus products. Accessible at IBM website.

[Zinn et al. 2010] Zinn, M., et al.: Device Services as Reusable Units of Modelling in a Service-Oriented Environment-An Analysis Case Study.In Proc. of ISIE’10, pages 1728–1735.IEEE CS, 2010.