Facilitating the Integration and Interaction of Real-World Services for the Web of Things

Simon Mayer
Inst. for Pervasive Computing
ETH Zurich
Email: mayersi@inf.ethz.ch

Dominique Guinard
Inst. for Pervasive Computing
ETH Zurich
and SAP Research, Zurich
Email: dguinard@inf.ethz.ch

Vlad Trifa
Inst. for Pervasive Computing
ETH Zurich
and SAP Research, Zurich
Email: trifa@inf.ethz.ch

Abstract—The "Web of Things" (WoT) is a vision of a World Wide Web that reaches into the physical world by providing a seamless integration of digitally augmented everyday objects. In this paper, we present the *AutoWoT* project, a toolkit that facilitates the rapid integration of smart devices into the Web. It thereby lowers the entry barrier for users to expose data and services provided by their smart things. AutoWoT offers a generic way of modeling Web resources and automatically builds web server components which expose the functionality of such digitally augmented devices. By abstracting the specific implementation of Web protocols, the toolkit enables prototype developers to focus on their use-case. In this paper we show how AutoWoT can be used to considerably facilitate the process of populating the Web of Things and illustrate the benefits with a concrete prototype application, a presence awareness tool that combines multiple Web-enabled real-world devices and services within a physical mashup.

I. INTRODUCTION

The "Web of Things" (WoT) [1], [2] envisions a World Wide Web that provides access to the data and services of physical objects through the use of widely deployed and highly accepted Web protocols and standards like the Hypertext Transfer Protocol (HTTP) and is built as a Representational State Transfer (REST) architecture [3]. Ultimately, the goal is to enable end-users to reuse real-world functionality provided by *smart things* (e.g., wireless sensor and actuator networks, tagged objects, electronic appliances, etc.) and combine these services within *physical mashups* [1]. As the potential benefit of a single mashup largely depends upon the data and services that are available for it to consume and process, a crucial step towards increasing the utility of such compositions is to lower the effort necessary for "WoT-enabling" smart devices that were initially not designed to provide their services via the Web.

The essence of the REST principle is to focus on creating loosely coupled services that can easily be reused. REST is core to the Web and uses URIs for encapsulating and identifying services. It relies on HTTP (1.1) as an *application protocol* and uses its verbs (e.g. `GET`, `POST`, `PUT`) to describe basic service semantics in a uniform manner. It finally decouples services from their presentation and thus enables clients to select the desired data format, for instance JavaScript Object Notation (JSON), at runtime. This, together with the pervasive availability of HTTP libraries, makes REST a very

good architecture for the integration of digitally augmented devices and a natural candidate to build a universal application programming interface (API) for smart things [1].

Integrating a digitally augmented device into the Web of Things currently requires a very good understanding of the REST principles and constraints as well as skills in programming web servers. These requirements are a barrier to the rapid prototyping of new devices in the WoT. This paper describes the *AutoWoT Deployment Toolkit*, a meta-program that enables the rapid integration of devices into the Web of Things. AutoWoT supports the user in creating a hierarchical description of the services provided by a specific device and then automatically creates web server software that RESTfully exposes the functionality offered by the smart object. Featuring an intuitive graphical user interface, AutoWoT enables developers and tech-savvy users to rapidly implement Web of Things prototypes by automating the Web-enablement of resources. As developers are not anymore required to implement Web protocols, they can better focus on the core functionality of their prototypes. The automatic creation of the web server software furthermore guarantees that the resulting program follows the REST specifications, thus relieving AutoWoT users from having to learn them and avoiding common mistakes and misinterpretations of the guidelines.

AutoWoT allows clients to interact with a device using multiple representational formats (i.e., JSON, XML and HTML). To foster machine-to-machine scenarios, it also provides Microformat-based semantic markup on the capabilities of the Web-enabled devices. The toolkit can furthermore be used in conjunction with virtual resources, for instance to enable Web-based interaction with databases. Within our efforts to populate the Web of Things, AutoWoT has proven to facilitate the integration of different resources, including RFID readers, LEGO NXT controllers, smart energy meters and Sun SPOT sensor platforms. To illustrate the benefits and ease of use of the toolkit, we have created an advanced physical/virtual mashup prototype that implements a personal presence awareness tool, i.e., analyzes a user's presence/absence characteristics for a specific location and uses this data to switch a lamp on the user's workplace.

After a survey of related work in Section II, we will detail different aspects of the AutoWoT Deployment Toolkit, starting

with a description of the common conceptional properties of resources from a RESTful/Web-standpoint in Section III. After that, we are going to describe the AutoWoT software itself in Section IV and elaborate on the structure and properties of the created web servers. To demonstrate the utility of this program when integrating resources into the Web of Things, an account of some devices that were Web-enabled using AutoWoT will be given in Section V and their composition within our prototype mashup will be described. We will conclude and motivate further work in Section VI.

II. RELATED WORK

The recent growing interest for lightweight service architectures based on REST has given birth to a number of projects that simplify the development of RESTful applications.

Most of these toolkits focus on providing developers with language constructs directly extracted from the guidelines of RESTful architectures. The Restlet[1] framework is a good example of a Java toolkit offering language constructs directly inspired by Fielding's seminal thesis on REST [3]. Some toolkits provide even higher abstractions that enable RESTful applications and prototypes to be created in a manner closer to traditional (object oriented) software development. Toolkits such as Jersey [4] or RESTeasy[2] implementing the JAX-RS (Java API for RESTful Web Services) standard[3] fall into this category. While very useful for experienced software developers, these toolkits still require a broad understanding of the concepts behind REST and require to write web server program code which is a significant barrier in fast prototyping Web-enabled devices.

Based on these shortcomings, tools for the automatic generation of RESTful web services [5] have started to appear. This is the case of the RESTful module of the Netbeans Java IDE (Integrated Development Environment)[4]. The idea of AutoWoT is similar to these tools, however, it is tailored to the needs of the physical world and especially to assisting developers when creating RESTful APIs for smart things based on the requirements of a resource oriented architecture for the Web of Things [1]. In particular, it provides a wizard-based assistant that helps developers to visually define the resources the smart things are composed of. It further automatically generates semantically annotated HTML representations of the smart things.

III. ABSTRACT RESOURCE PROPERTIES

Providing a tool that facilitates the Web integration of physical things requires to abstract from specific devices and provide a generic way to characterize devices and their functionality. AutoWoT uses the information contained within such a device description to generate software that provides Web access to the physical device. A conceptional description should hold information on *interactions* possible with devices,

---

[1]http://www.restlet.org
[2]http://jboss.org/resteasy
[3]http://jcp.org/en/jsr/detail?id=311
[4]http://www.netbeans.org/kb/

for instance on the kind of data that they provide or on functions that they are able to carry out. Additionally, in order to make device functions easily browseable, we have decided to include information on the *relationship* of different resources with each other (e.g., a light sensor being a child resource of a resource that identifies a collection of different sensors).

The process of enabling a device for the Web implies a transformation of the physical thing into a hierarchical collection of Web resources [1]. These resources act as interfaces for Web clients to trigger device functions (cf. Figure 1): For instance, a HTTP GET instruction that is sent to the URL http://.../mydevice/sensors/temperature would retreive the current temperature as measured by a temperature sensor. Thus, when creating the conceptional descriptions of physical devices, we specifically need to describe their behavior with respect to the main HTTP operations GET, POST, PUT and DELETE. In the case of the AutoWoT toolkit, we have created a classification of the different features of resources into five categories:

- **Getters** refer to data that is provided by a resource. In order to retrieve that data from the device, every Getter depends on a callback method that has to be provided by the user. Information on a Getter can be enriched by specifying a description of the Getter and/or an OnChange-method that enables push-based publish/subscribe paradigms. Getters are identified by their name, which, together with the optional description, is used to provide semantic markup when the resource is displayed to a client.
- **Posters** refer to HTTP POST-based interaction with a resource. They require a method that is triggered whenever the resource receives an HTTP POST request (with the POST argument as parameter). Furthermore, Posters feature *Argument Type* and *Presentation Type* properties that are used in request handling and automatic interface creation, respectively. Posters are identified by a name and can be described using the *Poster Description* property.
- **Putters** refer to interaction capabilities of a device that is triggered when the associated Web resource receives
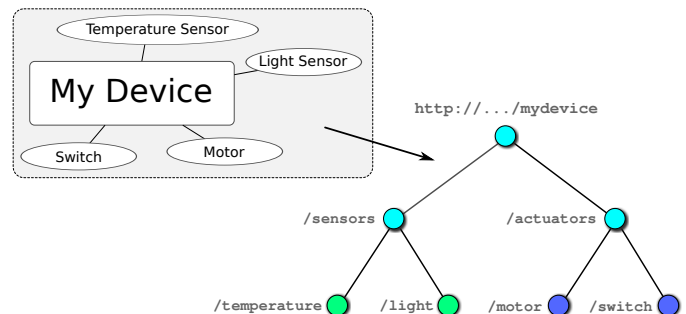

Fig. 1.   A real-world device and its hierarchical Web representation.

an HTTP `PUT` request. Putters are similar to Posters in that they are described by a name, description, callback method, argument type and presentation type.

- **Deleters** refer to handling HTTP `DELETE` requests that are received by a resource. They are described by a name, description and callback method.
- **Children** are the means of characterizing relationships between resources. They are described using the properties *Name*, *URI*, *Methods Allowed* and *Description*. Additionally, the optional property *Collection Callback Method* may be used. This attribute is used to describe resources that have multiple uniform children whose presence may vary at runtime. As an example for such a constellation, consider a Web resource that exposes multiple homogeneous resources as a list (e.g., "All sensor nodes currently in range"): This resource should react to the dynamic joining and leaving of single nodes by adding and removing them from its list of sub-resources.

The callback methods that have to be specified in the descriptions of these abstractions link services provided by the Web resources to the actual real-world functionality of a physical device. We refer to the piece of software that implements the callback methods as the *Driver Program* of a specific device. The information on a particular device is aggregated into a single XML document, the *Resource Configuration*. In addition to the interaction and relationship information, this configuration file also features wildcard-tokens that are replaced by actual values, for instance the name of a specific member of a resource collection, at runtime. As an example, consider a specific resource that carries the (possibly dynamic) name *"TestSensor3"* and is a sub-resource of a collection of sensors called *"sensors"*. The wildcard-tokens attend to structuring the Web representation such that this specific sensor is accessible at the URL `http://.../sensors/TestSensor3`.

IV. CORE

In this section, we present the main features of the AutoWoT software and discuss a number of properties of the web server software.

A. AutoWoT Application Flow

Before it can start generating web server software for a specific device, AutoWoT lets the user specify the structure and properties of the resource that represents the (physical) object to be included into the Web of Things via a Drag-and-Drop-based GUI (cf. Figure 2).

To extend the (default) top-level resource, the user may select one of the suggested options (e.g., a *Getter*, a *Child* or a *Description*) and drag the corresponding icon into the buildup-area on the left. AutoWoT then prompts for necessary information on the selected option and, as soon as all required information has been entered, creates and displays the matching resource. The user may continue modifying already created resources and can also browse the resource structure by selecting child/parent resources in the GUI. Furthermore, it

is possible to reconfigure resources created during earlier sessions by dropping their XML representation into the buildup-area.

After having created and finished setting up a Web resource structure, AutoWoT implements the web server program that connects the specified device to the Web of Things. Before this can happen, the user is prompted for the name and location of the *Handler* class that allows access to the functionality of the device via the different callback methods specified. While AutoWoT is able to create plain Java-based software to expose devices via the Web, we have integrated the possibility of deploying AutoWoT-built web servers as bundles that follow the specifications of the OSGi alliance[5]. The reason for also providing this option is to provide an architecture that structurally decouples the web server from the device driver program and thus to support users in integrating AutoWoT-generated software with existing systems. The OSGi framework provides a modularization system on top of the Java platform and thus represents a natural choice when creating modules (so-called *OSGi Bundles*) of composite applications. OSGi provides a feature called "Declarative Services" (DS) for facilitating the integration of different bundles via so-called *OSGi Components*. Indeed, what AutoWoT does is to create a standalone OSGi bundle that incorporates a component offering web server functionality that relies on OSGi DS for accessing the device's callback methods that are contained in the Driver Program (cf. Figure 3). To foster smooth interaction between the two software packages, AutoWoT additionally creates the DS component definitions and manifest additions that are necessary to wrap the driver software within an OSGi bundle.

AutoWoT stores the source code of the generated web server in a directory hierarchy that reflects the user's desired package configuration. The required Java libraries and the images and CSS-scripts that are used when building the HTML
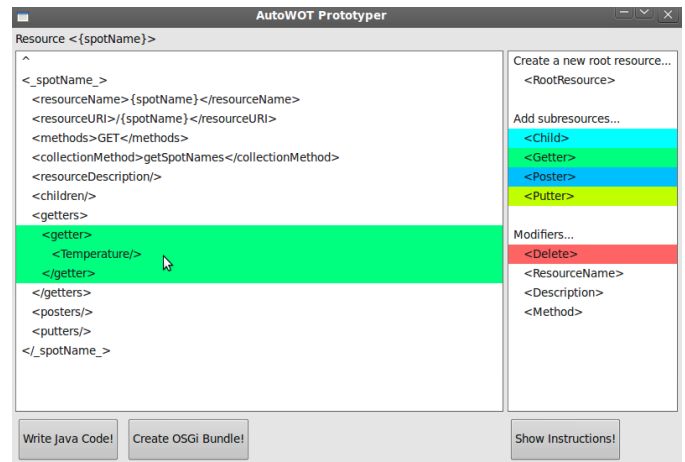
[5]`http://www.osgi.org`

Fig. 2. The AutoWoT User Interface. Left: The currently selected resource, a collection of Sun SPOT sensor platforms with a single *Getter*, each. Right: Suggested extension points for the resource.
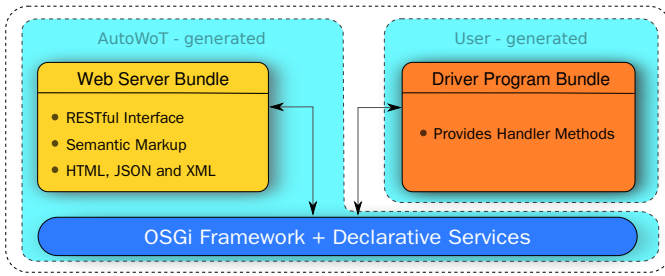
Fig. 3. The structure of AutoWoT-built web server software (OSGi-based) and its interaction with the provided handler methods.

representations of the resources are provided as well. The user may import these files into an IDE or directly create `.jar` files to be started within an OSGi framework. If the user decides to modify the Web resource at a later point in time, AutoWoT can be instructed to re-generate the software such that it reflects the new structure and/or functionality of the device – the only thing to be done by hand is thus the implementation of the callback routines.

B. System Design of AutoWoT-Generated Web Servers

The web server software generated by AutoWoT relies on *Restlet*, a lightweight open source REST framework for Java. For every structural (sub-)resource that has been defined by the user, AutoWoT creates a Java class and a routing instruction to that class in the main Restlet *Router*. These resource-specific classes inherit from an abstract *BaseResource* and implement the Getters, Posters, Putters and Deleters as specified by the user together with callbacks to trigger the associated device functionality at runtime. As a stub for basic event-based interaction, AutoWoT-generated web servers run several background threads that poll each Getter's callback method and trigger the method specified in its *OnChange*-property as soon as the callback's output changes.

C. Interaction with AutoWoT-enabled Resources

The single most important task of a WoT-enabling web server is to supply HTTP clients with a suitable representation of the enabled devices. AutoWoT-built web servers thus expose resources in three different formats where the actual resource representation that is delivered to a client is determined using the HTTP `accept`-header:

- **HTML** Especially for facilitating the interaction with human clients, an HTML/CSS-based website is created for every resource. These basic interfaces give users the ability to (a) navigate within the resource structure by following links to parents/children, (b) grasp the information provided by the resource (i.e., by the resource's Getters) and (c) interact with the Posters of the resource via an annotated web form. The provided HTML pages are enriched with Microformat-based semantic markup.
- **JSON** AutoWoT supports straightforward Web integration using the *JavaScript Object Notation* (JSON) which can be directly translated to JavaScript objects. It exposes

strings that describe a smart things' interaction capabilities and the relationships to other resources by including information on the Children, Getters, Posters, Putters and Deleters of the resource.
- **XML** Finally, an XML-based representation is offered for increasing the interoperability of the resources. As for the JSON string, this reproduction of the resource is easily parseable by clients.

The HTML, JSON and XML representations have exactly the same structure for all resources that are WoT-enabled using AutoWoT as they are produced by the very same subroutine for every resource. If, for a specific resource, a different representation is desired, that method may be overridden in the resource's implementation.

D. Semantic Markup of Resources

In order to enable advanced machine-to-machine interaction within the Web of Things context (i.e., when computers act as clients of resources), resources have to provide some kind of (machine-readable) information about the services they provide. This markup can then be used by clients to classify and systematically make use of the information and interaction functionalities provided by the resource. AutoWoT provides such information on resources by also creating a Microformat-based description of the resource that is deduced from the information provided by the user when configuring the Web resource. Specifically, this information is used to provide semantic markup on the resource using the `hRESTS` Microformat [6] that contains data describing the interaction possibilities regarding HTTP `GET`, `POST`, `PUT` and `DELETE` operations.

V. PROTOTYPE APPLICATIONS

In this section, we illustrate the convenience of using AutoWoT for the Web-enabling of smart devices using the example of our WoT-ITIPA (Interactively Trained Illumination-based Presence Awareness) prototype, a personal presence awareness tool that monitors users based on a real-time analysis of the light patterns at their working place. Within this prototype, multiple devices and services collaborate to (a) collect data on the current ambient light, (b) process this data using frequency analysis and a feed-forward artificial neural network (ANN) and (c) output and physically react to the result (i.e., switch on or off a lamp depending on whether the user is present or not). To provide a physical user interface and especially for training the neural network, the system uses an RFID reader in conjunction with a RESTful database. After describing the processes of connecting the different devices used within ITIPA to the Web of Things with the help of the AutoWoT toolkit, we will detail the structure and functionality of the monitoring tool itself.

A. Sun SPOT Sensor Platforms

Sun SPOT sensor platforms (*Sun SPOTs*), are Java-programmable wireless embedded devices featuring multiple

sensors (ambient light, temperature, acceleration), several tri-colored LEDs and a radio for communication using the IEEE 802.15.4 standard. These were the first devices that were Web-enabled using the AutoWoT toolkit. The resulting software was fully built using AutoWoT and employs callback methods contained in a driver program that had been developed during earlier work in our group. It is capable of RESTfully exposing multiple dynamically joining and leaving Sun SPOTs and their sensors and actuators. Additionally, it provides a RESTful con-figuration interface that allows administrators to, for instance, block specific sensor nodes or change the system's wireless broadcast port directly inside their Web browsers. Using the AutoWoT toolkit, the process of configuring and assembling the software was accomplished in under an hour, compared to about four days when the toolkit had not yet been available. Indeed, the time and effort necessary to Web-enable these sensor platforms were the initial trigger that inspired us to design AutoWoT.

B. Phidgets RFID Reader

AutoWoT was also used to integrate an RFID reader from Phidgets, Inc. into the Web of Things. The corresponding Web resource features a very simple structure of just a single resource (i.e., a root resource without any children). It provides a single Getter that informs about the currently read RFID tag.

C. Ploggs Smart Plugs

The third kind of physical devices that are used in con-junction with the ITIPA prototype are *ploggs*[6] smart meter plugs that provide data about the current power consumption of attached devices and can also be switched on and off remotely. AutoWoT has been used to provide a wrapper for several commands that can be transmitted to these outlets. By sending a HTTP `POST` request to the Web resource that represents a specific plogg, one can toggle the power supply of attached devices.

D. Enabling virtual resources for the Web of Things

AutoWoT is not constrained to WoT-enabling physical de-vices only. Rather, the toolkit can be applied to any function-ality that can be triggered via a Java program. To exemplify this capability, we have created multiple prototypes including a REST database providing retrieve, store, update and delete operations for key-value pairs using the HTTP methods `GET`, `PUT`, `POST` and `DELETE`, respectively.

E. The WoT-ITIPA Prototype

The ITIPA prototype makes use of a Web-enabled Sun SPOT, a Phidgets RFID reader, a smart plug (with a connected table lamp) and an instance of our RESTful database to provide a presence awareness tool capable of monitoring a single user. To find out whether a person is present or not, this mashup uses data obtained from a Sun SPOT's photosensor and, after performing a discrete Fourier transform, feeds the resulting spectrum into an artificial neural network. The ANN

---

[6]http://www.plogginternational.com/

---

is designed such that its (boolean) output directly yields the information on the user's presence which is used to control the smart plug and thereby switch the table lamp on or off, respectively. An RFID reader has been integrated for training and performance control purposes and to provide a physical user interface for managing the prototype: The system reacts to different RFID tags that are placed on the reader, where the instruction associated to a specific tag is resolved at runtime using the RESTful database. In the current implementation of ITIPA, six different instructions are used:

- *NormalMode*: This instruction is usually associated to the situation where no RFID tag is read. ITIPA fetches and processes ambient light data while neither training the ANN nor assessing its results.
- *PrivacyMode*: This instruction/tag is used to suspend ITIPA. When suspended, the application enters a self-correction mode and uses the collected training sets to further enhance the ANN's performance.
- *Present* and *Absent*: Using these instructions, a user can instruct ITIPA to enter a self-assessment routine where it creates correctness statistics and publishes the results on the Web via the REST database.
- *PresentTrain* and *AbsentTrain*: A user may place tags associated to these instructions on the reader to instruct ITIPA to engage in training the neural network. Using the tags, the system conveys information on the monitored person's state to the neural network which uses that data to construct training sets and thereby improve its performance.

By discussing this prototype, we do not want to emphasize explicitly its performance (regarding its ability to reliably detect a user's presence) but rather the ease and speed of its creation and implementation. Even more striking than the conciseness of the ITIPA core program (140 lines of code) is the time effort necessary for its creation: The first working prototype was runnable and produced ample results after about two hours of implementation work. However, we want to point out that most of the workload (about 80%) was required to get to know and implement the neural network (using the *Encog* Java library) and the associated training routines while the integration of the Web-enabled devices was by far the simplest step and accomplished within only a few minutes. Specifically, the JSON data interchange format proved to be easy and fast to parse and process.

F. ITIPA Evaluation by Prototyping

The prototype has been tested within a real-world deploy-ment in our office where it was monitoring one of the authors for multiple days. The system was trained for a period of half a day, i.e., the user swapped the *PresentTrain*- and *Absent-Train*-tags every time he left and returned, respectively. The remaining time of two days was used to assess the system by placing the *Present*- and *Absent*-tags on the reader. The system was switched off during the night (i.e., the privacy mode was activated) in order to avoid distorting the statistics (during the night, the illumination in the monitored office is zero). The
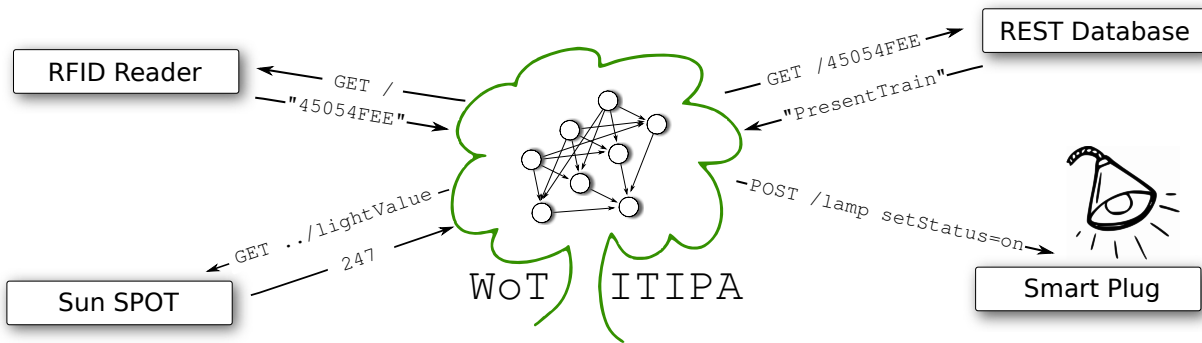
Fig. 4. Structure of the WoT-ITIPA mashup.

results indicate an error rate of about 24%, subdivided into roughly 17% false negatives and 7% false positives which is a surprisingly low value when considering the crudeness of the learning algorithm employed and especially the complexity of the provided data. Thus, we are confident that a better adaptation of the neural network to the problem would improve the system's performance.

As a next step, we reused the WoT-ITIPA core program but, instead of fetching information on the current illumination from a Sun SPOT (by accessing the SPOT's /sensors/light URL), retrieved the data from its accelerometer using the address /sensors/tilt (i.e., changed one of the input parameters of the program). After attaching the sensor node to one of our flexible office chairs and re-training the ANN, the system was immediately able to extract the user's presence, this time yielding a very high reliability of 98% (mainly due to the simpleness of the used accelerometer data). This further emphasizes the flexibility and design freedom associated with using AutoWoT for the Web-integration of smart things: Conveniently providing data from physical devices via the World Wide Web, AutoWoT facilitates and accelerates the creation of physical mashups.

VI. CONCLUSION

In this paper, we have presented the AutoWoT Deployment Toolkit. The key contributions of this meta-program are that it uses a GUI to let the user define the hierarchical structure and properties of a smart device that is to be integrated into the Web of Things. It then incorporates a single definition file, the XML structure configuration, into web server software that generates different RESTful Web representations of the device and also creates semantically annotated HTML to support discovery and look-up services.

We furthermore presented multiple examples of real-world entities and virtual resources that were enabled for the Web of Things using AutoWoT. The toolkit has proven to considerably facilitate the process of Web-enabling devices and thus allows for rapidly populating the Web of Things with different kinds of physical and virtual resources. To illustrate the ease of using AutoWoT-enabled devices within physical mashup applications, we presented the WoT-ITIPA prototype that incorporates multiple Web-enabled resources to retrieve

and process sensor information and physically react to this data.

A particularly interesting property of AutoWoT is that it creates multiple web representations of resources and annotates these, e.g. using Microformats, with semantic markup on their functionalities and interaction capabilities. This gives rise to many scenarios where machines use and process these representations, for instance with respect to automatic user interface creation or even for enabling fully automated RESTful machine-to-machine interaction.

The AutoWoT Toolkit represents a first step towards facilitating the integration of smart devices into the Web of Things. The idea of enabling anybody to specify the structure of resources to attach to the WoT using an easy-to-handle user interface does, in our opinion, hold great potential and should be explored further. Future investigations of this topic should thus include an extension of the basic resource model towards enhancing its expressiveness with respect to defining further properties of resources (e.g., the product category or brand as specified in the hProduct Microformat). Additionally, there is still room for improvements relating to the ease-of-use of the toolkit, specifically with respect to the integration of the automatically created web server implementation with the user-supplied core software which currently does require basic skills in handling the Eclipse IDE. A revision of the toolkit should also include a better user interface and better abstractions to further increase AutoWoT's ease of use.

REFERENCES

[1] D. Guinard, V. Trifa, and E. Wilde, "A Resource Oriented Architecture for the Web of Things," in *Proceedings of IoT 2010 (International Conference on the Internet of Things)*, Tokyo, Japan, Nov. 2010.

[2] E. Wilde, "Putting Things to REST," School of Information, UC Berkeley, Tech. Rep. 2007-015, 2007.

[3] R. T. Fielding, "Architectural Styles and the Design of Network-based Software Architectures," Ph.D. dissertation, University of California, Irvine, 2000.

[4] M. Hadley, S. Pericas-Geertsen, and P. Sandoz, "Exploring Hypermedia Support in Jersey," in *Proceedings of the First International Workshop on RESTful Design*. Raleigh, North Carolina: ACM, 2010, pp. 10–15.

[5] L. Richardson and S. Ruby, *RESTful Web Services*. O'Reilly Media, Inc., May 2007.

[6] J. Kopecký, K. Gomadam, and T. Vitvar, "hRESTS: An HTML Microformat for Describing RESTful Web Services," in *Proceedings of the 2008 IEEE/WIC/ACM International Conference on Web Intelligence and Intelligent Agent Technology*, Washington, DC, USA, 2008, pp. 619–625.