Semantic Metadata to Support Device Interaction in Smart Environments

Simon Mayer
ETH Zurich
Universitätstrasse 6
8092 Zurich, Switzerland
simon.mayer@inf.ethz.ch

Gianin Basler
ETH Zurich
Universitätstrasse 6
8092 Zurich, Switzerland
baslergi@student.ethz.ch

UbiComp'13 Adjunct, September 8–12, 2013, Zurich, Switzerland.

ACM 978-1-4503-2215-7/13/09...$15.00.

http://dx.doi.org/10.1145/2494091.2497584

Abstract
Facilitating the interaction of human users and machines with smart devices is important to drive the successful adoption of the Internet of Things in people's homes and at their workplaces. In this paper, we present a system that helps users control their smart environment, by embedding semantic metadata in the representations of smart things. The system enables users to specify a desirable state of their smart environment and produces a machine-readable description that details which steps are necessary to reach this state, where each step corresponds to a Web request to a smart device. A client application that, for instance, runs on the user's smartphone, then implements these steps to reconfigure the user's smart environment. We report on our experiences when integrating semantic technologies with smart devices and on two use cases from the home and office automation domains that we implemented in our office space.

Author Keywords
Machine-Machine Interaction, Reasoning, Smart Environments, Semantics, Web of Things.

ACM Classification Keywords
H.5.3 [Group and Organization Interfaces]: Web-based interaction.

Introduction

The emerging *Web of Things (WoT)* is a concrete implementation of the Internet of Things (IoT) vision that focuses on establishing *application-level connectivity* between heterogeneous devices [3]. It is based on protocols and patterns that have proven to be successful in the World Wide Web such as caching, load balancing, and searching as well as the stateless nature of the HTTP protocol. By applying these to physical devices and thus creating "smart things" that are modeled in a resource-oriented fashion and according to the Representational State Transfer (REST) constraints, it is possible to construct lightweight applications that leverage large amounts of real-time data and physical functionality.

The ability for people, devices, and software to find and connect with other information resources and physical artifacts is a crucial factor for the successful integration of the IoT with people's homes and working environments. Especially when considering densely populated smart environments, it is difficult to find and utilize relevant services in a fast and user-friendly way. We claim that the problem of finding and using smart devices that provide services relevant to the task a user wants to achieve in a given setting (e.g., displaying pictures from a networked digital camera on an ambient display) can be solved in WoT environments by embedding information about the capabilities of a smart device within its representation. This description of *what* services a device provides can then be integrated with its REST program interface, i.e. information about *how* the provided services can be used by clients.

In this paper, we describe a mechanism that integrates semantic technologies with the WoT. Our system enables human users and machines to find relevant devices and utilize their services on demand. It allows users to specify *goals* using interface devices (e.g., smartphones) which encode a user's desired state of his smart environment, for instance, to regulate the ambient temperature at his current location. The system then uses a reasoning service to determine whether these goals can be reached given the set of services available to the user. If the reasoner finds a path from the current state of the smart environment to the goal state, it produces a *proof* which details why it believes that the goal state can indeed be reached. From this proof, our system distills the necessary steps to reach the goal where one such step corresponds to a single Web request to a device or service. The client interface then executes these requests and thereby gradually modifies the user's environment to reach the desired goal state. We have deployed this system in our research group's office space, where we considered several use cases. We present two of these in this paper: *Music Escort* enables users to create music streams that follow them from room to room when they roam our office space. *Room Configurator* allows users to input their preferences regarding their current whereabouts and then tries to configure their environment accordingly.

Related Work

The basis of our system is formed by metadata that is embedded in the Web representations of smart devices and integrates a description of devices' Web APIs with semantic information about the capabilities of the described smart things. This data thus allows to specify the program interface for using services offered by Web-enabled devices and provides the necessary information for reasoning about the capabilities of environments that contain one or multiple smart things.

The problem of specifying a *program interface* for Web services, and using such a specification to create mashups of interlinked services has, for instance, been approached in the JOpera project [6]. JOpera provides a visual language to define control and data flow graphs and an execution engine for such workflows. In [1], more emphasis is placed on the linking of Web resources to guide RESTful machine-to-machine interaction by fully exploiting the REST "hypermedia as the engine of application state" (HATEOAS) constraint to coordinate interacting services. This approach is also increasingly being adopted by industry, where emerging REST frameworks require application designers to satisfy HATEOAS as a primary goal [5].

Regarding the *description of device capabilities*, our goal is to enable a semantic reasoner to deduce a possible path to a defined goal from the user's inputs and semantic metadata provided by devices. To achieve this, we build on the RESTdesc language [7] which integrates services' REST interface descriptions with metadata that is required to reason about device capabilities. The required information is encoded in the Notation3 (N3) format, which extends the RDF data model by adding assertion and logic capabilities.

As an example of other approaches that tackle the problem of supporting end users in smart environments, we mention the MIT Oxygen project, and especially its *GOALS* and *MetaGlue* [2] components. Our approach is distinguished from these systems as it avoids tight coupling of the involved smart things: Any device that provides appropriate semantic metadata can join our system without any reconfiguration, and can immediately be discovered and used by human users and machine clients. As a consequence of this decoupling, our system remains easy to setup and maintain, as all involved smart things can function by themselves, without any supporting infrastructure. Rather than tightly integrating the services provided in a smart environment, we thus use the embedded semantic metadata as a common ground for enabling automatic collaboration between smart devices.

System Design

In this section, we give details about the semantic metadata that is embedded in the Web representations of our smart things and the discovery of these descriptions. We also discuss the reasoning process and the implementation of the reasoner's proof by the client interface.

System Context

Our system leverages multiple services that are deployed in our research group's office space. All devices that are connected to our group's network have access to a Web-based management infrastructure [4] that provides a look-up service to search for smart devices and also keeps track of their location. For instance, a user can contact this system to find every device of type `dbpedia.org/resource/Thermostat` on a specific level of our building, or request it to deliver all devices that are located in a particular room on that level. Devices can be localized in our office space due to a custom-built indoor localization system with room-level accuracy. Alternatively, device locations can be set to a fixed value – this is especially useful to integrate services that are not deployed on a physical device but rather can be run on any server (e.g., as a cloud service). One example for such a service is the reasoning software that we use to collect and analyze the semantic descriptions of deployed devices.

In the following, we will frequently reference two use cases that were implemented on top of our system. The first, *Music Escort*, tracks users' locations (i.e., their mobile phones) and automatically adjusts audio streams to have their favorite music follow them around in our office space. This mashup involves many devices and complex interactions between them, including the on-the-fly creation and deletion of Web resources. We use it to exemplify how our system deals with more dynamic situations where information can also become invalid (e.g., as a user's location is updated) and the client has to compensate for such changes. Apart from the look-up and reasoning services, *Music Escort* makes use of several more devices and services:

- Multiple *stream receivers*[1] that can render audio streams and are distributed among different rooms in our office space.
- A single *stream transmitter* that can be configured to stream an audio track to one or multiple receivers.
- The *client application* that is running on the user's smartphone and can track his location in real time using a custom indoor localization system.

The second use case, *Room Configurator*, extends *Music Escort* to enable users to configure not only the currently playing music, but rather enter several comfort parameters (e.g., their comfort temperature) into the client interface, which has also been implemented as a smartphone application. This device in turn communicates with the surroundings to try and configure the user's whereabouts to match these settings. Our setup includes several different kinds of devices:

- Several *smart thermostats* are used to control the temperature within a room. They run the Constrained Application Protocol (CoAP), a protocol similar to HTTP, albeit for very resource-constrained devices. To communicate with the thermostats, a HTTP/CoAP cross-proxy is used.
- *Sun SPOT sensor nodes*, more powerful devices that can run their own Web server, are used to sense and report the ambient lighting level in a room.
- Mock-up implementations of *smart alarm clocks* enable users to configure alert signals and ambient reminders.

Embedded Semantic Metadata

All considered devices feature semantic descriptions of their program interface and the functionality they provide to enable automatic interaction between them. For instance, the look-up service provides a description that holds information about what requests to send to it to find an instance of a service of a specific `type` at a specific `location`[2]. The semantic metadata serves to decouple the different components of our system: For instance, the functioning of the whole system is not tied to using our own lookup service. Rather, any Web service that can be sent a semantic type and a location and will return URLs of devices of that very type at the given location automatically integrates with our system. To give another concrete example, the stream transmitter provides

[1] We use VLC (`videolan.org/vlc`) to stream media files.

[2] We use the semantic categories `type` and `location` as defined by the Dublin Core Metadata Initiative (`purl.org/dc/elements/1.1`) and the W3C WGS84 Geo Positioning vocabularies, respectively.

information that tells clients how to set its associated stream to a Playing or Paused state, or how to add receivers to a stream. Finally, the reasoner has an associated semantic description that tells clients how to request a proof. An example of what the sematic data that is provided by our smart things (in this case, the stream transmitter) looks like is given here:

```
@prefix dc: <purl.org/dc/elements/1.1/>.
@prefix http: <www.w3.org/2011/http#>.
@prefix headers: <www.w3.org/2011/http-
   headers#>.
@prefix dbpedia: <dbpedia.org/resource/>.

{
   _:media dc:title ?songTitle.
}
=>
{
   _:request http:methodName "GET";
     http:requestURI (</media?media=>?
         songTitle);
     headers:accept "text/plain";
     http:resp [ http:body ?path ].

   ?path a dbpedia:Path.
}.
```

From this information (a logical implication), the reasoner distills that it is possible to obtain an instance of a dbpedia:Path from the response message body of an HTTP GET to the resource at http://[...]/media?media=*songTitle*. The HTTP Accept request-header is set to text/plain and the variable *songTitle* is replaced by an instance of dc:title. An instance of dbpedia:Path is in turn accepted by the stream transmitter when creating a resource of type musicontology.com/#term_Stream which then can serve as an input when a client wants to reach the goal Playing (the semantic descriptions corresponding to these interactions are not produced in this paper).

To describe and classify devices in our office environment, we made use of well-known public ontologies such as the *Dublin Core Metadata Initiative* and *DBpedia*[3] whose goal it is to extract structured information from Wikipedia and make it accessible for machines. To describe HTTP requests, we used ontologies published by the World Wide Web Consortium (e.g., w3.org/2011/http). Devices advertise their semantic descriptions by including links to these documents in their responses to HTTP OPTIONS requests, as part of the HTTP Link entity-header. A reasoning service can thus use OPTIONS requests on well-known URLs to obtain the semantic descriptions of the respective devices and use this information for constructing proofs that show if and how a specific goal can be reached.

Reasoning

The reasoner is a software component that generates proofs from (i) the types of available inputs, (ii) a goal and (iii) links to the semantic descriptions of currently available devices (such as the description shown above). The inputs correspond to values that the client already knows about (e.g., the title of a song) and the goal defines what it would like to achieve (e.g., an audio stream in Playing-state and a corresponding stream receiver at the user's location). As reasoning engine, we use the (Prolog-based) Euler Yap Engine (EYE), an open-source backward-forward-backward chaining[4] reasoner. Given inputs and a goal, EYE produces a proof in the N3 format

[3] dublincore.org and dbpedia.org, respectively.

[4] Prolog backward chaining to reach inputs from goal; forward meta-level reasoning to avoid exploring unnecessary branches of the induction tree; backward construction of the proof.

from which the necessary HTTP requests to reach the goal can be extracted. The main reason for selecting the EYE engine is its high efficiency, which allows our system to do reasoning in-between user interaction steps and without having a great impact on the system performance as a whole: to produce a proof that contains the necessary information for a client to configure the devices involved in *Music Escort*, the reasoner requires approximately $5ms$ (excluding network communication).

For the system presented here, we modified an out-of-the-box EYE implementation by creating a Java wrapper that augments the software with a REST interface. We also added a semantic description to the reasoner itself, which specifies that it takes rules and a query as input and returns an object of the semantic type `dbpedia:Formal_proof` as output. This means that agents, when entering our smart environment, are only required to know that they need a service that returns formal proofs to find out how to obtain a proof from the reasoner. Thus, our system does not depend on the concrete implementation of the reasoning engine – any service that returns compatible proofs will do.

Implementation of Reasoning Results
Upon receiving a proof from the reasoner, it is the responsibility of the client application to *implement* the proof by executing all contained requests and thereby modifying its smart environment to reach the desired goal. We decided to have the client implement the proof rather than letting the reasoner do this because the alternative would raise privacy and security issues by requiring the reasoner to have full access to all devices involved in an interaction. Furthermore, we want the client to stay in control of the interaction, especially when a request does not produce the expected result – for instance, a request for a song that does not exist in the database of the stream transmitter would return a HTTP 404 `Not Found` response. How to handle such failures must remain entirely under the control of the client, as it is not the reasoner's responsibility to implement client logic.

In the case of *Music Escort*, to create an audio stream at her location, the user enters a song title and presses the `Play`-Button of the client application (cf. Figure 1). The application fetches the user's current location and uses the management infrastructure to find all devices in the system. It sends their semantic descriptions to the reasoner, together with the goal (a media stream that renders the specified song at the user's current location) and inputs (the song title and location). The reasoner then tries to find a proof that links the inputs to the client's goal. If successful, it transmits the proof to the client which executes the requests contained therein. In this use case, the client sends a total of 23 requests to 4 different devices to configure the system when first initializing an audio stream at his location. These include using the given song title to find the corresponding audio file and then creating a stream transmitter (by means of an HTTP `POST` request). The transmitter must then be configured to stream the file to a stream receiver at the client's location, which must also be created and configured to listen for media streams. Apart from the inputs, none of the values used in these interactions are known to the client in advance. The only information that the client knows from the beginning is the song title, its own location, the desired goal, and the network addresses of the reasoner and the look-up service.

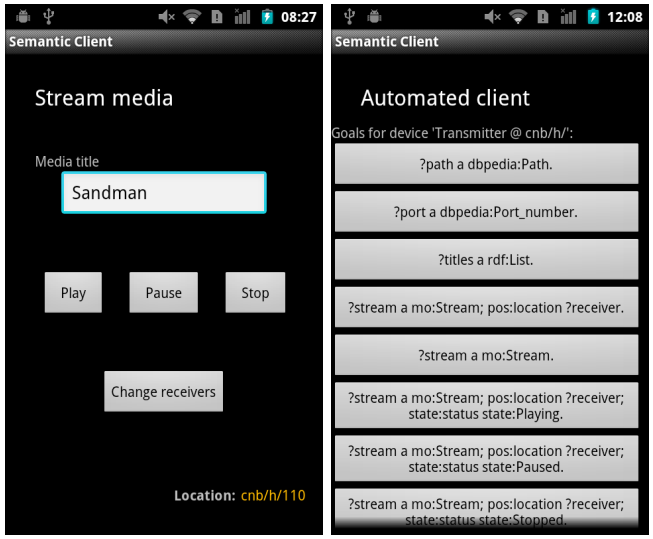

Figure 1: Screenshots of our client application. Left: The user interface for *Music Escort* lets the user specify the name of the song to be played as well as the state of the stream (Playing/ Paused/Stopped). Furthermore, the user's current location is used as input for the reasoning service. Right: The list of goals that are satisfiable by the stream transmitter given the client inputs.

Hypertext-driven Reasoning Multiple Web resources must be created by means of HTTP `POST` requests in the course of implementing a proof and are therefore not available from the beginning of an interaction. It is thus necessary that our system iteratively discovers new resources and re-requests a proof from the reasoner to include information about each newly discovered resource. This behavior is what we refer to as *hypertext-driven reasoning*. The process of requesting new proofs is repeated until a proof has been found for the given goal, or until no more new resources were created.

Invalidation of Local Values Whenever the client application obtains a new value for a variable that is used as an input during reasoning (e.g., upon a location change of the user for *Music Escort*), it must invalidate all variables and resources that logically depend on this variable – in the case of *Music Escort*, this would be the stream receiver that had been created at the user's (now invalidated) location. To find all such entities, the client again makes use of the reasoner. First, it fetches the semantic descriptions of all devices in the environment and *removes* all input variables that occur in these descriptions, *except* the invalidated variable. The modified descriptions are then passed to the reasoner with the invalidated variable as input and every *potentially* invalid variable as goal. If the reasoner can find a proof that links the invalidated variable to any other variable or a created resource, this entity is deemed invalid as well and deleted. Else, it can still be used as input in subsequent interactions. An example of an entity that stays valid in the case of the *Music Escort* application even though a user's location changes is the stream transmitter.

Discussion and Evaluation

Using the described semantic descriptions to facilitate the interaction with Web-enabled smart things was successful. However, reasoning software like EYE has not been created to target ubiquitous computing applications such as the use cases described in this paper. It is thus not fully straightforward to integrate semantic technologies and Web-enabled smart environments.

First, it can be difficult to find suitable ontologies that accurately describe the functionality of a smart thing. However, selecting a suboptimal ontology might lead to conflicts with other described concepts. Our use cases also highlight a trade-off between the ease of creating

semantic descriptions for smart devices and the extensibility of the system. Using simple types such as `dbpedia:Temperature` to describe an input value of the interaction with a smart thermostat can cause problems if there are other services that work with temperature values. This can lead to peculiar behavior of the system as a whole especially when dealing with coupled sensors and actuators, for instance, if a smart thermostat offers a service to *set* the desired temperature and another one to *sense* the current temperature. In this case, the reasoner will conclude that the output of the latter service can be used as input to the former and therefore propose two requests that do not have any combined effect at all. This problem of underspecification can, of course, have major implications when dealing with many more devices that all have their own specific version of generic semantic types like "temperature". In our system, this problem was solved by providing more detailed specifications of the input and output values, and using the semantic markup to express combinations of individual concepts. For instance, the temperature that is *sensed* by the thermostat is combined with its location information, and thus expressed as describing the state of a specific location. In contrast, the API that the thermostat provides to *set* the temperature is described as inducing a state change at a location. We can recommend this approach to solve the problem at hand. However, this requires that the author of the semantic description of a smart thing already anticipates such problems, and similar dependencies between services offered by different devices, to guarantee the extensibility of the system. This, in turn, makes it increasingly harder to create new semantic descriptions and might lead to the implementation of isolated semantic systems that cannot interact with each other at all.

Second, proofs generated by EYE sometimes contain too many requests, or even duplicate requests, which then have to be filtered before they can be implemented by the client application. The reason for this is that the reasoner outputs all *potential* goals that are reachable from a set of inputs rather than focusing on the goal specified by the user. A further disadvantage is that the reasoner requires both, inputs *and* goals, to be defined in each query. This makes it impossible to obtain certain information directly, for instance, to find the necessary inputs for a given goal. Such queries therefore require more requests than would be necessary on a reasoning system that is tailored to the applications at hand, which adds to the complexity of our system.

The *Music Escort* use case exemplifies an application that requires many interactions between devices to configure the system as wished by the user. Especially the iterative discovery of resources leads to a high number of requests

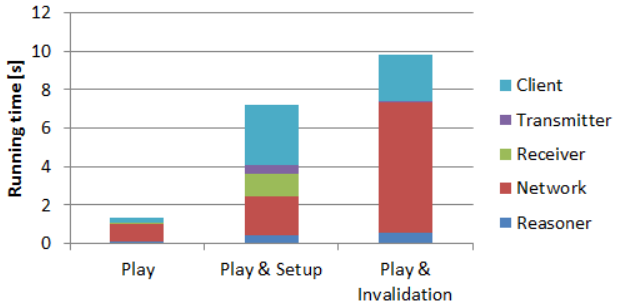

Figure 2: The execution time for different goals. *Play* refers to a transition of the stream transmitter from the *Paused* to the *Playing* state. *Play & Setup* additionally involves setting up the stream transmitter and receiver. *Play & Invalidation* refers to a context change that triggers an invalidation of variables and resources, in this case following a change in the song title.

between the reasoner and other entities in the system. The number of requests necessary to *implement* a proof can furthermore vary between a single request and many, for instance when initializing a new stream receiver. The amount of time required to reach a user's goal therefore also fluctuates heavily, depending on the goal itself and the environment context before the interaction. Figure 2 shows the required amount of time in three different situations. The first is a `Play` request when all services are already set up and the system is paused. The second is again a `Play` request, but this time involving a full setup of the system. The third situation shows how much time it takes to react to a change in the song title, which includes finding out which other variables were invalidated. In the described setting, the considered smart environment consisted of a total of 11 devices including the stream receivers and transmitter, the reasoner, the client application and the management infrastructure. Much time is spent communicating via the network, which is mostly due to the reasoner searching for devices and fetching their semantic descriptions. Our tests with up to 100 simulated smart things showed that the reasoner represents no bottleneck in our current implementation, but that the network latency in such scenarios is too high for meaningful interaction with smart things. In the next implementation of the software, we will therefore focus on reducing the induced network communication.

Conclusions

We have presented a system that brings together semantic technologies and Web-enabled smart environments to facilitate interactions between smart devices and human users. The biggest advantage of our approach is that adding a new device to an already running system is fast and straightforward. New services will seamlessly interact with other services in the system, given that their semantic descriptions are correct and sufficiently detailed. Another advantage is that the use of semantic descriptions greatly reduces the amount of information that smart things and user interface devices must have to interact with other devices in smart environments. For instance, a smartphone application that lets the user control the ambient temperature requires the definition of a single goal, along with the network address of the look-up service and the notion that a semantic reasoner is required. Given these pieces of information, the application can deduce all steps necessary to modify previously unknown environments on behalf of their user.

Tests with up to 100 simulated smart things that offered semantic descriptions revealed that the EYE reasoning engine can deal with much more complexity than shown in our example use cases. Still, we plan to implement more and more complex use case scenarios in the future, which will allow us to explore the scalability of the reasoning service in more real-life situations. We are furthermore confident that we will be able to significantly reduce the time requirements to execute goals in future implementations of our system. To do this, we plan to have the reasoner take a more active part rather than merely responding to client requests. For instance, it could load and buffer semantic device descriptions locally, and also pre-calculate paths and reachable goals for faster reference.

Acknowledgements

This work was supported by the Swiss National Science Foundation under grant number 134631.

References

[1] J. Bellido, R. Alarcon, and C. Sepulveda. Web Linking-Based Protocols for Guiding RESTful M2M

Interaction. In A. Harth and N. Koch, editors, *Current Trends in Web Engineering*, volume 7059 of *Lecture Notes in Computer Science*, pages 74–85. Springer, 2012.

[2] M. Coen, B. Phillips, N. Warshawsky, L. Weisman, S. Peters, and P. Finin. Meeting the Computational Needs of Intelligent Environments: The Metaglue System. In *In Proceedings of the 1st International Workshop on Managing Interactions in Smart Environments (MANSE '99)*, pages 201–212. Springer, 1999.

[3] D. Guinard, V. Trifa, F. Mattern, and E. Wilde. From the Internet of Things to the Web of Things: Resource Oriented Architecture and Best Practices. In D. Uckelmann, M. Harrison, and F. Michahelles, editors, *Architecting the Internet of Things*. Springer, 2011.

[4] S. Mayer, D. Guinard, and V. Trifa. Searching in a Web-based Infrastructure for Smart Things. In *Proceedings of the 3rd International Conference on the Internet of Things (IoT2012)*, Wuxi, China, Oct. 2012.

[5] S. Parastatidis, J. Webber, G. Silveira, and I. S. Robinson. The Role of Hypermedia in Distributed System Development. In *Proceedings of the 1st International Workshop on RESTful Design (WS-REST 2010)*, Raleigh, USA, Apr. 2010. ACM.

[6] C. Pautasso. Composing RESTful services with JOpera. In A. Bergel and J. Fabry, editors, *Proceedings of the 8th International Conference on Software Composition (SC 2009)*, volume 5634 of *Lecture Notes in Computer Science*, pages 142–159. Springer, 2009.

[7] R. Verborgh, T. Steiner, D. Van Deursen, R. Van de Walle, and J. Gabarró Vallés. Efficient Runtime Service Discovery and Consumption with Hyperlinked RESTdesc. In *Proceedings of the 7th International Conference on Next Generation Web Services Practices (NWeSP 2011)*, Salamanca, Spain, Oct. 2011.