Configuration of Smart Environments Made Simple – Combining Visual Modeling with Semantic Metadata and Reasoning

Simon Mayer, Nadine Inhelder, Ruben Verborgh, Rik Van de Walle, and Friedemann Mattern

In: Proceedings of the International Conference on the Internet of Things, 61–66, 2014.

Configuration of Smart Environments Made Simple

Combining Visual Modeling with Semantic Metadata and Reasoning

Simon Mayer*, Nadine Inhelder*, Ruben Verborgh†, Rik Van de Walle†, and Friedemann Mattern*
*Institute for Pervasive Computing, ETH Zurich, Zurich, Switzerland
†Multimedia Lab, Ghent University - iMinds, Ghent, Belgium

***Abstract*—We present an approach that combines semantic metadata and reasoning with a visual modeling tool to enable the goal-driven configuration of smart environments for end users. In contrast to process-driven systems where service mashups are statically defined, this approach makes use of embedded semantic API descriptions to dynamically create mashups that fulfill the user's goal. The main advantage of the presented system is its high degree of flexibility, as service mashups can adapt to dynamic environments and are fault-tolerant with respect to individual services becoming unavailable. To support end users in expressing their goals, we integrated a visual programming tool with our system. This tool enables users to model the desired state of their smart environment graphically and thus hides the technicalities of the underlying semantics and the reasoning. Possible applications of the presented system include the configuration of smart homes to increase individual well-being, and reconfigurations of smart environments, for instance in the industrial automation or healthcare domains.**

I. Motivation

For a long time, researchers in academia and industry alike have been focusing on bringing to life the vision of *smart environments*, and in particular the *smart home* by integrating ideas from pervasive computing and the Internet of Things (IoT) with "classical" home automation systems. Despite this grand effort to put residents in control of a growing number of smart devices in their home and workplace environments and despite the necessary technologies being readily available, smart homes have not yet been widely adopted, which seems to be caused to a large extent by the poor manageability and inflexibility of current home automation solutions [1]–[3].

Indeed, in environments populated by many heterogeneous smart things that can modify the users' environment (e.g., smart thermostats) or provide relevant contextual information (e.g., motion sensors), it is difficult for users to find and utilize services that provide the functionality they require [4]. This is especially true when considering more complex user requirements that involve multiple co-operating devices within so-called "physical mashups," i.e. composite applications that involve sensing or actuation of the physical world [5]. Enabling owners of smart homes to create such mashups has been identified as an important domain in the field of ubiquitous computing research [1], and the easy configuration of smart environments is also gaining importance in the context of industrial automation in "smart factories" [6].

To facilitate the configuration of smart environments for end users, others have applied graphical programming abstractions for home automation (e.g., [7], [8]) that allow users to create mashups that are able to perform complex tasks by integrating functionality across multiple devices or services – an excellent overview of such systems that also considers semantically assisted mashup creation is presented in [9]. These approaches, however, take a process-driven approach where users create *static* links between individual services: once created, they cannot adapt to a dynamic context where devices might become unavailable or new services might appear at runtime.

In this paper, we explore a novel method to facilitate the composition of heterogeneous services for end users. We propose a *goal-driven* approach, meaning that we ask users to state which properties their smart environment should have (e.g., regarding their personal comfort, such as setting the ambient temperature). Given this statement of a user's goal, our system determines whether the goal can be reached given the set of available services, and also infers which user actions (i.e., HTTP requests in this case) are necessary to reach it. The user can then execute these requests and thereby modify his environment to reach the desired goal. Because service mashups are created at runtime from user goals, this approach can handle highly dynamic service environments.

To find out whether the system can provide a mashup that achieves the user's goal, it requires a way of determining what functionality the individual services provide and how to use them. Proposals of tackling this challenge include middleware platforms that are tightly integrated with a user's smart home (e.g., [3]) and API directories such as ProgrammableWeb[1]. Because these, however, do not support the functionality-based discovery of services, we instead suggest to combine semantic information about *what functionality* a service provides with a description of its REST program interface (i.e. of *how* it can be invoked by clients) and expose this data using metadata documents that are linked to the service URLs (see sections II-A and II-B for details about what these descriptions look like). However, since we cannot assume that users are familiar with semantic languages such as RDF, we integrated our system with a visual programming language that allows users to graphically specify the desired state of their smart environment (see Section II-C).

A. Use Case: The Smart Environments Configurator

The concrete use case that we will use as an example to illustrate the components of our system throughout the rest of this paper is an application that automatically modifies smart environments to match end-user preferences. Using a handheld or wearable device such as a tablet computer, smartphone, or smartwatch, users specify the song or radio station to be played in their environment and their comfort temperature, set ambient alarms in the environment, or adjust the lighting to match their preferred levels. This device then negotiates with the environment to adjust the specified parameters to the user's comfort settings and can also provide feedback (see Figure 1). Our goal is to enable such applications to operate successfully in arbitrary environments, i.e. not only in the user's

[1] http://programmableweb.com/

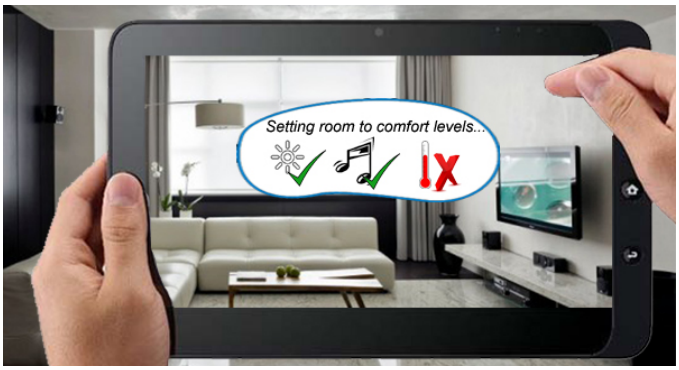

Fig. 1. Feedback of the *Smart Environments Configurator* application after attempting to modify a smart environment to match the user's preferences with respect to lighting, audio playback, and the ambient temperature (mock-up).

private home but also in an office environment, in hotel rooms, cars, and potentially also in public places. In an industrial context, machines or assembly lines could automatically adjust to support their current operator, and semantics could assist the rapid reconfiguration of manufacturing systems [6]. Finally, such a system could be helpful in medical environments, for instance to increase the oxygen saturation to aid asthmatics, or could automatically configure monitor systems to support doctors during clinical diagnostics.

As a simple example to illustrate our approach, we assume that a user wants to set the temperature at his current location to a specific value that is given in degrees Celsius. We further assume that the locally available smart thermostat can take Fahrenheit temperatures as input, which necessitates the usage of a converter and thus can demonstrate our system's ability to chain services for yielding higher-level functionality.

B. System Context

To be able to create service mashups, our system must have access to a means of discovering the *individual* services that are available in the current environment (such as a smart thermostat that is located at a specific location), as well as the semantic metadata they embed. Our system uses a specialized search engine for Web-enabled smart things [10] to accomplish this. However, any search engine or repository that allows to browse or search for service endpoints can be used instead, such as the CoRE Resource Directory[2] when considering resource-constrained devices that support the CoAP protocol, or even industrial standards like UPnP.

Our system also requires access to a semantic reasoner that is able to infer the global structure of a physical mashup from the metadata provided by individual services. While it is possible to use a reasoner that is situated in the "cloud" for this task, we decided to instead use a local implementation in our prototype deployment because of the higher delays when using a remote reasoning instance, and privacy and security considerations: we consider the reasoner a "trusted entity" from the user's perspective, as it requires access to all service metadata. Moreover, the client eventually executes HTTP requests that were proposed by the reasoner.

The final component of our system is the user interface that could, for instance, be a smartphone or tablet computer, or a Web application that allows the user to configure his smart environment. This interface is also responsible for executing the requests that are proposed by the reasoner.

[2]http://datatracker.ietf.org/doc/draft-ietf-core-resource-directory/

II. Usable Semantics for Smart Environments

In the following, we describe the integration of our service discovery and look-up component with services that embed functional semantic metadata in the form of *RESTdesc* descriptions [11]. We also show how a semantic reasoner can use such metadata to infer which mashups can be created in a smart environment and the actions that these can perform for a user (Section II-A). Furthermore, we discuss an important extension to the RESTdesc format that guarantees its usability in the context of smart environments while preserving the logical integrity of the reasoning (Section II-B). Finally, we present how we integrated our approach with *ClickScript*[3], a visual programming tool that enables end users to easily describe the desired state of their smart environment (Section II-C).

A. Semantic Metadata for REST Services

All services that we consider feature Web APIs that are modeled according to the REST principles [12]. While their protocol semantics are thus specified by HTTP, we define their high-level domain semantics (i.e., which function a service can perform) using RESTdesc, a machine-interpretable functional service description format for REST APIs. To enable the automated discovery of the semantic metadata, services advertise their functionality by linking to RESTdesc description documents using the `Link` HTTP header's `describedby` relation[4] as part of their responses to HTTP `GET` and `OPTIONS` requests. This enables any component to look up the metadata for a specific service, given its URL.

RESTdesc descriptions are expressed in Notation3 (N3)[5], an RDF superset that adds support for quantification. As for RDF/XML, the basic unit in N3 is the *triple* that is expressed in the format "`Subject Predicate Object.`" RESTdesc descriptions are regular N3 implications and can be applied by any N3 reasoner without requiring special support. As an example to illustrate the main RESTdesc concepts, the following description of a Celsius-to-Fahrenheit converter communicates how the service can convert a Celsius temperature to the equivalent Fahrenheit temperature:

```
1  {
2    ?tempC a dbpedia:Temperature; ex:hasValue ?cVal;
3        ex:hasUnit "Celsius".
4  }
5  =>
6  {
7    _:tempF a dbpedia:Temperature; ex:hasValue ?fVal;
8        ex:hasUnit "Fahrenheit"; owl:sameAs ?tempC.
9
10   _:request http:methodName "GET";
11       http:requestURI ("converter.net?temp=" ?cVal);
12       http:resp [ http:body ?fVal ].
13 }.
```

At the highest level, a RESTdesc description consists of three parts: preconditions, postconditions, and an HTTP request that realizes the postconditions from the preconditions. In the above example, the preconditions (lines 2 and 3) stipulate that

[3]http://clickscript.ch
[4]http://tools.ietf.org/html/rfc5988
[5]http://www.w3.org/TeamSubmission/n3/

a certain temperature expressed in degrees Celsius exists, and that this temperature has a specific value. The postconditions (lines 7 and 8) warrant that there exists a temperature expressed in degrees Fahrenheit that is the same as the Celsius temperature. Finally, the HTTP request (lines 10 to 12) is a `GET` request to a URL determined by the Celsius value that returns the Fahrenheit value in the response body. This HTTP request is described by the *HTTP in RDF* vocabulary[6]. In this example, the method name, request URI, and the body of the response are specified. When multiple predicate-object pairs are separated using semicolons (e.g., line 2), all of these pairs are associated to the leading subject. Note how N3 adds formulas (between braces {}) that group together triples, variables (starting with a question mark `?`), and implications (triples where the predicate is `=>`). We omitted the necessary `@prefix` declarations that indicate which ontologies are used within the document.

Given a reasoner that has access to the service description of the temperature converter, users can now formulate a goal to ask which Fahrenheit temperature is equivalent to 20°C:

```
1 :myTemp a dbpedia:Temperature; ex:hasValue "20";
2       ex:hasUnit "Celsius".
3 _:convTemp ex:hasValue ?value; ex:hasUnit "Fahrenheit";
4       ex:sameAs :myTemp.
```

From this goal, a reasoner can instantiate the temperature conversion description that will indicate that the answer is given in the response body of an HTTP `GET` request to `http://converter.net?temp=20`. When a reasoner has access to multiple rules, it can *chain the implications* they contain and thereby find out how the client must co-ordinate invocations of different services that can *together* achieve the user goal. For instance, if the user wants to set a temperature of 20°C in an environment that contains a smart thermostat that can only take inputs in degrees Fahrenheit, the reasoner will instruct it to first send an HTTP `GET` to the converter service, unpack the response body, and send the obtained temperature value (in degrees Fahrenheit) to the thermostat. The combination of RESTdesc descriptions with reasoning thus yields a powerful service composition mechanism.

In our concrete system, this process works as follows: To access the individual service descriptions, the reasoner first does a look-up (as described in Section I-B) to find service entry URLs. It accesses these URLs using HTTP `OPTIONS` requests, follows the links that are returned within the `Link` header fields, and parses the N3 documents at these locations, thereby creating a local service catalog. When the user asks for a certain goal, the reasoner is invoked with the service descriptions from that catalog and the user's goal. Using backwards chaining, the reasoner then searches for a path from the current state to the goal state. If successful, it returns the necessary HTTP requests that are executed by the client to realize the goal.

B. Reasoning in Smart Environments

In the previous section, we described how RESTdesc can be used to semantically annotate REST service APIs and how a reasoner can infer whether – and how – a user's goal can be reached by integrating functionality across services. It is however not straightforward to apply RESTdesc in the context of the configuration of smart environments: the main issue when trying to integrate its semantic descriptions with our systems is that RESTdesc – being grounded in first-order logic – is not able to distinguish between mutually exclusive *states*. Therefore, while RESTdesc works very well for describing services that do not induce incompatible states such as the temperature converter in the previous example, already the most basic use cases that involve *stateful* objects cause problems regarding the soundness of the reasoning. As an example, assume that the system has access to the fact that a room has a temperature of 23°C and the user sets the same room to 22°C. This introduces a logical contradiction because the room cannot have two different temperatures at the same moment. We therefore extended RESTdesc by incorporating a mechanism that allows to explicitly describe states, in our case of smart environments and of the devices they contain. Furthermore, we introduced the concept of state transitions to enable the annotation of services that induce state changes. Consider the following RESTdesc description of a smart thermostat:

```
1  {
2    ?newTemp a ex:Temperature; ex:hasValue ?fVal; ex:
         hasUnit "Fahrenheit".
3    ?thermostat a dbpedia:Thermostat; geonames:locatedIn
         ?place.
4    ?state a st:State; log:includes { ?place ex:hasTemp
         ?oldTemp. }.
5  }
6  =>
7  {
8    [ a st:StateChange;
9        st:removed { ?place ex:hasTemp ?oldTemp. };
10       st:added { ?place ex:hasTemp ?newTemp. };
11       st:parent ?state ].
12
13   _:request http:methodName "PUT";
14       http:requestURI (?thermostat);
15       http:reqBody ?fVal.
16 }.
```

From the antecedent of this rule, we can see that an execution of the service requires a temperature value in degrees Fahrenheit (line 2) as well as the presence of a device of type `Thermostat` at a specific location (line 3). The preconditions furthermore contain a state description that specifies that the ambient temperature at the location of the thermostat is `?oldTemp` (line 4). From the consequent of the rule, we learn that an HTTP `PUT` request to the thermostat (lines 13 to 15) will result in a state transition: the new state does not anymore include the `?place` having a temperature of `?oldTemp`, but rather includes the new fact that the temperature at the location of the thermostat is `?newTemp`. The semantics of state transitions are described in a publicly available states ontology[7] that can be looked up by reasoners for successful state handling. As an example, to set the ambient temperature at a specific location to 23°C, a user would formulate the following goal:

```
1 :temp23 a ex:Temperature; ex:hasValue "23"; ex:hasUnit
      "Celsius".
2 ?state a st:State; log:includes { :Office ex:hasTemp
      :temp23. }.
```

In this goal, the user first defines the `temp23` constant that includes the desired temperature value in degrees Celsius. This entity is then used when defining the desired state of the location `Office`. This goal can now be sent to a reasoner that will indicate that the goal state can be reached by first sending an HTTP `GET` request to the converter service to obtain the corresponding Fahrenheit value, and then sending an HTTP

[6] http://www.w3.org/TR/HTTP-in-RDF10/

[7] http://purl.org/restdesc/states#

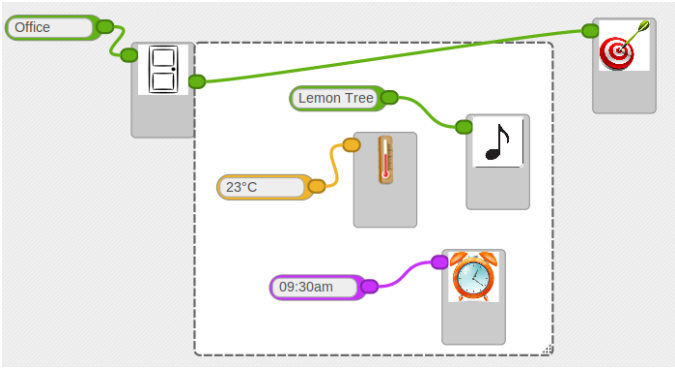

Fig. 2. A ClickScript representation of a user goal that specifies the desired ambient temperature for the room `Office`, selects a song for audio playback, and sets an alarm. The connected "Target" component (top right) will lead to the goal being textually displayed on the screen.

`PUT` request to the URL of the thermostat at the location `Office`. The URL of that thermostat is found at runtime with the help of line 3 of the thermostat's description shown before.

To summarize, we have successfully extended RESTdesc with the concepts of states and state changes. This allows to use the system to describe any service that induces state transitions, and specifically to model smart environments. Based on our approach, it is now possible to create an application that runs on the user's smartphone and lets the user specify the desired state of different named locations with respect to properties such as the ambient temperature or the desired media playback. Since, however, the introduction of states adds a lot of complexity to the goal creation, end users cannot be expected to write valid goal descriptions by themselves: users would not only need to know about the *predicates* to use (e.g., `ex:hasTemp`) but also about the *states ontology* itself, as well as the correct *N3 syntax* to express their goal.

C. Creation of User Goals: Usability Considerations

To enable end users to generate goals themselves, we have integrated our system with ClickScript, a visual programming tool that allows users to graphically model their desired environment [13]. To model the different attributes of a room state, users drag icons that represent the matching components to ClickScript's editing view and connect them to the desired inputs (see Figure 2). Note that these icons do not represent concrete devices or services, but rather stand for different aspects of the state of the environment. When satisfied with the configuration, a user can choose among multiple options of how the created model should be processed by ClickScript: ClickScript can display the goal in N3 notation, invoke the reasoner to get textual feedback about which HTTP requests should be executed to reach the specified room state, or even execute the derived requests itself, thus directly modifying the user's smart environment to match the modeled goal state. Although we did not conduct a usability evaluation, according to our own and others' experience [14], ClickScript is simple to use even for persons without any programming experience.

III. Discussion

In the previous section, we presented the RESTdesc format, as well as an extension to RESTdesc that allows to use its description style to configure smart environments. To facilitate the goal creation for end users, we integrated our system with ClickScript, an intuitive graphical editor.

Others have also employed visual programming abstractions to create mashups that perform higher-level tasks by composing individual services, and even for managing service interactions in smart environments. An excellent overview of different approaches to end-user mashup development is provided in [9] that also presents a "User Assisted Composition" tool that uses semantic technologies to assist users in assembling individual services. Systems such as *homeBLOX* [8] or *Pervasive Maps* [15] also allow users to configure their smart homes via a graphical editor, the former using a metaphor of stacking blocks to create composite services and the latter with the help of pictures of devices instead of icons. Earlier commercial projects in this domain include *Yahoo! Pipes* and the *Ninja*[8] platform for service composition in the home automation domain.

While many of the systems presented in the literature thus are *process-driven*, meaning that end users or developers use them to create mashups manually by connecting or stacking representations of individual services, we demonstrated that by adopting a goal-driven approach, the complexity of the development process as a whole can be reduced: users are only required to model their goals and semantic reasoning is used for the composition itself. Because the mashup creation is done on-the-fly by inferring a path to the user's goal state, this method avoids static linking of services within mashups and therefore is much more flexible than process-driven mashup creation: it automatically adapts to incorporate services that newly appear in a smart environment and circumvents those that become unavailable, thus adding fault-tolerant behavior to physical mashups by finding alternative routes to reach the user's goal in the case of a service outage.

Automatically composing heterogeneous services has been a highly researched topic already before the advent of Web services – [16] represents an excellent overview of previous work. Still, the creation of a generic automatic service composition tool represents an open challenge which is, according to [16], largely due to the lack of expressibility of the planning languages that are used in approaches presented in the literature. Furthermore, many of the proposed automatic composition systems are unable to adapt to dynamic environments – context dynamicity, however, should be considered the default rather than an exception in pervasive computing scenarios and, especially, when targeting applications on mobile devices whose entire smart environment changes when they are on the move. A very interesting system that is in many ways similar to our approach is presented in [17]. The authors use a graph matching algorithm to create a service composition given a user task and introduce a hierarchy of service levels that are offered by devices, where "higher-level" services such as service discovery are in a different service category than less complex entities such as sensors or actuators. In our approach, we do not require to differentiate between different service levels, and our RESTdesc service descriptions hold the necessary API information to invoke appropriate services along with the semantic descriptions of the services themselves. However, the biggest advantage of the approach presented in this paper is that we use a standard semantic language for specifying service

[8] http://ninjablocks.com

TABLE I. DELAYS INCURRED BY PARSING AND REASONING.

#services	1,024	4,096	16,384	65,536	131,072
parsing	276 ms	1,001 ms	3,916 ms	17,127 ms	34,526 ms
reasoning	12 ms	18 ms	107 ms	122 ms	228 ms
total	289 ms	1,019 ms	4,023 ms	17,249 ms	34,754 ms

capabilities and therefore do not require custom matching algorithms but can make use of the existing wealth of research in reasoning technologies, as standard reasoners are used to combine services. This is also what allows our system to potentially make use of a wealth of remotely hosted services and knowledge sources (i.e. third-party ontologies). To illustrate this property, consider again the example that was discussed in Section II-B, where users configured their preferred temperature (in degrees Celsius): to enable interactions with thermostats that take temperatures in degrees Fahrenheit as input, the reasoner is simply required to have access to a (local or remote) service that can convert temperatures from one unit to the other. Many more supporting services could be provided by third parties, published on the Web, and used globally by reasoners in smart environments.

A. Scalability of the Reasoning

Especially when considering third-party services and knowledge sources, a major concern is whether the proposed system remains scalable in light of a multitude of different services being considered by the reasoner. Although other tests with reasoner-based composition already give positive indications [18], we conducted a test to see how fast the reasoning engine we used[9] can compose the required services when the number of available services grows. During the tests, we fixed the composition length to 32 services which is sufficiently high in the context of configuring smart environments, and increased the number of services that are considered during the reasoning to up to 2^{17}. The results indicate that the reasoning time remains well under a few hundred milliseconds on an average consumer computer, and thus within reasonable limits (see Table I). The time required for downloading and parsing the rules, however, does significantly increase, but this problem can be mitigated by caching service descriptions locally at the reasoner.

B. Conflicting Semantic Information

Apart from the required high scalability, incorporating third-party services and ontologies in the reasoning also gives rise to a challenge at the heart of the Semantic Web: the issue of conflicting semantic information. While the Semantic Web and related technologies exist for more than a decade, many researchers are still skeptical about their fitness for real-world applications [19], [20]. In particular, some find it questionable whether these technologies are actually able to achieve the promised interoperability. Indeed, RDF (and, by extension, N3) is not a universal remedy: there will probably always be cases in smart environments that are impossible to solve using a solution based on it. Furthermore, two services that might match in terms of functionality could be using different vocabularies in their RESTdesc descriptions, which would deter a reasoner from deriving that match. Thus, subtle differences in meaning might give rise to false positives or negatives [11]. These are limitations the Semantic Web community is still working on, but we believe that we must take a pragmatic viewpoint with respect to these problems: while many examples can be found that cannot (yet) be solved using the technologies proposed in this paper, many others do already work. By focusing on the cases we *can* solve, we were able to show that semantic technologies can offer a flexible solution – an example of a comparatively complex use case that involves interactions between eleven different services and has been implemented using RESTdesc descriptions is presented in [21].

In the context of smart environments, we thus view semantic technologies as a very flexible form of standardization. Certainly, standards – if honored by all relevant stakeholders – could also accomplish the use cases that we put forward in this paper. While standardization can improve interoperability among compliant components, it however impedes or complicates the integration of elements that were out of scope at the time the standard was designed [6]. Furthermore, semantic technologies offer more freedom with respect to the description of REST endpoints and semantic goals itself. For instance, the RESTdesc descriptions of the services offered by different smart thermostats may differ, or they could use different formats to express the semantics of their offered functionality altogether. However, semantics offer the benefit of allowing decentralized decisions and they can evolve faster than standards can. Using semantics within service descriptions thus represents a lightweight approach to support new services in an evolving way. They also bring with them other benefits, for instance with respect to the customization of user environments: the reasoning process could take into account tailored ontologies that could also capture user preferences, and thus derive service compositions that are even better adapted to individuals.

C. Semantics and Linked Data

A different approach to enable automatic service composition that we want to mention and that is complementary to ours has its roots in the *Linked Data* [22] movement and aims to exploit the REST "Hypermedia as the Engine of Application State" (HATEOAS) constraint to guide service interactions. In contrast to our scheme, this approach does not make use of embedded functional rules or reasoning to construct mashups, but focuses on exposing links between services. The task of actually composing services is left to the client that uses the linking information for determining its next step in an ad-hoc fashion. Links between services are published in so-called *Linkbases*[10], thereby "globalizing" HATEOAS – the concrete way of how these are implemented differs between projects.

One major shortcoming of the Linked Data approach is that such Linkbases for the most part have to be created manually, and therefore are a lot less flexible than our approach when adapting to dynamic contexts. Also, it is not clear how a client would navigate a mashup that is constructed lazily at runtime in a goal-driven fashion, i.e. how it would know about which step to take next without looking further ahead. However, Linked Data principles can be integrated with our approach, and can indeed serve to facilitate the reasoning about service capabilities: if links between services are exposed and can be discovered by our system, the reasoning engine could use that information to narrow down the number of potential paths.

[9]The Euler Yap Engine (EYE), available at http://eulersharp.sourceforge.net/

[10]http://www.w3.org/TR/xlink/

D. Usability Considerations

To make our system usable for end users, we presented an integration with a graphical programming language for goal creation. Potentially, however, end users will never have to formulate goals themselves, as these could be encapsulated in tailored applications on smartphones or other devices, which could also integrate further knowledge about the user's context and his preferences. As an example, it would be perfectly feasible that experts create an application that uses the user's history to infer his favorite songs, automatically creates goals to make his environment play these songs, and executes the corresponding requests without any intervention by the user. Similar applications could be created for office environments, or to support specific use cases in industrial settings.

IV. Conclusion

In this paper, we presented a novel approach that enables end users to configure their smart environment at home, at their workplace, and potentially even at public places. An intuitive graphical editor can be used to create a model of the desired state of the user's environment that is translated into a semantic goal in the Notation3 format. A reasoning engine that has access to functional semantic metadata of services in the user's environment can then infer whether or not it is possible to reach the desired state, and outputs the HTTP requests that are necessary to do so. We implemented and presented a prototypical use case, where the system configures a user's surroundings with respect to the ambient temperature, media playback, and ambient alarms.

Our approach combines concepts from the Semantic Web domain with the configuration of smart environments that contain services offered by physical devices. We created a system that can flexibly combine heterogeneous services to accomplish complex user goals while integrating local and remote services. As it avoids creating statically connected service mashups, it is more flexible than other approaches and has advantages regarding the availability of mashups and their fault tolerance with respect to individual services becoming unavailable. This property is enabled using semantic metadata and reasoning that remain, to a certain extent, brittle. We were however able to hide the complexity and to mitigate the system's fragility by constraining end users' leeway to the specific functionality that is offered by the graphical editor.

In this paper, we did not investigate the requirement to enable multi-user support any further, but believe that majority- or consensus-based systems are conceivable for multi-user environments. In the future, we plan to further explore the potential of the precondition/postcondition-style of RESTdesc descriptions to enable the client to actively guide the reasoning, for instance regarding security or QoS guarantees. A reasoner could, for instance, automatically restrict the inference to mashups that "ensure confidentiality of the transmitted data" or whose execution "costs less than 3$." Finally, we want to experiment with mixed interaction scenarios: here, the reasoner can ask the user for more instructions in situations where multiple paths lead to the same goal. We expect that this approach will increase the robustness of the system and that the additional feedback will help users gain confidence in it.

Acknowledgments: This work was supported by the Swiss National Science Foundation under grant number 341627. The authors thank Jos De Roo for his help with the EYE reasoner.

References

[1] A. J. Brush, B. Lee, R. Mahajan, S. Agarwal, S. Saroiu, and C. Dixon, "Home Automation in the Wild: Challenges and Opportunities," in *Proc. CHI*, ACM, 2011, pp. 2115–2124.

[2] R. Harper, "From Smart Home to Connected Home," in *The Connected Home - The Future of Domestic Life*, R. Harper, Ed., Springer, 2011, pp. 3–18.

[3] C. Dixon, R. Mahajan, S. Agarwal, A. J. Brush, B. Lee, S. Saroiu, and P. Bahl, "An Operating System for the Home," in *Proc. NSDI*, 2012, pp. 337–352.

[4] L. Takayama, C. Pantofaru, D. Robson, B. Soto, and M. Barry, "Making Technology Homey: Finding Sources of Satisfaction and Meaning in Home Automation," in *Proc. UbiComp*, ACM, 2012, pp. 511–520.

[5] D. Guinard, V. Trifa, T. Pham, and O. Liechti, "Towards Physical Mashups in the Web of Things," in *Proc. INSS*, 2009, pp. 196–199.

[6] J. L. M. Lastra and I. M. Delamer, "Semantic Web Services in Factory Automation: Fundamental Insights and Research Roadmap," *IEEE Trans. Ind. Informat.*, vol. 2, no. 1, pp. 1–11, 2006.

[7] J. Humble, A. Crabtree, T. Hemmings, K.-P. Åkesson, B. Koleva, T. Rodden, and P. Hansson, "Playing with the Bits - User-Configuration of Ubiquitous Domestic Environments," in *Proc. UbiComp*, Springer, 2003, pp. 256–263.

[8] M. Rietzler, J. Greim, M. Walch, F. Schaub, B. Wiedersheim, and M. Weber, "HomeBLOX: Introducing Process-Driven Home Automation," in *Adj. Proc. UbiComp*, ACM, 2013, pp. 801–808.

[9] N. Mehandjiev, A. Namoun, F. Lécué, U. Wajid, and G. Kleanthous, "End Users Developing Mashups," in *Web Services Foundations*, Springer, 2014, pp. 709–736.

[10] S. Mayer, D. Guinard, and V. Trifa, "Searching in a Web-based Infrastructure for Smart Things," in *Proc. IoT*, 2012, pp. 119–126.

[11] R. Verborgh, T. Steiner, D. Van Deursen, S. Coppens, J. Gabarró Vallés, and R. Van de Walle, "Functional Descriptions as the Bridge between Hypermedia APIs and the Semantic Web," in *Proc. 3rd Int. Workshop on RESTful Design*, ACM, 2012, pp. 33–40.

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

[13] S. Mayer, N. Inhelder, R. Verborgh, and R. V. de Walle, "User-friendly Configuration of Smart Environments," in *Proc. PerCom*, IEEE, 2014, pp. 163–165.

[14] D. Guinard, C. Floerkemeier, and S. Sarma, "Cloud Computing, REST and Mashups to Simplify RFID Application Development and Deployment," in *Proc. WoT*, 2011.

[15] G. Vanderhulst, K. Luyten, and K. Coninx, "Pervasive Maps: Explore and Interact with Pervasive Environments," in *Proc. PerCom*, IEEE, 2010, pp. 227–234.

[16] Q. Z. Sheng, X. Qiao, A. V. Vasilakos, C. Szabo, S. Bourne, and X. Xu, "Web services composition: A decade's overview," *Information Sciences*, vol. 280, pp. 218–238, 2014.

[17] S. Kalasapur, M. Kumar, and B. Shirazi, "Dynamic Service Composition in Pervasive Computing," *IEEE Trans. Parallel Distrib. Syst.*, vol. 18, no. 7, pp. 907–918, 2007.

[18] R. Verborgh, V. Haerinck, T. Steiner, D. Van Deursen, S. Van Hoecke, J. De Roo, R. Van de Walle, and J. Gabarró Vallés, "Functional Composition of Sensor Web APIs," in *Proc. 5th Int. Workshop on Semantic Sensor Networks*, 2012, pp. 65–80.

[19] N. Shadbolt, T. Berners-Lee, and W. Hall, "The Semantic Web revisited," *IEEE Intell. Syst.*, vol. 21, no. 3, pp. 96–101, Mar. 2006.

[20] C. C. Marshall and F. M. Shipman, "Which Semantic Web?" in *Proc. 14th ACM Conf. on Hypertext and Hypermedia*, ACM, 2003, pp. 57–66.

[21] S. Mayer and G. Basler, "Semantic Metadata to Support Device Interaction in Smart Environments," in *Adj. Proc. UbiComp*, ACM, 2013, pp. 1505–1514.

[22] E. Wilde, "Linked Data and Service Orientation," in *Proc. Service-Oriented Computing*, Springer, 2010, pp. 61–76.