Modeling RESTful Web of Things Services

Concepts and Tools

Christian Prehofer[1,a,*] and Ilias Gerostathopoulos[*]
[*] Fakultät für Informatik, Software & Systems Engineering Research Group, Technische Universität München, Germany
[a] Corresponding: prehofer@fortiss.org

Abstract

In this chapter we consider tools and methodologies for the development of Web of Things applications based on REST design principles. We discuss tools and methods for creating Web of Things services, in particular mashup tools as well as model-driven engineering approaches. While mashup tools mainly focus on modeling the data flow and rapid development, model-driven engineering approaches permit different views and more expressive modeling concepts. We analyze both concepts and techniques regarding expressiveness, suitability for the problem domain as well as ease of use and scalability. Then, we discuss how mashup tools can be extended based on model-driven engineering concepts, while preserving the advantages of simplicity and ease of use. In particular, we show how mashup tools can be extended to more flexible, generic operations on sets of things, based on advanced modeling concepts.

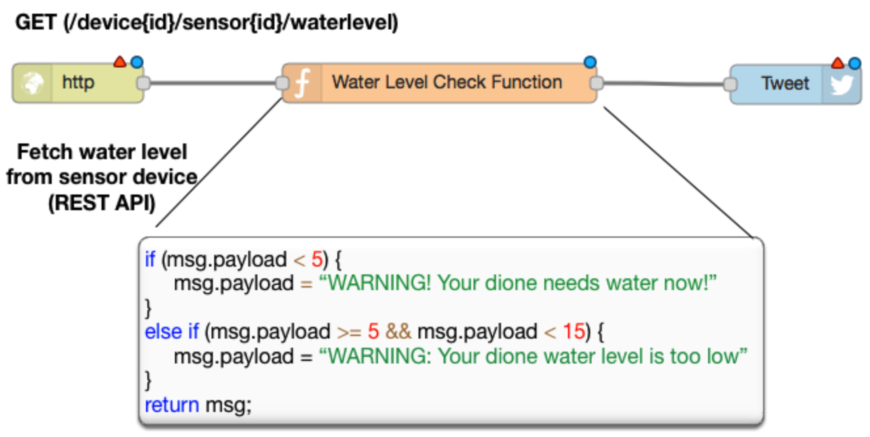
Chapter points

- Mashup tools are widely available for creating Web of Things services in simple and graphical way.
- Mashup tools can be compared to more expressive but also more complex model-driven approaches, as both aim for high-level modeling of services.
- Mashup tools can be extended based on model-driven approaches, while preserving the advantages of simple service creation. This is shown for generic operations on sets of things.

1. Introduction

The basic idea of the Internet of Things (IoT) vision is the pervasive presence of a variety of things or objects — such as Radio-Frequency Identification (RFID) tags, sensors, actuators, mobile phones, etc. — which are able to interact with each other and cooperate to reach common goals [18, 1]. There is a growing interest in research on new technologies and novel applications for the IoT, as well as in related areas

[1]fortiss, GmbH, Germany, prehofer@fortiss.org


GET (/device{id}/sensor{id}/waterlevel)
http
Water Level Check Function
Tweet
Fetch water level from sensor device (REST API)

```
if (msg.payload < 5) {
    msg.payload = "WARNING! Your dione needs water now!"
}
else if (msg.payload >= 5 && msg.payload < 15) {
    msg.payload = "WARNING: Your dione water level is too low"
}
return msg;
```

Figure 0.1 Model of a plant monitoring application in Node-RED.

such as wireless sensor networks and ubiquitous and pervasive computing [46]. Web of Things (WoT) builds on this momentum and aims at building an application layer for the creation of IoT services. It builds on existing Web protocols, in order to allow devices from multiple vendors to interoperate seamlessly.

In this chapter, we focus on methods and tools for the development of WoT systems. As WoT systems grow in functionality, size and management complexity, we believe that a systematic software-engineering approach towards building WoT systems becomes highly relevant. We argue that such an approach needs to combine lightweight data-flow-based IoT modeling tools, such as mashup tools, with more sophisticated model-driven engineering (MDE) methods and tools, such as UML-based design tools. Following the WoT vision, it has to be built around well-known web architectural styles that allow for resource discovery and API interoperability, such as the REST style. Finally, the approach has to promote code reuse and cater for both static settings and dynamic ones, where WoT devices operate in an opportunistic fashion.

This chapter is based on earlier work towards such an integrated WoT development approach (published in [44, 47, 45]). We first provide an overview of *mashups*, which is a popular approach for application development in the WoT. There are several tools implementing this concept, e.g. IBM Node-RED [26], glue.things [28], WotKit [4], as well as Clickscript [20]. Such *mashup tools* allow for visual, interactive modeling of the data flow between IoT devices and Web services and are well-suited for rapid prototyping of WoT systems. As an example, Figure 0.1 depicts the model of a simple plant monitoring application in Node-RED.

MDE methods and tools have also been proposed for the WoT/IoT, with ThingML [16] being a prominent example among other proposals [49, 42, 43]. Although they need more upfront effort in creating the involved models and setting up the underlying infrastructure, MDE tools allow for more expressive modeling than mashup tools, since they provide multiple views and diagrammatic notations (not just a single data flow view). Following a classical MDE approach, models created in MDE tools can be used to automatically generate code via a series of model-to-model and model-to-text transformations. These models are also typically amenable to sophisticated verification methods, both manual (e.g. model-based testing) and automatic (e.g. model checking).

MDE methods and tools can be used in the WoT in the modeling of (i) RESTful interfaces, and (ii) behavior of actuators. In case of large WoT systems with multiple sensors and actuators, the RESTful interfaces of sensors and actuators become lengthy and complex. A systematic model-driven approach to modeling and generating rich RESTful interfaces for the discovery, reading and manipulation of resources for the WoT is thus becoming important. At the same time, the behavior of actuators can be naturally modeled by state machines; this is common practice in embedded systems development.

Once WoT sensors and actuators are provided via RESTful interfaces, mashup tools can be used to model the business logic of a WoT system. In this respect, a limitation of mashup tools is that they require that all connections in the visual data flows are statically defined. At the same time, similar connections cannot be grouped together; there is no way e.g. to reuse the part of the business logic that switches on a light in one room to repeat the same action in another room. Instead, each such operation has to be modeled explicitly in a mashup tool.

To address this limitation, we propose the concept of *generic mashup operations*. In particular, we propose the extension of mashup modeling with $1 : n$ relations, which models a set of resources in a concise way. Having this in place, we can use RESTful operations to sense and actuate a whole set of resources at the same time. This provides the necessary flexibility to work with dynamic scenarios, where new resources are added to a set and are automatically considered in collective sensing and actuating. At the same time, it lifts the necessity to model identical operations explicitly, thus leads to more natural and easy-to-use programming abstractions.

The rest of the chapter is structured as follows. Section 2 provides the basic background and definitions of REST architectural style and WoT mashups. Section 3 provides an overview mashup tools and details on three predominant tools. Section 4 provides a more general account on MDE methods and tools for WoT, while Section 5 compares them to the mashup tools. Section 6 explains how to model WoT applications comprised of RESTful services. Section 7 extends the RESTful WoT development by proposing generic mashup operations. Finally, Section 8 summarizes the important points of our integrated approach for WoT development.

2. Background

2.1. RESTful design

The REST architectural style is aligned with the concepts used in the HTTP protocol; the work by Roy Fielding has shaped the concepts of RESTful design [15]. Following [37], the main ingredients of RESTful design are as follows:

- Identification of resources via Uniform Resource Identifiers (URI). These are hierarchically structured and each resource must have at least one URI.
- Uniform interfaces to read and manipulate the resources. These are the four basic HTTP operations GET, POST, PUT and DELETE. Other operations, e.g. HEAD and OPTIONS, deal with metadata.
- Self-descriptive messages. Representation of the resources can be accessed in different formats, e.g. HTML, JSON or XML. The messages, both requests and reply, contain the complete context and are self-descriptive in this sense.
- Stateless interactions, i.e. the server does not maintain session state on the interactions with the clients. This means that all information to fulfill a request is included in the HTTP request, i.e. the resource name and message.

2.2. Mashups and Mashup tools

A mashup is a composite application that integrates two or more existing components available on the web. These components can either be data, application logic, or user interfaces. The individual components are called "mashup component"; the gluing mechanism is called "mashup logic". The mashup logic is the internal logic which defines how a mashup operates or how the mashup components have been orchestrated [38]. It specifies which components are selected, the control flow, the data flow and data mediation as well as data transformation between different components [12].

Composition of a mashup extensively deals with the kind of components that make it up. The application stack has been broadly classified into data, logic, and presentation (user interface) layer. The mashup created accordingly is called either a data, logic, or user interface mashup.

Mashup components are the building blocks of a mashup. In practice, several technologies and standards are used in the development of mashup components. Simple Object Access Protocol (SOAP) web services [50], RESTful web services, Javascript APIs, Really Simple Syndication (RSS) [29], Comma-Separated Values (CSV) [57] etc. are some of the prominent ones. Depending on their functionality the mashup components have been broadly classified into three categories (Figure 0.2):

1. Data components provide access to data. They can be static like RSS feeds or dynamic like web services which can be queried with inputs.

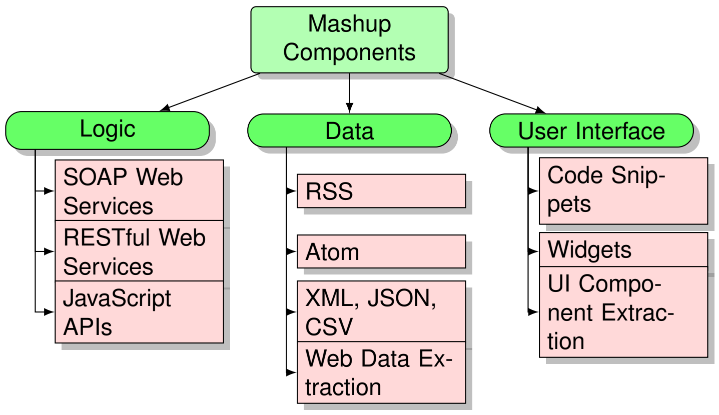


Figure 0.2 Classification of Mashup Components, following [12]

2. Logic components provide access to functionality in the form of reusable algorithms to achieve specific functions.
3. User interface components provide standard component technologies for easy reuse and integration of user interfaces pieces fetched from third-party Web applications with in the existing user interface of the mashup application.

Mashup tools have been proposed as a simple way to develop mashups. This was supported by uniform communication protocols and APIs based on REST principles. Early mashup tools are Microsoft Popfly and Yahoo Pipes; for an overview we refer to [24]. In recent years, there has been a lot of interest in applying the same ideas to the IoT/WoT, also building on REST interfaces [5, 39, 19].

According to [30], mashup tools typically include data mediation. This involves converting, transforming, and combining the data elements from one or multiple services to meet the needs of the operations of another.

For connecting services, there are different concepts as discussed in [59]. The main, predominant one is modelling data flow. For others, mainly in the enterprise area, also centralized approaches with processing rules are considered. For communication, asynchronous messages are used, e.g. using REST-style communication. In general, orchestration can be described by data flow and/or workflow, or through a publish-subscribe model [59].

IoT/WoT mashup tools typically provide a graphical editor for the composition of services for one application. This models the message flow between the components. Components can be sensor nodes, processing or aggregation entities as well as external web-based services. Thus, mashup tools can also be seen as specific cases of end-user programming [58] but are however limited to the specific model of describing message flow. In addition, some mashup tools provide simulation tools and also interoperability for messaging between different platforms.

3. State of the Art in Mashup Tools

In this section, we detail on the most prominent mashup tools available in the market.

3.1. Node-RED

Node-RED[1] is an open-source mashup tool developed by IBM and released under Apache 2 license [26]. It is based on the server side JavaScript platform framework Node.js[2] (that is why the "Node" in its name). It uses an event-driven, non-blocking I/O model suited to data-intensive, real-time applications that run across distributed devices.

Node-RED provides a GUI where users drag-and-drop blocks that represent components of a larger system which can either be devices, software platforms or web services that are to be connected. These blocks are called nodes. A node is a visual representation of a block of JavaScript code designed to carry out a specific task. Additional blocks (nodes) can be placed in between these components to represent software functions that manipulate and transform the data during its passage [22]. Two nodes can be wired together by connecting the output port of a node to the input port of the other node. After connecting many such nodes, the finished visual diagram is called a flow. An example of a flow is depicted in Figure 0.1.

IoT solutions often need to wire different hardware devices, APIs, online web services in interesting ways. The amount of boilerplate code that the developer has to write to wire such different systems, e.g. to access the temperature data from a sensor connected to a device's serial port or to manage authentications using OAuth [11], is typically large. In contrast, to use a serial port using Node-RED, all a developer has to do is to drag on a node and specify the serial port details. Hence, with Node-RED the time and effort spent on writing boilerplate code is greatly reduced, and the developer can focus on the business parts of the application.

Node-RED flows are represented in JSON and can be serialized, in order to e.g. be imported anew to Node-RED or shared online. There is a new concept of "sub-flows" that is being introduced into the world of Node-RED. Sub-flows allow creating composite nodes encompassing complex logic represented by internal data flows.

Since in Node-RED nodes are blocks of JavaScript code, it is—technically—possible to wrap any kind of functionality and encapsulate that as a node in the platform. Indeed, new nodes for interacting with new hardware, software and web services are constantly being added, making Node-RED a very rich and easily extensible system.

[1]http://nodered.org/
[2]https://nodejs.org/

Lastly, the learning curve to develop a new node for the platform is low for Node.js developers since a node is simply an encapsulation of Node.js code.

To make a device or a service compatible with Node-RED, a native Node.js library capable to talk to the particular device or service is required. However, with the growing acceptance of REST style in Web and IoT systems, more and more devices and services provide RESTful APIs that can be readily used from Node-RED.

3.2. glue.things

The objective of glue.things[3] is to build a hub for rapid development of IoT applications [28]. It heavily employs open source technologies for easy device integration, service composition and deployment [28]. TVs, phones, and various other home/business tools can be hooked up to this platform through a wide range of protocols like Message Queue Telemetry Transport (MQTT) [54], Constrained Application Protocol (CoAP) [54] or REST APIs over HTTP.

The development of mashup applications in glue.things goes through the following three main stages [28]:

Firstly, the devices are connected to the platform to make them web accessible using protocols like MQTT, CoAP or HTTP/TCP etc. Device registration and management is handled by the extensible "Smart Object Manager" layer in the glue.things architecture (Figure 0.3). REST APIs provide communication capabilities and JSON data model is used for propagating device updates. These facilities are leveraged using the client libraries or, for a more intuitive experience of device addition, the web-based dashboard can be used. The dashboard also features several templates for connecting devices and simplifying the tasks for the developer.

The second stage deals with creation of mashups. The glue.things system uses a version of Node-RED that has been enhanced to support multi-users, sessions and automatic detection and listing of new registered devices. External web services like Twitter, Foursquare etc. can also be used during mashup composition. The "Smart Object Composer" layer in the glue.things architecture houses the mashup tool. This layer also has a virtualized device container for managing the registered devices.

Lastly, the created mashups are deployed as Node-RED applications including various triggers, actions and authorization settings. These deployed mashup applications are accessible by REST APIs to the developers who may want to use them in their own custom web applications. To the normal end users, they can be browsed through a collection of mashup applications which can be used after suitable alterations to the connection settings and other environment-specific values. Sharing and trading of these mashup applications is also supported by the platform. This functionality is

[3]`http://www.gluethings.com/`

reflected in the "Smart Object Marketplace" layer in the architecture.

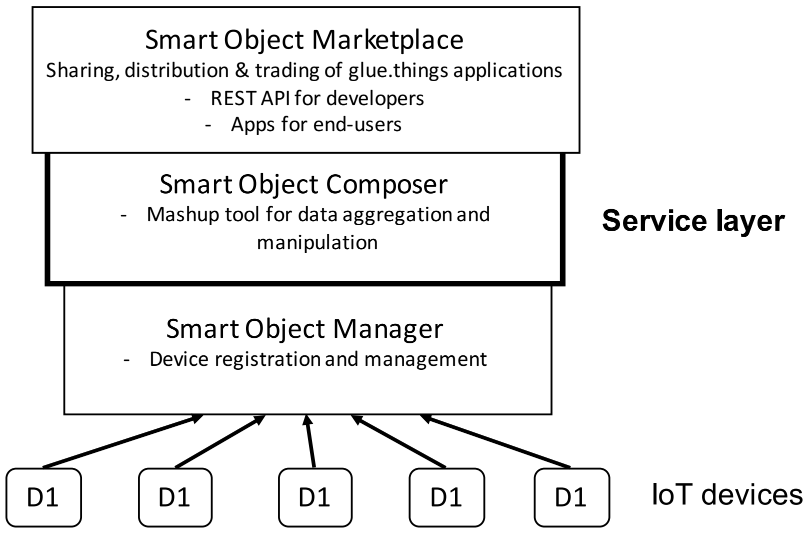

Smart Object Marketplace
Sharing, distribution & trading of glue.things applications
- REST API for developers
- Apps for end-users
Smart Object Composer
- Mashup tool for data aggregation and manipulation
Service layer
Smart Object Manager
- Device registration and management
D1
D1
D1
D1
D1
IoT devices

Figure 0.3 glue.things Architecture [28].

3.3. WoTKit

WoT aims to leverage web protocols and technologies to facilitate rapid construction of web applications exploiting real world objects [4]. WoTKit[4] is a lightweight mashup toolkit and platform that provides a simple way for end-users to find, control, visualize and share data from a variety of things [6]. WoTKit aims for:

1. Easy integration of physical devices, virtual devices and the toolkit.
2. Easy visualization of data collected from different devices.
3. Smart and efficient information processing capability for converting low level data collected from devices to high level sensible data to be used in mashups.
4. Ability to quickly combine different data streams and apply various transformations, triggers i.e. easy service composition or mashup creation.
5. Easy sharing of created mashups and accessibility of features via APIs.

For quick visualization of data collected from different devices, WoTKit uses a JavaScript-based dashboard, which supports the creation of user-defined widgets. Every widget holds some specific set of data collected from devices and an associated visualization. The system comes with visualization plugins like Flot[5]; more visualiza-

[4]`http://hub.urbanopus.net/wotkit/`
[5]**Flot :** Attractive JavaScript plotting for jQuery. `http://www.flotcharts.org`

tion plugins can be hooked up into the dashboard at run-time.

WoTKit also contains an event-based data processing subsystem that processes the low-level data collected from devices and converts them into more sensible high-level data before they are fed into the system. It also features a visual programming environment(mashup tool) for mashing up different data sets. This is similar to the data flow model adopted by Yahoo Pipes. The mashup created using this environment is basically a pipe which consists of connected modules to generate new data from the input data sets. A pipe created is analogous to a flow created in Node-RED.

The toolkit supports end-user scripting to create new custom modules using Python and sharing of created pipes and devices registered in the system. It provides a RESTful API for interacting with the registered devices, thereby facilitating easy creation and integration of applications.

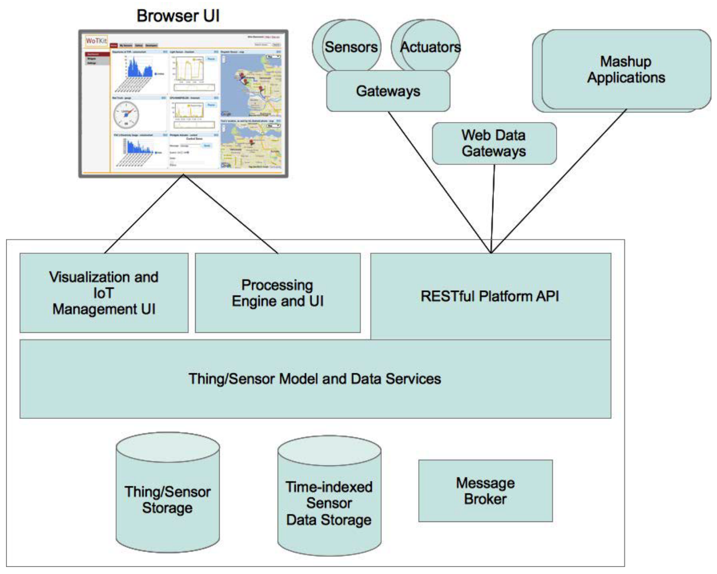


Figure 0.4 WoTkit Architecture [6]

The high level architecture of WoTKit is depicted in Figure 0.4. WoTKit is essentially a Java based web application developed with the Spring Framework. The "UI" part provides the dashboard to interact with the system components graphically while the "RESTful Platform API" provides access to the created mashup applications and registered devices in the system (which obtain unique APIs). The "Thing/Sensor Storage" is the repository containing all the registered devices while the data fetched from devices and pushed into the system are stored in the "Time-indexed Sensor Data Storage". The data model consists of sensors and sensor data having a unique timestamp attached to it. The "Message Broker" is used to deliver data between different components and has been implemented with the Apache ActiveMQ message broker [6].

3.4. Other Prominent IoT/WoT Tools

Paraimpu[7] is a web-based platform which allows to add, use, share and interconnect real HTTP-enabled smart objects and "virtual" things like services on the Web and social networks [40]. User can easily create IoT applications to facilitate their devices to react to environmental changes and activities [28]. To have a unifying view on different devices, these devices are segregated based on their functionality. "Sensors" are devices/services capable of producing data in an acceptable format while "Actuators" are entities that can consume data and in the process of consumption generate some actions. Sensors and actuators communicate using the HTTP protocol and therefore it is easy to create hybrid mashups.

ThingWorx[8] platform aims to build and run applications for the IoT landscape using a so-called model-driven approach [13]. It composes services, applications and sensors as data sources and interconnects these through a virtual bus. The framework supports a wide range of connection protocols for devices like CoAP, MQTT, REST/HTTP and Web Sockets. It can integrate with other cloud providers such as Xively and web services such as Twitter, Facebook or various weather services as data sources. Once data sources are connected to dashboards, they can be used for data gathering and monitoring and can be mashed up to create mashup applications. The data can also be subjected to analytics.

3.5. Features and Limitations

The mashup tools and platforms for IoT landscape have been described in Sections 3.1-3.4 from a very high level with their key features. One of the common objective of these mashup tools is to reduce the development time of applications for the IoT landscape. All these mashup tools are cloud based i.e. they provide the hosting platforms and application API for interacting between devices from applications running in the cloud. This is especially good for business platforms where a centralized application's presence is highly sought [13]. For example in a large scale factory, installing temperature sensors, gathering and analyzing data from them manually is tedious. But if there is a centralized IoT platform offering device registration and management services then implementation and maintenance of an IoT scenario becomes relatively easy. The administrator need not remember the physical address of all the innumerable temperature sensors scattered throughout the factory, instead just login to the central-

[6]**The Apache ActiveMQ Message Broker :** `http://activemq.apache.org/`
[7]`https://www.paraimpu.com/`
[8]`https://www.thingworx.com/`

ized platform to look how the devices are functioning, select some devices to check their data and even name the devices for easy reference and remembrance.

Although the mashup tools vary in degree to which they strive to ease the development process but nevertheless the underlying concepts they adopt is the same. Almost all tools, e.g. WoTKit or Node-RED rely on the concepts of data flow for developing an IoT application. Different data streams from different devices are connected in a logical way and data transformation is applied during the transit of the data. ThingWorx advertises to heavily rely on model-based software development approach for creating IoT applications but nevertheless we believe that the underlying concepts used and features offered by the platform largely correspondent to other existing platforms.

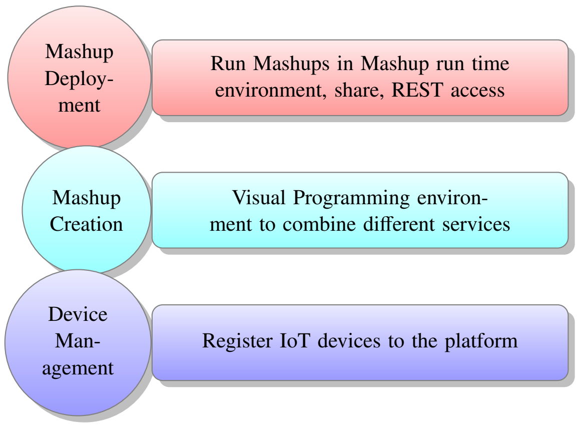

Mashup Deployment
Run Mashups in Mashup run time environment, share, REST access
Mashup Creation
Visual Programming environment to combine different services
Device Management
Register IoT devices to the platform

Figure 0.5 Conceptualization of Features Available in Mashup Tools

After careful observation of many exiting tool-kits, it is appropriate to say that they use different terminologies to denote similar concepts. Mashups are known by different names in different tool-kits but in essence they reflect the same conceptual approach. For example in Node-RED a mashup is called as a flow while in WoTKit it is called a process. The created mashups are generally deployed in a mashup run-time environment. Here the name of the run-time environment differs. For example it is called "Smart Object Marketplace" in glue.things while "RESTful Platform API module" in WoTKit. The commonality among these mashup run-time environments is that all the mashup applications deployed can be shared online and accessed by REST APIs.

Figure 0.5 summarizes the essential features provided by these tool-kits under the banner of different terminologies. Difference arises in the features provided by these tool-kits in these three distinct layers of service. For example in "Device Management", the protocols supported by a toolkit with which we can connect and register

IoT devices vary. Almost all the tool-kits support common protocols like MQTT and CoAP. But glue.things also has support for extra protocols like PubNub (Real time publish/subscribe messaging API for web and mobile apps), Meshblu (Machine to machine instant messaging network and API) etc. Similarly, in "Mashup Creation", Node RED permits the user to embed JavaScript codes while WoTKit has support for Python scripting. In "Mashup Deployment" almost all tool-kits provide the same features which include sharing of created applications and accessing them by REST APIs.

It is interesting to mention a difference between IBM Node-RED and other tools described in this Section: Node RED is just a visual programming environment and not a complete platform by itself. For instance it does not provide a device management layer, so we cannot explicitly register IoT devices to it but it supports a wide range of connection protocols enabling it to communicate to different devices. This limitation is eliminated in glue.things which is a platform in itself. It provides support for device registration and management and uses an improved version of Node RED as its mashup tool i.e Node RED is embedded with in this tool to provide a complete IoT platform functionality.

4. Model-Driven Engineering for WoT

There is a broad range of model-driven engineering (MDE) approaches, especially including domain specific modeling languages. Here, we mainly assume general-purpose modeling languages like UML, even though many more specific approaches exist. For instance, there are several proposals for MDE approaches for developing IoT/WoT applications, e.g. the ThingML language [16].

The motivation for model-driven engineering is to describe a system on a higher level of abstraction. This can be done in UML and other languages by diagrams modeling specific aspects or views of a system. Typical in our setting are architecture models and state machines.

Architecture models may describe the logical role of classes by class diagrams, or the logical role of components by component diagrams. Furthermore, deployment diagrams are used to show the mapping of (software) components to physical entities (hardware).

Behavior is typically described by examples in sequence diagrams, or by state machines and activity diagrams. Activity diagrams describe the data and event flow, similar to models used in mashup tools. State diagrams are used in many embedded domains to model the behavior of specific objects. Also, state diagrams can be analyzed and verified formally (see e.g. [43]) and code can be generated automatically. In this way, it is also possible to generate code for different platforms, even though this still requires to consider the different platform APIs.

MDE development includes the following steps:

1. Model and design the application in device-independent model, here state (transition) diagrams and architecture models (not shown).
2. Code generation and compilation to device-specific, native code.

An advantage of MDE tools is that there is considerable work on semantics, which means that common understanding of diagrams is formally defined. While there are challenges in semantics for the full-featured UML [9], there exist subsets of UML which are semantically well understood [17].

On the down side, the indirection layer created by separating logical and deployment models, together with the upfront effort to set up model-to-model transformations and code generation scripts, renders MDE tools a heavyweight solution compared to mashup tools (for a detailed comparison see Section 5). .

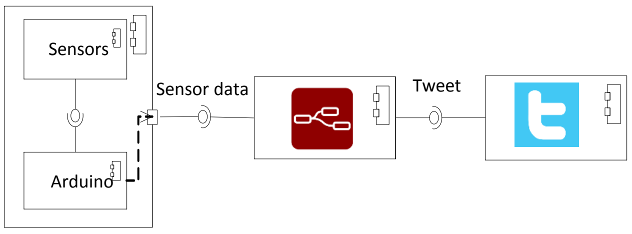


Figure 0.6 Component diagram.

Consider again the Node-RED example of Figure 0.1. In UML, we can model the logical structure of such a scenario with the component diagram of Figure 0.6. Here, each involved component—physical or logical—is represented by a box. Boxes are connected through channels with usual required and provided semantics. In our case, sensors and the Arduino board—gathered in the same component—produce temperature data, that are used by Node-RED for internal computation. In turn, Node-RED produces tweets, used as principal input by the Twitter component.

While the component diagram models the overall logical structure of our system, the activity diagram (Figure 0.7) captures the data flow. In our scenario, the data coming from sensors is collected by the Arduino controller and is then sent to Node-RED. Node-RED in turn will produce a corresponding tweet published on Twitter, based on the received value.

In these diagrams, we can already note an important aspect: mashup models (e.g. the Node-RED model in Figure 0.1) essentially correspond to activity diagrams in MDE terms, since they capture the data flow between components.

Finally, the discrete behavior of each diagram's component is usually defined in MDE through state machines. For example, the behavior of the Node-RED com-

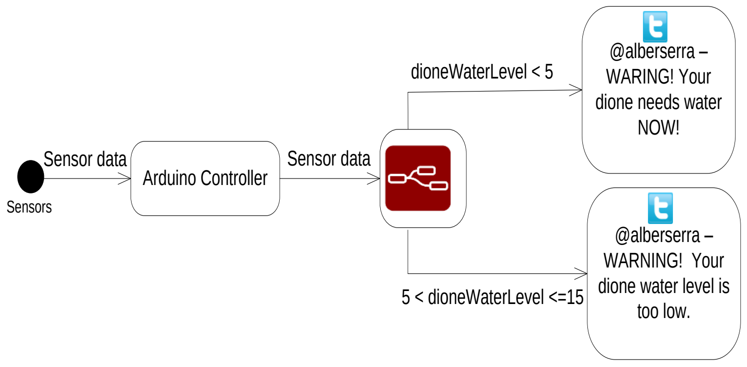

Sensors
Sensor data
Arduino Controller
Sensor data
dioneWaterLevel < 5
@alberserra – WARING! Your dione needs water NOW!
5 < dioneWaterLevel <=15
@alberserra – WARNING! Your dione water level is too low.

Figure 0.7 Activity diagram.

ponent (center component in Figure 0.6) could be defined as in Figure 0.8. In Figure 0.8, WL(X) is an event that, when detected—i.e. we have new input data from sensors—entails the instantiation of X with the current water level and enables the two transitions—labeled with guard/action—to be, eventually, triggered.

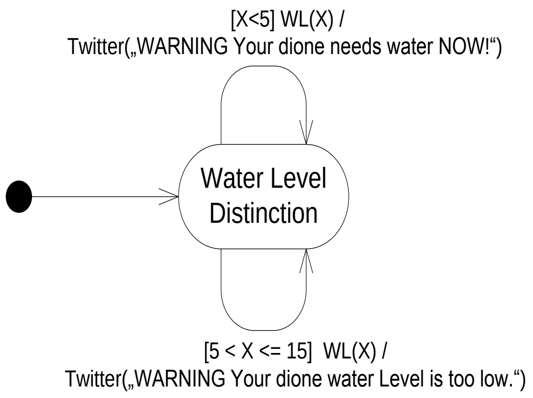

[X<5] WL(X) /
Twitter(„WARNING Your dione needs water NOW!")
Water Level Distinction
[5 < X <= 15] WL(X) /
Twitter(„WARNING Your dione water Level is too low.")

Figure 0.8 State machine for the Node-RED component.

5. Comparing Mashup and Model-Driven Engineering Approaches

When comparing MDE and mashup tools for modeling WoT systems, we need to realize that MDE approaches have a much wider set of modeling techniques and a much more detailed separation of different views and concerns. Thus, we take mashup concepts as the basis and discuss how this fits in the MDE world.

5.1. Execution and Modeling

As mentioned in Section 4, MDE approaches distinguish between (logical) objects and components and between the deployments of components. Components have well defined interfaces and ports, thus matching the notion of components found in mashup tools. The main motivation in MDE is to describe the system from different perspectives. For some models, in particular state machines, execution is possible. On the other hand, consistency between modeling perspectives can be an issue [53].

In contrast, mashup tools essentially define the data/message flow between components. As all data flows are described in one diagram, this also describes the system architecture by showing the connected components in the work flow. Thus, mashup tools integrate a component model with a deployment model. This is the first main difference to MDE approaches. Separating deployment from logical components in MDE creates a layer of indirection and makes prototyping less immediate, compared to mashup tools. In these, it is very easy to develop concrete systems. On the other hand, in a realistic development, applications often need to be mapped to different target deployments. For instance, in ThingML, so-called “configurations” map a logical model to a specific deployment.

Secondly, since mashup tools define a data/message flow between components, they resemble UML activity diagrams, where events are also exchanged. Activity diagrams have a semantics based on events, which drive the control flow. According to [36], there are synchronous and asynchronous semantics for control flow in activity diagrams. In almost all mashup tools, the exchanged messages are asynchronous. However, most mashup tools do not describe semantics formally.

A third difference is that components in mashup tools are either black-box entities or need to be programmed in a general-purpose programming language, e.g. Javascript, C, or Java. In contrast, modeling tools provide rich concepts to model the behavior of components. A widely used approach for developing embedded systems is to use state machines, which are also found in ThingML. From these, it is possible to generate native code for different platforms, hence avoiding execution environments for Javascript or similar languages.

5.2. Expressiveness and Reflecting the Problem-Domain

When using abstractions, as done in MDE and mashup tools, a main question is what can be expressed and whether this reflects the modeled domain in a natural way. This is essential to ensure user acceptance and long-term success of a tool.

Regarding our problem domain, mashup tools model the data flow between sensors, actuators and services. The behavior of the services however needs to be specified in some other programming language, often Javascript. In MDE approaches for IoT/WoT [16], state machines and statecharts [21] are often used for modeling the behavior of individual components as they naturally represent (logical) states of sensors and actuators. WoT modeling and development would clearly benefit by the integration of data flow modeling at the level of the system with state-based modeling at the level of individual components. This goes to the direction of *multi-paradigm modeling* [2], a concept already in use in the embedded domain in tools such as SCADE [3].

We should note that general-purpose modeling languages can be complex and, at the same time, are not ideally tailored to some specific domain. This is the reason for the considerable research on domain-specific modeling languages (DSML) [53, 32], which focus on capturing the important aspects of a particular domain in the domain's idiom and abstraction level. Following this, we can see the modeling language used in a mashup tool as a DSML, since it provides the constructs needed to express the important elements of the mashups domain, i.e. the connections between device APIs and services and the data flow in the system. The syntax of such a graphical DSML provides the rules on how nodes can be connected with each other; the semantics provides the meaning of having the nodes and their connections in the data flow. For instance, in Node-RED, connecting the right side of a function node to the left side of a Twitter node is allowed and means that the output of the particular function will be tweeted. Viewed as DSMLs, mashups tools share both their benefits, i.e. expressiveness and involvement of domain experts, and limitations, mainly related to the need to learn and maintain yet another language.

Another question is the separation of platform-dependent from platform-independent code and other artifacts. MDE has taken considerable effort to separate platform specifics, e.g. deployment or low-level APIs from higher-level models. The main motivation is to produce application models that can be reused in generating code for different deployment environments (thus reducing costs by speeding up development). In some mashup tools, such separation is already in place, as there are separate mappings of logical nodes to physical ones (e.g. WoTKit [4]). On the other hand, some approaches exclusively focus on specific target platforms, e.g. Arduino. Note that this gives a very natural view on the target platform, but not as such on the problem domain and makes the code non-portable.

Another aspect is the modeling of concurrent behavior. In MDE approaches, we can model concurrent behavior in different ways, e.g. by different, parallel areas in an activity diagram or by parallel state machines. As an example, consider a controller for two lights: In this case, we can easily use two parallel state machines, one for each light. For mashup tools, concurrency is not explicitly specified; it is only assumed that asynchronous events are processed in proper order.

5.3. Tool Support and Ease of Use

An important factor for both kinds of approaches is the tool support. While mashup approaches essentially have been tool-based from the very beginning, MDE approaches have a waste body of standards [36] and formal background and often tools implement only specific versions or subsets. Furthermore, mashup tools typically provide a rich integration with existing services (sensors, web services), as well as deployment platforms, from the very beginning.

This is a major difference to MDE approaches, which typically start from the models as such. There are many commercial tools for specific domains, especially for embedded systems. There is however considerable less tool support for the WoT/IoT domain. Regarding this, the work in [53, 56] states that *“current MDE technologies are often demonstrated using well-known scenarios that consider the MDE infrastructure to be already in place. If developers need to develop their own infrastructure because existing tools are insufficient, they will encounter a number of challenges”*.

As a result, the complex MDE infrastructure that allows for model manipulation (merging, transformations), graphical editing of models and code generation is more flexible but also requires more effort to both set up and operate w.r.t. mashup tools.

5.4. Scalability and Runtime Adaptation

We consider scalability in terms of size of models. Generally, visual tools have issues when the views become very big. Then, abstraction or hierarchy concepts are needed. This is an issue for both approaches, as discussed in [53]. For modeling, some concepts for abstractions exist (e.g. hierarchical state machines).

Orthogonal to the scalability, adaptation should also be considered, i.e. the need for the models (e.g. data flow or behavior model) to be adapted at run-time. For instance, a new sensor or service may have to be added. Note that both kinds of tools thrive on presenting a static view of the system, i.e. a view that is invariant during execution. If systems are very dynamic, both MDE and mashup tools cannot properly represent the system anymore. In MDE, there are several approaches addressing this issue, e.g. models at run-time [33, 7] as well as more generic and flexible modeling concepts [34].

5.5. Summary

Table 0.1 summarizes this section by providing a condensed view of the similarities and differences between mashups and MDE approaches for WoT.

6. Modeling of RESTful Services

MDE methods and tools should be combined with mashup tools and concepts for an integrated development of WoT systems. In this section and in Section 7 we consider

	Mashups	MDE approaches
Execution	Mostly asynchronous messages; no formal execution semantics	Synchronous & asynchronous messages; formal semantics
Modeling	Data/message flow diagrams	Architecture, deployment, state & activity diagrams
Expressiveness	Data/message flow; externalized specification of service behavior	Specification of both logical/-physical structure and service behavior
Reflecting the Problem-Domain	Intuitive modeling of data flow between WoT sensors, actuators and services	Domain-specific modeling languages for WoT (e.g. ThingML)
Tool Support	Dedicated mashup tools for WoT (e.g. NodeRED, WoTKit)	Sophisticated tool chains with considerable configuration effort, often only subsets of standards supported in tools
Ease of Use	Low effort of setting up mashup tools, intuitive graphical editing available	High effort of setting up and working with tool chains
Scalability	Issues in representing large models	Issues in representing large models; some concepts for abstraction (e.g. hierarchical state machines)
Runtime Adaptation	No tools to capture dynamic views	Some approaches exist e.g. models at runtime

Table 0.1 Summary of comparison between mashups and MDE approaches in WoT.

such a combination on the basis of RESTful design of WoT systems.

For Internet applications the paradigm of RESTful interfaces is now widely used [48], because it provides a consistent, scalable and flexible model for a large variety of interfaces. We aim to generate RESTful interfaces for WoT systems with sensors and actuators in a systematic and automatic way. This is motivated by the following two observations.

First, interfaces for larger WoT systems with multiple sensors and actuators, including discovery, reading and actuation are lengthy and complex, even using RESTful concepts (see e.g. [23]). Secondly, the WoT also includes the control of actuators. In the MDE world, it is common to model control algorithms by state machines; this is claimed to bring considerable gains in productivity [10].

For the case of complex networks of resources, modeling concepts have been proposed to describe the relations between sensors and actuators at a more high level, see e.g. [41, 51]. We use common UML-based concepts to model WoT systems, and then to generate RESTful APIs from these. In this way, we can describe systems at a higher level of abstraction.

While there has been some effort to use higher-level models for RESTful APIs in web applications, e.g. [60, 41, 51], these do not address the needs of IoT/WoT. Here, we present UML models for typical patterns of WoT systems and show how to generate REST interfaces. In particular, we use composite relations in *class diagrams* to describe physical relations of WoT devices, e.g. which sensor is in which room. This is for instance used in the recent ZigBee Smart Energy standard [23], which provides sophisticated REST interfaces for the discovery, access and recording of sensors and actuators. Secondly, we use *state machines* to model the behavior of actuators. From these models, we show how to generate rich RESTful interfaces for the discovery, reading and manipulation of resources for the WoT.

In the following subsection, we provide background on REST architectural style. Then, we introduce modeling concepts for WoT systems and show how REST interfaces can be generated from these.

6.1. Restful Design and Interfaces

In this section, we complement the description of the main ingredients of REST, presented in Section 2.1, with a high-level model of REST as in [27]. This is defined via a tuple $RS = (R, I, B, \eta, C, D,)$, where R is a set of resources. I is a set of resource identifiers. $B \subseteq I$ is a finite set of root identifiers. $\eta : I \rightarrow R$ is a naming function, mapping identifiers to resources, it is a partial function which is not defined for all elements. C is a set of client identifiers and D is a set of data values, with an equivalence relation $\sim\subseteq (D \times D)$. Note that REST permits several resource identifiers to point to the same actual resource, thus we separate resources and resource identifiers.

Resources identifiers are modeled as URIs, represented as a set I, in the usual form.

```
URI = scheme ":" authority/path ["?"query]
```

Note that the authority part is optional, and also a possible fragment of the form `[ "#" fragment ]` can be added.

For the resource representation of a resource, we write (ids, d), where ids is a list of linked resources and d is a data value. For simplicity, we write just ids or d if one of these parts if empty or missing. This abstracts from typical resource representations in HTML or XML form.

We associate a partial function $deref : I \mapsto 2^I D$ with the state of the server; $deref(i)$, if defined, is the current representation of the resource $\eta(i)$ (which must be defined if $deref(i)$ is defined).

A REST communication is of the form

$$op(i, args)/rc(rvals),$$

where op is a REST operation and rc denotes the return code with return values. We use a simplified form of return codes, using OK and POSTED for successful execution, and ERROR the non successful case. A communication sequence is a sequence of communications carried out between a set of clients and the server.

We thus have the following operations as in [27]:

- $GET(i)/OK(deref(i))$: The method returns the current entity (resource representation) of the resource identified by i from the server.
- $DELETE(i)/OK$: The method dissociates the resource identifier i on the server, resulting in $deref(i)$ being undefined.
- $PUT(i, (uris, d))/OK$: The method associates a resource identified by i, if it is not already associated, and assigns a value to its corresponding entity so that $deref(i) = (uris, d)$. If this is a new association, then $S(i) = \{\}$.
- $POST(i, (uris, d))/CREATED(j)$: The method associates a fresh resource, which is identified by j, and sets $S(j) = \{\}$ and $deref(j) = (uris, d)$. The resource identified by j becomes a subordinate of the resource identified by i, and j is added to $S(i)$.
- $POST(i, d)/OK()$: This provides a data item d to a data-handling process [14]. This is the second type of POST. It does not create a new (subordinate) resource. We assume here it does not affect the linked resources. However, the state of some resources may change, as discussed later.

The last item in the above list is a very common usage of POST, originally intended for posting the content of HTML forms in HTTP requests. This is by many regarded as a different kind of usage of POST and also not included in [27]. The main author of the RESTful APIs is discussing this in a blog post, stating that a separate operation would be more suitable[9] and it is called "overloaded POST" in [48].

6.2. Modeling Restful Design and Interfaces

In this section, we show how to describe WoT systems with simple models and to generate REST interfaces. This will include hierarchically structured sensors and actuators, but also associated entities like sensor readings. Clearly, REST has the ambition to provide homogeneous and clean interfaces. We claim that we can give a more con-

[9]`http://groups.yahoo.com/neo/groups/rest-discuss/conversations/topics/4732`

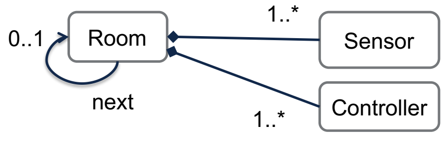


Figure 0.9 Class model for the WoT example.

cise and simpler way to describe sensors and actuators, and then show how to generate RESTful interfaces from these models.

6.2.1. Resource Models for WoT

As a first step we discuss how to derive a resource model for WoT devices and the corresponding discovery of resources. Consider the example shown in Figure 0.9. The description of RESTful resources by a UML model is widely used, e.g. the OData standard for RESTful Web APIs is using this [35]. Here, we focus on the IoT/WoT specifics to model the physical relations of sensors and actuators.

The diagram shows a room with several sensors and controllers, e.g. for some smart home which can be controlled based on sensor readings. The association with the room class is a composition, which expresses that the sensors and controllers are part of this room.

For the generation of a resource model, we need a root identifier, here assumed to be $/$. As the class *Room* is the top element of composition relation, elements of this class will be resources under the root identifier. The other classes will then be immediate sub-resources based on the composition relation in the diagram. The other relation, *next*, between instances of the class *Room* is not a composition and is just modeled as an associated link, but not as a sub-resource. We need to assume that the composition relation in the diagram constitutes a proper tree, such that we can build a resource tree.

In a concrete example with several instances of the class diagram of Figure 0.9, the URIs may be as depicted in Figure 0.10.

This example assumes two rooms with associated sensors and controllers. For these URIs, we can derive the following REST operations to discover the resources in an incremental way as expected for REST. Recall that RESTful design includes the incremental, layered discovery of resources along the resource links. For instance, we have the following operations (assuming the notation $[e1, e1, \ldots]$ for lists.)

- $GET(/)/OK([/room1/, /room2/])$:
- $GET(/room1)/OK([/room1/sensor1,\ /room1/sensor2, /room1/controller1])$

Other relations between classes, here the next link, can be done by some link model in an object diagram. As an example, $GET(/room1/next)/OK(/room2)$ returns the

```
/
/room1
/room1/sensor1
/room1/sensor2
/room1/controller1
/room1/next
/room2
/room2/sensor1
/room2/next
```

Figure 0.10 Example URIs.

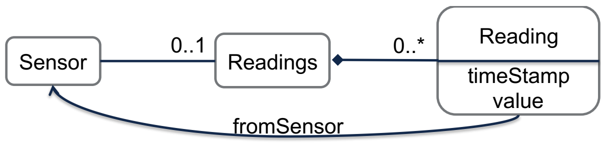


Figure 0.11 Class model for WoT sensor readings.

next link of *room*1, which is assumed to be *room*2.

There is already existing work on modeling resources as a UML class model, e.g. [41, 51]. The main difference here is that we use the hierarchical structure inherent in our IoT/WoT application domain. Secondly, we specify not just the generic relation on the class level, but also discuss how to specify the object level relationships.

6.2.2. Modeling Sensors and Sensor Readings

Based on the resource structure of Figure 0.9, we can now define the interfaces for sensors. Clearly, the most obvious case is reading the sensor value, which is done by a simple *GET* operation. For example, reading the room temperature from *sensor*1 may result in:

$GET(/room1/sensor1)/OK(21C)$

As discussed in Section 6.1, we abstract from the actual data value, which should be in a self-descriptive XML or HTML format.

The next typical step for sensors is to create new resources for specific readings, typically with a time stamp to record a specific reading and to make it available for others. This is modeled in the UML class diagram in Figure 0.11, which shows a relation between sensors and *Reading* resources which are linked via $readings$ and $fromSensor$. Note that this is not a composition, as the readings may be located in some other physical location. An example where this technique is used is the ZigBee Smart Energy standard [23].

Based on this, we get the following RESTful operations to post and to discover the readings, as well as to trace the readings back to the sensor.

- $GET(/room1/sensor1)/\ OK((room1/sensor1/readings, 21C))$
- $POST(/room1/sensor1/readings, (21C, timeStamp))$ $/CREATED(/room1/sensor1/readings/reading1)$
- $GET(/room1/sensor1/readings)$ $/OK([/room1/sensor1/readings/reading1, /room1/sensor1/readings/reading2])$
- $GET(/room1/sensor1/readings/reading1/fromSensor)\ /OK(/room1/sensor1/)$
- $GET(/room1/sensor1/readings/reading1/value)/\ OK(21C)$

Now, the GET operation on the sensor also returns a link to the readings. Note that we have a choice here how to model the (sensor) readings resources. We model these as sub-resources of the sensors in this example.

The second example adds a new reading. This will be inserted after the most recent other reading. The third get operation yields the list of all readings. Then, the get operation retrieves the associated sensor URI. Finally, the last get operation retrieves the value of the reading.

Next we can also delete readings in the expected way.

- $DELETE(/room1/sensor1/readings/reading1/)/\ OK()$:
- $DELETE(/room1/sensor1/readings/)/OK()$:

Note that second delete deletes all readings for this sensor. This is the usual semantics of DELETE for a composite URL.

6.2.3. Modeling Actuators

The basic case of actuators is to model an actuator by a resource [8]. In this case, we identify an actuator with a resource and use get and put to read and put values, respectively.

For more complex actuators, we show in the following how sets of actuators can be modeled by state machines. It is in fact a very common to model control algorithms by state machines in other areas like embedded systems, see e.g. [10]. While such state machines are often used for real-time control systems, we focus on control APIs which can be used locally or over a network with generic APIs.

We show a simple example of two lights in Figure 0.12. Since the two actuators are independent, their behavior is modeled in parallel state machines. The lights themselves are modeled as sub-resources of a room.

When the state machines are translated to RESTful APIs, it is important to discuss what is modeled as resources and what operations are used for triggering state transitions. We assume here that the main objective of state machines is to abstract from the internals of a sensor and to provide an interface to trigger the transitions. These will cause internal state changes. First, it is generally accepted to use POST for triggering state transitions, see e.g. [41]. This falls into the second case of POST for

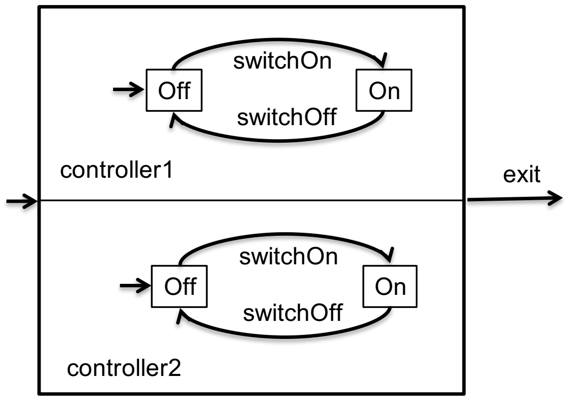


Figure 0.12 Actuator model for the WoT example.

invoking some processing on the server side, and does not create sub-resources on the server. Note that PUT is not suitable for implementing transitions as transitions are not idempotent.

Following this line, we choose to model transitions as subordinate resources, and state as a property to the resource. GET can be used to retrieve the possible transitions and the current state. Note that this violates the concept of hiding internals, but we have included this for completeness. It appears now that PUT can be used to write the state, yet this must be discouraged as the transitions are used to initiate actions and also may have preconditions.

The following shows a few examples, assuming the two controllers of Figure 0.12 are in $room1$.

- $GET(/room1/)/$ $OK([/room1/controller1, /room1/controller2])$
- $GET(/room1/controller1)/\ OK(([switchOff, switchOn], state = Off))$
- $POST(/room1/controller1/switchOff/)/Error()$
- $POST(/room1/controller1/switchOn/)/OK()$

6.2.4. Modeling Services

Based on the defined REST APIs, we can build services. While there is ample experience how to use RESTful services, we show here that our modeling concepts can also be used for services. As an example, consider a switch to handle all lights in one composite resource, as shown in Figure 0.13. The state machine in this example abstracts from the state machines in Figure 0.12 and offers simple operations to control both lights. It uses actions on the transitions which trigger transitions in the actual controller in Figure 0.13.

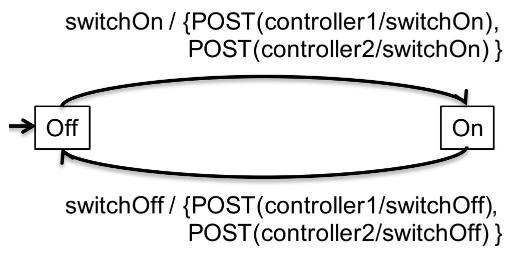

switchOn / {POST(controller1/switchOn), POST(controller2/switchOn) }
Off
On
switchOff / {POST(controller1/switchOff), POST(controller2/switchOff) }

Figure 0.13 Example service using the actuator model.

The RESTful APIs enable a number of services for WoT systems, in addition to the basic functionality of reading and controlling sensors and actuators. First, we have incremental discovery of resources via GET operations to retrieve the links to (subordinate) resources. Secondly, we can search resources similar to web applications. For instance, we can limit the result of a GET request to specific kinds of sensors. Similarly, we could filter all lights which are on or off. For more details and other services like lists or transactions we refer to [48, 55]

This is following the patterns of other web applications. In this way, we can achieve consistent APIs, from the sensors and actuators in the WoT to cloud services.

7. Modeling WoT Systems with Generic RESTful Operations

While the graphical presentation of the components and the message flow in a mashup tool is very attractive, it is also a limitation in itself. It essentially requires that the components and connections are statically defined. This is clearly limiting in cases where devices and services are connected dynamically. For instance, lights may be added dynamically.

A further, related problem in the WoT is that we often have multiple, almost identical devices in one scenario. Consider for instance a building with several lights and temperature sensors. With current mashup tools, all of these have to be modeled in an explicit way. As all of these operate in the same way, it is desirable to abstract from the individual device and just consider them as a set of components.

In this section, we present a novel concept for generic components which encapsulate a set of devices, for which we can define operations in a generic way. This generic behavior is specified in terms of REST interfaces, which enable one to program interactions with the components on a more abstract level. Similar to other generics concepts, we can aggregate different kinds of resources as long as they understand the same RESTful operations. Furthermore, we can also model dynamic sets of devices based on RESTful filtering on resources. For instance, we can create a new set of lights which have the property of being close to an emergency exit. Then, we can define an operation to turn on all of these lights.

There are also other approaches to lift programming of IoT devices to a more abstract level. For example, [31] proposes a reasoning-based approach where user goals are implemented based on suitable logic. Here, we focus on programming techniques, which can be combined with such reasoning engines.

7.1. Generic Components by 1:n Relations

In the following, we propose a simple model for generic functions using generic components. We aim to extend existing mashup concepts and employ concepts to generic or polymorphic functions as in many programming languages.

The main idea is to define a $1 : n$ relation for components in mashup tools, which generalizes the usual $1 : 1$ relations. Based on this, the data flow can be modeled as shown in Figure 0.14. The relation expresses that there are several light sub-resources associated with each room. Similarly, each building has 1 to n Rooms.

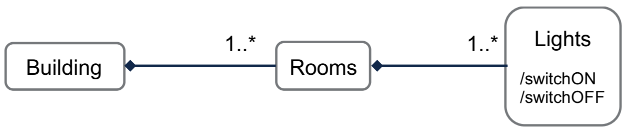


Figure 0.14 Generic components via 1:n relations.

Operationally, the idea is to conduct REST operations over all n sub-resources in a generic way. As an example, we consider switching off all lights in a room *room*1. We can thus just write a function using the generic version of POST, denoted as $POST^*$:

$$POST^*(room1/Lights/switchOFF) \qquad [7.1.1]$$

The implementation will then send the POST operation to all elements in *Lights*.

Similarly, we can switch off all lights in all rooms in *building*1:

$$POST^*(building1/Rooms/Lights/switchOFF) \qquad [7.1.2]$$

This will be translated into the following (schematic) code, ignoring error handling for now.[10]

```
for room in GET(/building1/Rooms):                        [7.1.3]
  for light in GET(/building1/room/Lights):
      POST(/building1/room/light/switchOFF)
```

This is similar to polymorphic or generic programming. Since, however, we deal here with distributed objects, failures cannot be easily ignored. There are different

[10]Note that we do not use multicast sending of GET messages, which is not possible for HTTP.

ways to handle failures. An obvious way is to handle failures locally, i.e. at each instance. Another option is to record all error messages for a generic operation in a separate aggregator object, to which a resource is added for each error that occurs, as shown in Figure 0.15. This can be analyzed after the execution of a generic operation.

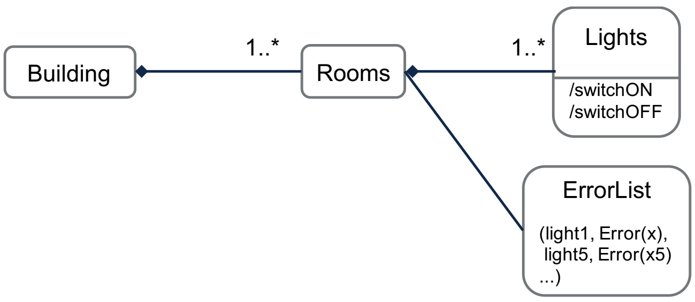


Figure 0.15 Generic programming and error handling.

By using and extending RESTful concepts, we can integrate this approach in a more consistent and coherent way into mashup tools. First, we use the sub-resource concept to aggregate objects. Secondly, we use the concept of polymorphism via generic resources. In example 7.1.1, all light resources in $room1$ have to offer the operation $switchOff$. As the lights are sub-resources, we can build the URIs for the operations by appending the resource name as in example 7.1.3. This means to perform a generic operation over linked data structures of sub-resources.

One limitation of this approach is that each of the generic operations is executed individually. An example is to calculate the energy consumption of all devices in one room. Here, we could devise generic code like

$$SUM(\ GET^{*}(room1/Lights/currentEnergy)) \qquad [7.1.4]$$

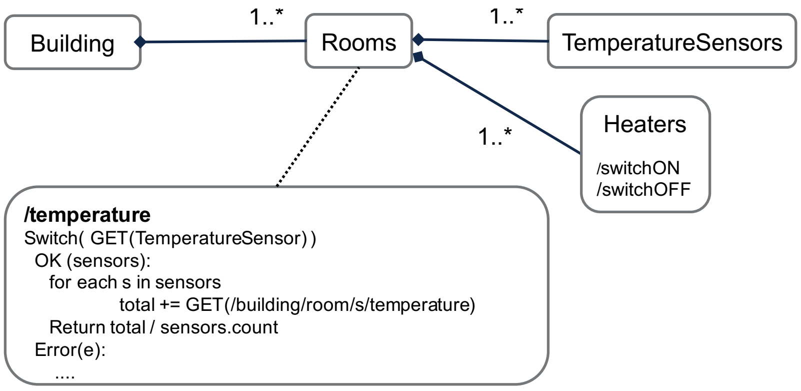
In this example, we assume GET^{*} returns a list, which is added up by SUM. Assuming functional languages like Haskell, one could also construct more complex operations [25]. For instance, we may want to calculate the total energy consumption and at the same time count the number of traversed resources. However, for more complex operations without such language support, we have to resort to a more explicit model, which is discussed in Section 7.2.

A similar concept of multicasting of REST requests is possible with CoAP multicasting [52]. However, this is designed only for requests without confirmation. As no errors or confirmations can occur, this case is easy to handle by just multicasting the request to a set of destinations. In such a case, we could of course map the construct of example 7.1.4 to such a CoAP multicast request.

7.2. Generic Operations via Sub-Resources

In addition to executing functions on each device separately, there is also a need for functionality that is aware of a set of (sub-) resources. Consider, for example, rooms which have heaters and temperature sensors. In order to control the heaters, we iterate over the sensors and switch off the corresponding heater if the temperature is above 25° C. This is shown, with simplified code, in Figure 0.16. Note that we explicitly use the sub-resources for modeling operations on a set of sub-resources.

This could also be done by the approach described in Section 7.1, but would require considerable more complex programming concepts (e.g. based on function parameters).



/temperature

```
Switch( GET(TemperatureSensor) )
  OK (sensors):
    for each s in sensors
              total += GET(/building/room/s/temperature)
    Return total / sensors.count
  Error(e):
    ....
```

Figure 0.16 Generic programming with explicit sub-resources.

The main advantage is that we use an explicit notation with the “1:n” link to represent the link between the iterator and the set of objects. Thus, the resources relations are very explicit, and we argue that this fit better into the original concepts of mashup tools. Overall, it is more flexible and extensible compared to the multicast option. Another extension is to use a flexible generation of such components based on filtering, as shown in Section 7.3.

7.3. Managing Generic Components

In this section, we discuss how resources in generic components can be managed. For instance, in the building example, we may want to add or remove sensors in a room. The approach here is to have a separate *RoomManager* component which is in charge of such updates, as shown in Figure 0.17.

In terms of a RESTful implementation, we can use sub-resources to manage a set of resources in a generic component. For instance, in the building example, we can model the sensors and controllers as a sub-resources of a room. Then, we can use

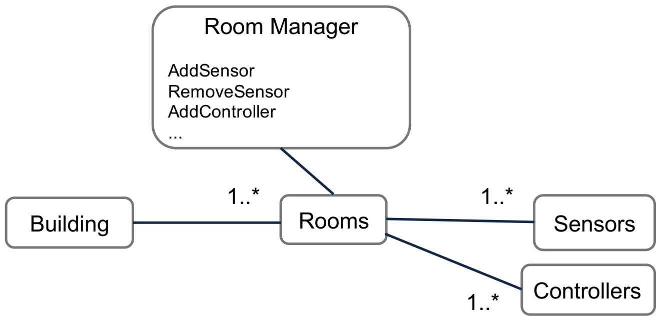


Figure 0.17 Managing generic components.

POST and DELETE to manage these.

Furthermore, we can also create sets of devices based on RESTful filtering on resources, as in usual search queries. This can be implemented with the *query* option of an URL, as defined in Section 6.1. In the building example, we may create a set of lights with specific properties. For instance, we can create a new set of lights which have the property of being close to an emergency exit by filtering based appropriate parameters in the resources.

One issue that REST does not fully address is consistency in case of multiple simultaneous operations. For instance, a generic operation on a component may be carried out by several REST operations. Then, consider that a sensor is added while this generic operation is on the way. To ensure consistency, we need to ensure that generic operations are completed before such updates. While REST does not directly support this, we should note that REST does not require that DELETE operations are executed immediately, even if the *OK* return code has been sent. According to [14], it is sufficient that the server intends to delete the resource. Hence, in such cases, we can wait for other, complex operations to conclude before deleting.

8. Conclusions

The goal of the chapter was to review tools and methodologies for the development of applications for the Web of Things (WoT), as well as to propose new tool concepts. We have reviewed the main concepts and selected tools in the area of Internet of Things mashups, which focus on modeling data flow as well as easy data integration. Furthermore, mashup tools are mainly cloud based and some tools also offer device management as well as component marketplaces. Thus, these tools aim at rapid service creation which simple concepts, tailored for the Internet of Things. On the other hand, model-driven engineering (MDE) approaches permit different views and more expressive modeling concepts. We have analyzed both concepts and techniques regarding expressiveness, suitability for the problem domain as well as ease of use and scalability.

As one particular case, we show how mashup tools can be extended to more flexible, generic operations on sets of things, based on advanced modeling concepts. This includes design and code generation in the specification of the *REST APIs* of resources offered by WoT sensors and actuators, as well as the specification of *control flow* of the individual actuators.

This shows that mashup concepts can benefit from MDE approaches, but one has to carefully balance the added expressiveness with the ease of use. Thus, we believe that there is potential in improving WoT development by combining mashup tools with MDE tools and methods.

Acknowledgments

The authors are grateful to Tanmaya Mahapatra, who very actively contributed to this work. Furthermore, we would like acknowledge the input from Dominik Schinner and Luca Chiarabini to earlier publications on this topic.

References

1. Atzori, L., Iera, A. and Morabito, G. [2010], 'The internet of things: A survey', *Computer Networks* **54**(15), 2787 – 2805.
   URL: *http://www.sciencedirect.com/science/article/pii/S1389128610001568*
2. Balasubramanian, D., Levendovszky, T., Dubey, A. and Karsai, G. [2014], Taming Multi-Paradigm Integration in a Software Architecture Description Language, *in* 'Proceedings of MPM 2014, Valencia, Spain', pp. 67–76.
3. Berry, G. [2007], SCADE: Synchronous design and validation of embedded control software, *in* 'Proceedings of GM R&D Workshop, Bangalore, India', Springer, pp. 19–33.
4. Blackstock, M. and Lea, R. [2012*a*], IoT mashups with the WoTKit, *in* 'Internet of Things (IOT), 2012 3rd International Conference on the', pp. 159–166.
5. Blackstock, M. and Lea, R. [2012*b*], Iot mashups with the WoTKit, *in* 'Internet of Things (IOT), 2012 3rd International Conference on the', IEEE, pp. 159–166.
6. Blackstock, M. and Lea, R. [2012*c*], Wotkit: A lightweight toolkit for the web of things, *in* 'Proceedings of the Third International Workshop on the Web of Things', WOT '12, ACM, New York, NY, USA, pp. 3:1–3:6.
   URL: *http://doi.acm.org/10.1145/2379756.2379759*
7. Blair, G., Bencomo, N. and France, R. [2009], 'Models@ run.time', *Computer* **42**(10), 22–27.
8. Bormann, C., Castellani, A. P. and Shelby, Z. [2012], 'CoAP: An application protocol for billions of tiny internet nodes', *IEEE Internet Computing* (2), 62–67.
9. Broy, M., Crane, M. L., Dingel, J., Hartman, A., Rumpe, B. and Selic, B. [2006], 2nd UML 2 Semantics Symposium: Formal Semantics for UML, *in* T. Khne, ed., 'Models in Software Engineering', number 4364 *in* 'Lecture Notes in Computer Science', Springer Berlin Heidelberg, pp. 318–323. DOI: 10.1007/978-3-540-69489-2_39.
10. Broy, M., Kirstan, S., Krcmar, H., Schätz, B. and Zimmermann, J. [2013], 'What is the benefit of a model-based design of embedded software systems in the car industry?', *Software Design and Development: Concepts, Methodologies, Tools, and Applications: Concepts, Methodologies, Tools,*

and Applications p. 310.

11. Cirani, S., Picone, M., Gonizzi, P., Veltri, L. and Ferrari, G. [2015], 'Iot-oas: An oauth-based authorization service architecture for secure services in iot scenarios', *IEEE Sensors Journal* **15**(2), 1224–1234.
12. Daniel, F. and Matera, M. [2014], *Mashups: Concepts, Models and Architectures*, Springer Berlin Heidelberg, Berlin, Heidelberg.
13. Derhamy, H., Eliasson, J., Delsing, J. and Priller, P. [2015], A survey of commercial frameworks for the internet of things, *in* '2015 IEEE 20th Conference on Emerging Technologies Factory Automation (ETFA)', pp. 1–8.
14. Fielding, R., Gettys, J., Mogul, J., Frystyk, H., Masinter, L., Leach, P. and Berners-Lee, T. [1999], Hypertext transfer protocol–http/1.1, Technical report.
15. Fielding, R. T. and Taylor, R. N. [2002], 'Principled design of the modern web architecture', *ACM Transactions on Internet Technology (TOIT)* **2**(2), 115–150.
16. Fleurey, F., Morin, B., Solberg, A. and Barais, O. [2011], MDE to Manage Communications with and between Resource-Constrained Systems, *in* J. Whittle, T. Clark and T. Khne, eds, 'Model Driven Engineering Languages and Systems', number 6981 *in* 'Lecture Notes in Computer Science', Springer Berlin Heidelberg, pp. 349–363. DOI: 10.1007/978-3-642-24485-8_25.
17. *Foundational Subset For Executable UML Models (FUML)* [2016], http://www.omg.org/spec/FUML/Current.
18. Giusto, D., Iera, A., Morabito, G. and Atzori, L., eds [2010], *The Internet of Things*, Springer New York, New York, NY.
19. Guinard, D., Trifa, V., Mattern, F. and Wilde, E. [2011], From the internet of things to the web of things: Resource-oriented architecture and best practices, *in* 'Architecting the Internet of Things', Springer, pp. 97–129.
20. Guinard, D., Trifa, V. and Wilde, E. [2010], A resource oriented architecture for the Web of Things, *in* 'Internet of Things (IOT), 2010', pp. 1–8.
21. Harel, D. [1987], 'Statecharts: A Visual Formalism for Complex Systems', *Sci. Comput. Program.* **8**(3), 231–274.
22. Health, N. [2014], 'How ibm's node-red is hacking together the internet of things'. TechRepublic.com [Online; posted 13-March-2014].
23. Hersent, O., Boswarthick, D. and Elloumi, O. [2012], 'Zigbee smart energy 2.0', *The Internet of Things: Key Applications and Protocols* pp. 209–236.
24. Hoyer, V. and Fischer, M. [2008], Market overview of enterprise mashup tools, *in* 'Service-Oriented Computing–ICSOC 2008', Springer, pp. 708–721.
25. Hudak, P., Peyton Jones, S., Wadler, P., Boutel, B., Fairbairn, J., Fasel, J., Guzmán, M. M., Hammond, K., Hughes, J., Johnsson, T. et al. [1992], 'Report on the programming language haskell: a non-strict, purely functional language version 1.2', *ACM SigPlan notices* **27**(5), 1–164.
26. *IBM Node-RED, A visual tool for wiring the Internet of things* [n.d.].
    URL: *http://nodered.org/*
27. Klein, U. and Namjoshi, K. S. [2011], Formalization and automated verification of RESTful behavior, *in* 'Formalization and automated verification of RESTful behavior', Vol. Computer Aided Verification, Springer, pp. 541–556.
28. Kleinfeld, R., Steglich, S., Radziwonowicz, L. and Doukas, C. [2014], glue.things: A Mashup Platform for wiring the Internet of Things with the Internet of Services, *in* 'Proceedings of the 5th International Workshop on Web of Things', WoT '14, ACM, New York, NY, USA, pp. 16–21.
    URL: *http://doi.acm.org/10.1145/2684432.2684436*
29. Ma, D. [2009], Offering rss feeds: Does it help to gain competitive advantage?, *in* 'System Sciences, 2009. HICSS '09. 42nd Hawaii International Conference on', pp. 1–10.
30. Maximilien, E. M., Wilkinson, H., Desai, N. and Tai, S. [2007], *A domain-specific language for web apis and services mashups*, Springer.
31. Mayer, S., Inhelder, N., Verborgh, R., Van de Walle, R. and Mattern, F. [2014], Configuration of smart environments made simple: Combining visual modeling with semantic metadata and reasoning, *in* 'Internet of Things (IOT), 2014 International Conference on the'.

32. Mernik, M., Heering, J. and Sloane, A. M. [2005], 'When and How to Develop Domain-specific Languages', *ACM Comput. Surv.* **37**(4), 316–344.
URL: *http://doi.acm.org/10.1145/1118890.1118892*
33. Morin, B., Barais, O., Jezequel, J.-M., Fleurey, F. and Solberg, A. [2009], 'Models@ Run.time to Support Dynamic Adaptation', *Computer* **42**(10), 44–51.
34. Myllrniemi, V., Prehofer, C., Raatikainen, M., Gurp, J. v. and Mnnist, T. [2008], Approach for Dynamically Composing Decentralised Service Architectures with Cross-Cutting Constraints, *in* R. Morrison, D. Balasubramaniam and K. Falkner, eds, 'Software Architecture', number 5292 *in* 'Lecture Notes in Computer Science', Springer Berlin Heidelberg, pp. 180–195. DOI: 10.1007/978-3-540-88030-1_14.
35. *OData Version 4.0* [2015].
URL: *http://www.odata.org/documentation/*
36. OMG [2007], OMG Unified Modeling Language TM (OMG UML Version 2.5), Technical Report formal/2015-03-01.
URL: *http://www.omg.org/spec/UML/2.5*
37. Pautasso, C., Zimmermann, O. and Leymann, F. [2008], Restful web services vs. bigweb services: making the right architectural decision, *in* 'Restful web services vs. bigweb services: making the right architectural decision', Vol. Proceedings of the 17th international conference on World Wide Web, ACM, pp. 805–814.
38. Peltz, C. [2003], 'Web services orchestration and choreography', *Computer* **36**(10), 46–52.
39. Pintus, A., Carboni, D. and Piras, A. [2012*a*], Paraimpu: a platform for a social web of things, *in* 'Proceedings of the 21st international conference companion on World Wide Web', ACM, pp. 401–404.
40. Pintus, A., Carboni, D. and Piras, A. [2012*b*], Paraimpu: A platform for a social web of things, *in* 'Proceedings of the 21st International Conference on World Wide Web', WWW '12 Companion, ACM, New York, NY, USA, pp. 401–404.
URL: *http://doi.acm.org/10.1145/2187980.2188059*
41. Porres, I. and Rauf, I. [2011], Modeling behavioral RESTful web service interfaces in UML, *in* 'Modeling behavioral RESTful web service interfaces in UML', Vol. Proceedings of the 2011 ACM Symposium on Applied Computing, ACM, pp. 1598–1605.
42. Pramudianto, F., Kamienski, C. A., Souto, E., Borelli, F., Gomes, L. L., Sadok, D. and Jarke, M. [2014], IoT Link: An Internet of Things Prototyping Toolkit, *in* 'Ubiquitous Intelligence and Computing, 2014 IEEE 11th Intl Conf on and IEEE 11th Intl Conf on and Autonomic and Trusted Computing, and IEEE 14th Intl Conf on Scalable Computing and Communications and Its Associated Workshops (UTC-ATC-ScalCom)', pp. 1–9.
43. Prehofer, C. [2013], From the Internet of Things to Trusted Apps for Things, *in* 'Green Computing and Communications (GreenCom), 2013 IEEE and Internet of Things (iThings/CPSCom), IEEE International Conference on and IEEE Cyber, Physical and Social Computing', pp. 2037–2042.
44. Prehofer, C. [2015], Models at REST or modelling RESTful interfaces for the Internet of Things, *in* 'Internet of Things (WF-IoT), 2015 IEEE 2nd World Forum on', IEEE, pp. 251–255.
45. Prehofer, C. and Chiarabini, L. [2015], From Internet of Things Mashups to Model-Based Development, *in* 'Computer Software and Applications Conference (COMPSAC), 2015 IEEE 39th Annual', IEEE, pp. 499 – 504.
46. Prehofer, C., Gurp, J. v., Stirbu, V., Satish, S., Tarkoma, S., Flora, C. d. and Liimatainen, P. P. [2010], 'Practical Web-Based Smart Spaces', *IEEE Pervasive Computing* **9**(3), 72–80.
47. Prehofer, C. and Schinner, D. [2015], Generic Operations on RESTful Resources in Mashup Tools, ACM Press, pp. 1–6.
URL: *http://dl.acm.org/citation.cfm?doid=2834791.2834795*
48. Richardson, L., Amundsen, M. and Ruby, S. [2013], *RESTful Web APIs*, O'Reilly Media, Inc.
49. Riedel, T., Yordanov, D., Fantana, N., Scholz, M. and Decker, C. [2010], A Model Driven Internet of Things, *in* 'Networked Sensing Systems (INSS), 2010 Seventh International Conference on', IEEE, pp. 265–268.
50. Ryman, A. [2001], Simple object access protocol (soap) and web services, *in* 'Proceedings of the 23rd

International Conference on Software Engineering', ICSE '01, IEEE Computer Society, Washington, DC, USA, pp. 689–.
URL: *http://dl.acm.org/citation.cfm?id=381473.381580*

51. Schreier, S. [2011], Modeling restful applications, *in* 'Modeling restful applications', Vol. Proceedings of the second international workshop on restful design, ACM, pp. 15–21.
52. Shelby, Z., Hartke, K. and Bormann, C. [2014], 'Frc 7251, the constrained application protocol (coap)', https://tools.ietf.org/html/rfc7252.
53. Straeten, R. V. D., Mens, T. and Baelen, S. V. [2008], Challenges in Model-Driven Software Engineering, *in* M. R. V. Chaudron, ed., 'Models in Software Engineering', number 5421 *in* 'Lecture Notes in Computer Science', Springer Berlin Heidelberg, pp. 35–47. DOI: 10.1007/978-3-642-01648-6_4.
54. Thangavel, D., Ma, X., Valera, A., Tan, H. X. and Tan, C. K. Y. [2014], Performance evaluation of mqtt and coap via a common middleware, *in* 'Intelligent Sensors, Sensor Networks and Information Processing (ISSNIP), 2014 IEEE Ninth International Conference on', pp. 1–6.
55. Tilkov, S. [2009], 'Rest und http', *Einsatz der Architektur des Web für Integrationsszenarien, dpunkt. verlag* .
56. Wagelaa, D. [2008], Challenges in bootstrapping a model-driven way of software development, *in* 'First International Workshop on Challenges in Model-Driven Software Engineering (ChaMDE 2008) , held in conjunction with MoDELS 2008'.
57. Wang, J., Xu, Z. and Zhang, J. [2015], Implementation strategies for csv fragment retrieval over http, *in* '2015 12th Web Information System and Application Conference (WISA)', pp. 223–228.
58. Wong, J. and Hong, J. I. [2007], Making mashups with marmite: towards end-user programming for the web, *in* 'Proceedings of the SIGCHI conference on Human factors in computing systems', ACM, pp. 1435–1444.
59. Yu, J., Benatallah, B., Casati, F. and Daniel, F. [2008], 'Understanding mashup development', *Internet Computing, IEEE* **12**(5), 44–52.
60. Zuzak, I., Budiselic, I. and Delac, G. [2011], Formal modeling of RESTful systems using finite-state machines, *in* 'Web Engineering', Springer, pp. 346–360.