Smart Configuration of Smart Environments

Simon Mayer*‡, Ruben Verborgh†, Matthias Kovatsch*, and Friedemann Mattern*
*Department of Computer Science, ETH Zurich, Zurich, Switzerland †Multimedia Lab, Ghent University – iMinds, Ghent, Belgium ‡Siemens Corporate Technology, Berkeley, USA

***Abstract*—One of the central research challenges in the Internet of Things and Ubiquitous Computing domains is how users can be enabled to "program" their personal and industrial smart environments by combining services that are provided by devices around them. We present a service composition system that enables the goal-driven configuration of smart environments for end users by combining semantic metadata and reasoning with a visual modeling tool. In contrast to process-driven approaches where service mashups are statically defined, we make use of embedded semantic API descriptions to dynamically create mashups that fulfill the user's goal. The main advantage of our 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 users in expressing their goals, we integrated a visual programming tool with our system that allows to model the desired state of a smart environment graphically, thereby hiding the technicalities of the underlying semantics. Possible applications of the presented system include the management of smart homes to increase individual well-being, and reconfigurations of smart environments, for instance in the industrial automation or healthcare domains.**

***Note to Practitioners*—Machine tooling times are an important factor especially when producing small batch sizes. Our approach holds the potential to have manufacturing lines reconfigure themselves at runtime, based on descriptions of the functionality of individual devices. It can even consider properties that influence the process indirectly ("non-functional"), such as the time or monetary cost of a process. We additionally implemented a system that makes these rather complex descriptions understandable for non-specialists. In the article, we describe use cases from the home automation and future manufacturing domains.**

Keywords*—*Internet of Things, Semantic Technologies, Service Composition, Smart Factory, Smart Home, Web of Things

I. Motivation

A main goal of the Internet of Things (IoT) is to empower users by giving them the ability to "program" everyday things and create new public and personal services based on ubiquitous connected devices. It is paramount that users can easily learn *how to use* these services (e.g., fetch data from sensors and trigger actuators) and *how to combine* the capabilities of different devices and services to create advanced composite functions that provide added value. The adoption of Web patterns for the provisioning of services by smart things in the Web of Things (WoT) domain has made it simple for users to use them (this literally is as simple as browsing the Web), and also machines can automatically deduce how a service can be *invoked* because the low-level protocol semantics are specified by the standard Web architecture style.

This article is an extended and updated version of [1].

The focus of this paper is on enabling users to *combine* services that are provided by smart devices in their environment, which remains a heavily researched problem and one of the central challenges for Ubiquitous Computing [2], [3]. For instance, in a smart home, many heterogeneous devices can modify the environment (e.g., smart thermostats) and provide contextual information about the home (e.g., motion sensors). These could also interact to enable more complex applications that involve *multiple cooperating services*: motion sensors and smart thermostats could together infer and apply an optimal heating schedule for the home [4]. The same is true for smart factories, where the easy (re)configuration of manufacturing environments is gaining importance [5] – in that domain, the focus is on supporting operators of industrial plants in managing the increasing dynamicity of manufacturing processes.

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 *semantic statement* of a user's goal, our system determines whether the goal can be reached given the set of available services and infers which user actions (i.e., API requests) 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, this approach can handle highly dynamic situations.

One concrete use case that we use for illustration throughout the rest of this paper is the *Smart Environments Configurator*, an application that automatically modifies smart environments to match end-user preferences. Using a handheld or wearable device such as a tablet computer or smartwatch, users specify the song or radio station to be played in their environment, set ambient alarms, or adjust the lighting and temperature 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 – one application that we developed on top of this configurator is a service that is aware of the user's current location and music preferences and streams his favorite songs directly to media devices in his vicinity. Our goal is to enable such applications to successfully operate in arbitrary environments, that is, not only in the user's private home, but also in an office environment, in hotel rooms, cars, and public places. They could also be helpful in medical environments, for instance to increase the oxygen saturation to aid asthmatics, or to automatically configure monitor systems to support doctors during clinical diagnostics.

A similar idea can be applied in an industrial context, where machines or assembly lines could automatically adjust

to support their current operator, and semantics could assist the rapid reconfiguration of manufacturing systems [5]. To this end, we have implemented a proof of concept to derive execution plans for manufacturing environments: this system is able to move workable items between different manufacturing stations by integrating a semi-professional stationary robot (a Universal Robotics UR5) with a Fischertechnik toy robot. The integration between these very heterogeneous devices – the UR5 is driven by the open-source Robot Operating System (ROS), while the Fischertechnik robot connects to a Siemens PLC – takes place on a semantic level. This provides the foundation for rapidly exchangeable manufacturing devices in industrial environments, which is relevant to reduce tooling times and coordinate maintenance operations.

II. Service Composition for Smart Devices

Many different approaches to (Web) service composition have been proposed over the last decades since "connecting to customers, suppliers, or partners electronically" is considered the top global management issue in the IT domain [6], necessitating tools that allow to compose services globally and across company boundaries. It quickly became obvious that the manual composition of services, where designers use a language such as the Business Process Execution Language (BPEL) *directly* to define a service mashup, is too time-consuming, inflexible, and error-prone, especially when considering the size of the Web and its dynamicity [7]. This led to the development of semi-automatic composition systems that provide support tools to facilitate the mashup design process [8].

A. Semi-automatic Service Composition Systems

Semi-automatic service composition systems proposed in the literature often adopt a *process-driven* paradigm, meaning that users create a composite service by connecting multiple individual, elementary services. Usually, these systems support end users (i.e., process designers) through a formal language or a visual model of the composite service [9], [10], and create executable service specifications from the user input [7].[1]

Many mashup editors from academia and almost all prominent tools used for service composition in business environments feature visual composition interfaces to make them usable by mashup designers without programming skills [7]. Visual programming abstractions have also been applied in the context of facilitating the configuration of smart environments for end users in home automation scenarios: for instance, end users can stack blocks that represent individual services [10] or connect pictures of smart objects to describe the desired composite functionality [12]. Specifically for IoT scenarios, the ClickScript [13] visual programming language has been extended to handle smart things in the WoT [1].

To support users during the service composition process, some of the composition tools that are proposed in the literature can automatically suggest appropriate individual services. While many of these systems provide very limited – often only syntactic – support during service selection, some stand out by providing semantics-based assistance to users when designing composite services (e.g., WebDG [14] and SOA4All [11]). These use ontologies to categorize services and help mashup designers quickly select appropriate services. However, according to [7], even with basic semantic selection support, the main shortcoming of semi-automatic service composition systems is that they only provide very limited support for runtime adaptation of the composite services: the tools help users to create static links between elementary services, which cannot adapt when services become unavailable or new services appear.

B. Approaches to Fully Automatic Service Composition

To enable more dynamic service composition, *fully automatic* composition engines have been proposed [7]. These typically take a set of descriptions of elementary services and a design goal as input and attempt to synthesize a composite service using matching techniques that are based on (often graph-based) planning and scheduling techniques and on different ways of semantically describing the "mini-world" of the individual services. Together with a user-defined planning query, these systems then generate a composition plan for the individual services. Taking a *goal-driven* approach to service composition, this aims at reducing the complexity of the development process as a whole by automating the composition step. To our knowledge, none of the currently proposed fully automated service composition solutions are in use in industry, although major research initiatives are targeting their deployment in this context [15]. In particular, [16] is an interesting solution that uses the GOLOG logic programming language to construct composite services from primitive actions. Our system is similar in that we also employ first-order logic to compose services. However, our approach goes beyond the adaptation of a generic composition template and enables the composition of arbitrary APIs in smart environments that adhere to the Representational State Transfer (REST) architectural style.

Aside from the greater level of automation that goal-driven systems bring, they have the major advantage of enabling the on-the-fly inference of service mashups, and thus avoid static linkage of services. Largely due to lacking interoperability of the different approaches and the isolated application domains of individual systems, automatic service composition still represents an open challenge [7]: many of the currently proposed systems use planning languages such as the Planning Domain Definition Language (PDDL) that are limited regarding their expressibility across different domains. Furthermore, many of the automatic composition systems suffer from the same drawback as the semi-automatic approaches in that they cannot adapt to highly dynamic environments [7]. However, context dynamicity should be considered the default rather than an exception in smart environments, especially when targeting applications on mobile devices whose entire environment changes when they are on the move. This disadvantage is indeed considered one of the main currently open issues regarding future research in the service composition domain: future composition systems should be adaptable to dynamic environments,

[1] This is true for well-known tools in industry (e.g., IBM Business Process Manager and Oracle BPEL Process Manager), as well as open source solutions such as Apache ODE and research prototypes (e.g., SOA4ALL [11]).

ideally supporting self-configuration and automatic optimization relative to the current environment and QoS constraints, as well as relevant security policies [7], [17].

III. Automatic Service Composition

Several of the currently proposed systems for composing Web services are based on the Web Services Description Language (WSDL) and its semantic extensions – however, the severely limited support for REST that WSDL provides [18] represents a missed opportunity since REST itself already defines the low-level protocol semantics of a Web interaction. Semantic descriptions for REST services can therefore focus on specifying the high-level functionality of a service to create a more lightweight automatic composition system. To achieve this, however, it is not sufficient to annotate Web resources with "hints" about the functionality they provide, as proposed with the hRESTS Microformat [19] – annotations should rather contain explicit machine-readable functional service descriptions [20].

In the following, we discuss an approach to fully automatic service composition that allows clients to automatically create and apply service compositions in WoT scenarios with the help of functional semantic annotations while exploiting the REST principles for defining the services' low-level protocol semantics. Our system thus enables the goal-driven configuration of smart environments for end users which means that, instead of having to design a service mashup that achieves their goal, they are merely required to state that goal in a machine-understandable way. Given this goal statement, a reasoning component determines whether the goal can be reached given the set of available services and infers which user actions (i.e., requests involving REST resources) are necessary to reach it.

A. System Overview

To compose services that are provided by devices in smart environments, our system must be able to discover the individual services and their semantic descriptions. In our prototype implementation, we make use of a proprietary Web-based search infrastructure for HTTP and the CoRE Resource Directory [21] for CoAP, a Web protocol for resource-constrained devices [22], [23]. However, our approach is compatible with any system that enables clients to find the URIs of service endpoints, including search engines for the WoT such as Dyser [24] and industry standards such as Universal Plug and Play (UPnP).

To specify the high-level domain semantics of a service, we use *RESTdesc* [20], which we have extended to make it suitable for reasoning about service capabilities in smart environments. Next, our system needs to be able to infer the global structure of a service mashup from information about individual services – for this, we use a *semantic reasoner* that can infer logical consequences from the semantic service specifications. In principle, this reasoner could be hosted anywhere on the Web – in our prototype, however, we use a local instance to reduce delays and to avoid privacy and security implications.

Finally, our system requires a component that interacts with the reasoner and the services in the smart environment on behalf of the user. This interface (e.g., a Web application on a smartphone) is used by people to formulate goals, queries the reasoner for a service mashup that allows to reach the goal, and executes the requests proposed by the reasoner.

```
1  {
2    ?tempC a dbpedia:Temperature;
3           ex:hasValue ?cVal;
4           ex:hasUnit "Celsius".
5  }
6  =>
7  {
8    _:tempF a dbpedia:Temperature;
9            ex:hasValue ?fVal;
10           ex:hasUnit "Fahrenheit";
11           owl:sameAs ?tempC.
12
13   _:request http:methodName "GET";
14             http:requestURI ("http://converter.example.
       com/cel2degf?temp=" ?cVal);
15             http:resp [ http:body ?fVal ].
16 }.
```

Listing 1: A RESTdesc description of a temperature conversion service.

B. Semantic Metadata for REST Services

We consider all smart devices and services to feature Web APIs that are modeled according to the REST principles, so that their protocol semantics are already well-defined, either by HTTP or by CoAP. On top of this, we define their high-level domain semantics (i.e., what function a service provides) using RESTdesc, a machine-interpretable functional service description format for REST APIs. RESTdesc descriptions are expressed in Notation3 (N3) [25], an RDF superset that adds support for quantification. Services expose these descriptions for automated discovery – thus advertising their functionality – through Web Linking [26]. HTTP can carry a link to the RESTdesc document as part of the `Link` header field in responses to HTTP `GET` and `OPTIONS` requests. For resource-constrained devices, this information can be provided out-of-band using the CoRE Link Format [27], an extension to Web Linking that defines an Internet Media Type for Web links.

We illustrate the main concepts of RESTdesc using the example of a service that can convert temperatures given in degrees Celsius to Fahrenheit values, whose semantic description is shown in Listing 1. Later, we will show how a semantic reasoner can automatically create a service mashup that combines the functionality of this converter with a smart thermostat (see Section III-C). At the highest level, a RESTdesc description consists of three parts: preconditions, postconditions, and a REST request that realizes the postconditions from the preconditions. In our example, the preconditions (lines 2 to 4) in the antecedent stipulate that a certain temperature expressed in degrees Celsius exists, and that this temperature has a specific value. The postconditions (lines 8 to 11) in the consequent warrant that there exists a temperature expressed in degrees Fahrenheit that is the same as the Celsius temperature. Finally, the HTTP request (lines 13 to 15) is a `GET` request to a URI determined by the value of the `cVal` variable that returns the Fahrenheit value in the response body. This HTTP request is described by the *HTTP in RDF* vocabulary [28], which provides a semantic way to describe HTTP exchanges and is compatible with CoAP. The description as a whole communicates in a machine-interpretable way how a Celsius temperature can be converted to the equivalent Fahrenheit temperature.

```
_:roomTemp a ex:Temperature; ex:hasValue "20";
                             ex:hasUnit "Celsius".

_:convertedTemp ex:hasValue ?value;
                ex:hasUnit "Fahrenheit".
```

Listing 2: A semantic goal for converting a temperature value.

In more detail, the basic unit in N3 is the *triple* that is expressed in the format "`Subject Predicate Object.`" N3 also has formulas that group together triples (between braces {}), variables (starting with a question mark `?`), and implications (i.e., triples where the predicate is `=>`). When multiple predicate-object pairs are separated using semicolons, all of these pairs are associated to the leading subject. For instance, lines 2 to 4 state that `tempC` is a "Temperature," that its relation to `cVal` is "hasValue," and that its relation to the constant `Celsius` is "hasUnit." For conciseness, we omit the required `@prefix` declarations in our listings that allow the abbreviation of URIs of subjects, predicates, and objects.

Because RESTdesc descriptions are regular N3 implications, they can be applied as inference rules by N3 reasoners without requiring any special support. For each rule it holds that, if the triples in the antecedent can be matched, the triples in the consequent can be concluded. To find out whether a specific goal can be reached in a given context, users can thus use a semantic reasoner that has access to service descriptions such as that of the temperature converter shown above. For instance, a user could ask which Fahrenheit temperature is equivalent to 20°C (Listing 2). Given this goal, a reasoner can instantiate the description of the temperature conversion service, which will indicate that the answer is given by an HTTP `GET` request to the URI `http://converter.example.com/cel2degf?temp=20`.

When a reasoner has access to multiple rules, it can *chain* them and thereby find out how the client must coordinate invocations of different services that 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 only takes inputs in degrees Fahrenheit, the reasoner will generate a plan to first send an HTTP `GET` to the converter service, unpack the response body, and send the obtained Fahrenheit temperature value to the thermostat. The combination of RESTdesc descriptions with reasoning thus yields a powerful service composition mechanism [20].

C. Reasoning in Smart Environments

Unfortunately, it is not straightforward to apply RESTdesc in the context of configuring smart environments: one main issue when trying to integrate its semantic descriptions with our systems and implementing use cases from the field of pervasive computing is that RESTdesc – being grounded in first-order logic – is not able to distinguish between mutually exclusive *states* of components of the system (e.g., of a specific device in the user's smart environment). Therefore, while RESTdesc works very well for describing services that do not induce incompatible states such as the temperature converter in the previous section, already the most basic use cases that involve *stateful* objects cause problems. As a simple example, assume the system has access to the fact that a room has a temperature

```
{
  ?newTemp a ex:Temperature; ex:hasValue ?fVal;
                             ex:hasUnit "Fahrenheit".

  ?thermostat a dbpedia:Thermostat;
              geonames:locatedIn ?place.
}
=>
{
  _:request http:methodName "PUT";
            http:requestURI (?thermostat "?t=" ?fVal).

  [ a st:StateChange;
    st:replaced { ?place ex:hasTemp ?newTemp. };
    st:parent ?state ].
}.
```

Listing 3: A RESTdesc description of a temperature conversion service.

```
:temp23 a ex:Temperature; ex:hasValue "23";
                          ex:hasUnit "Celsius".

?state a st:State;
       log:includes { :Office ex:hasTemp :temp23. }.
```

Listing 4: A RESTdesc description of the temperature goal.

of 20°C. If the user then defines a goal where the same room has a temperature of 23°C, this introduces a logical contradiction because no room can have two different temperatures at any given moment (note that it is not possible to *remove* facts from the knowledge of a first-order logic system).

For this reason, we extended RESTdesc by incorporating a mechanism that allows to describe states of devices within smart environments. We also introduced the concept of *state transitions* to enable the annotation of services that induce state changes, the semantics of which are described in a publicly available states ontology.[2] As an example of a service that makes use of our states definition, consider the description of a smart thermostat in Listing 3. From the antecedent of this rule, we can see that an execution of the service requires a temperature value in degrees Fahrenheit (lines 2 to 4) as well as the presence of a device of type `Thermostat` at a specific location (lines 6 and 7). The consequent of the rule specifies that an HTTP `PUT` request to the thermostat (lines 11 and 12) will result in a state transition (lines 14 to 16): in the new state of the `?place`, the object of the `ex:hasTemp` relation is replaced by `?newTemp`, the new temperature at the location.

To find out how to set the ambient temperature at the location "Office" to 23°C, a user would now formulate the goal shown in Listing 4. In this goal, the user first defines the `temp23` constant that includes the desired temperature value as well as the information that this value is given in degrees Celsius. This entity is then used when defining the desired state of the location "Office." As described in Section III-B, this goal can now be sent to a reasoner, which will indicate that the goal state can be reached by first sending an HTTP `GET` request to the converter service that includes the Celsius value to obtain the corresponding Fahrenheit value, and then sending an HTTP `PUT` request to the URI of the thermostat at the location "Office." Note that because the concrete location is dynamic in the service description in Listing 3, the URI of a correct smart thermostat is given by the `?thermostat` variable and found

[2] See http://purl.org/restdesc/states

at runtime from all available thermostats in the system.

IV. Facilitating the Creation of User Goals

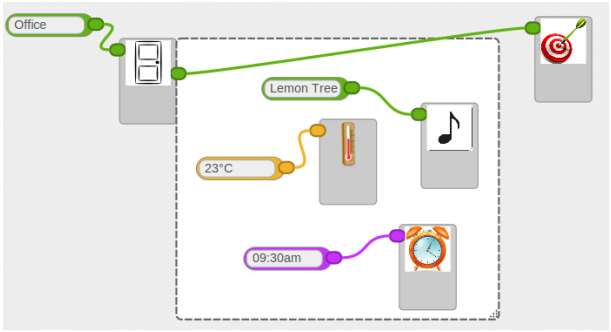
To summarize, we have successfully extended RESTdesc with the concepts of states and state changes. This allows to describe services that induce state transitions, and specifically to model states of smart environments. However, especially because these extensions add a lot of complexity to the goal creation, we cannot expect users to write valid goal descriptions by themselves: they would not only need to know about the correct *N3 syntax*, but also about the *states ontology* and all the *predicates* to use (e.g., `ex:hasTemp`) to express their goal.

To facilitate the goal formulation step for end users, we integrated our system with *ClickScript* [13], a JavaScript-based visual programming tool. We already used ClickScript in earlier projects, when it was extended with the capability of connecting to Web services that run on smart devices using AJAX. For this work, we have further extended the tool to enable its usage for designing semantic goals and for using it as an interface to a semantic reasoning service. Specifically, we have equipped ClickScript with components that represent the different predicates that are useful to describe a smart environment with entities that encapsulate the state of real-world items such as rooms (see Fig. 1(a)). The components available for modeling attributes of a smart environment, such as an abstraction for a *Room* entity or a thermometer that is associated with the `hasTemp` predicate, can be dragged to the editing view of ClickScript to use them within goal definitions.

For instance, to model different desired attributes of a room as in Fig. 1(a), the first step is to create a new room entity and to connect it to the corresponding room identifier (in this case, *Office*). Next, the user configures the desired state of this room by adding components that represent different aspects of this state: the *note* icon represents media playback, the *thermometer* icon stands for the ambient temperature (i.e., the `hasTemp` property in the goal shown in Listing 4), and the *alarm clock* icon is used to model ambient alarms. Finally, the user can infer the correct data types of the input parameters of the components (for instance, the concrete temperature value) from the colors of the component inputs. Thus, users merely have to drag the desired elements into the editing view, connect the matching input and output types, and enter the parameter values.

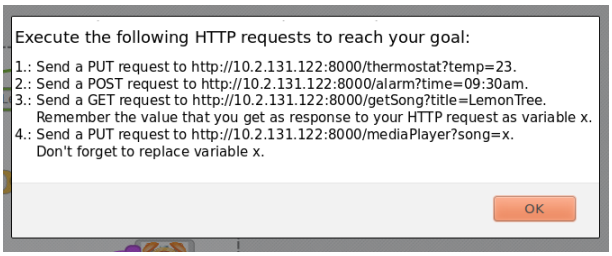
When satisfied with the configuration of the smart environment, the user can choose among multiple options of how the created model should be processed by ClickScript by connecting different components to the output connector of the room entity: users can then decide whether (a) the goal should be displayed, (b) the reasoner should be invoked and the necessary requests displayed in human-readable form, or (c) the requests should be immediately executed by the ClickScript tool itself.

ClickScript does not only parse syntactically correct goals from modeled environments, but also constrains the user to specific ontologies that describe services in smart environments, thus mitigating the problem of conflicting semantic information discussed in Section V. Still, to extend the graphical interface with additional predicates, it is sufficient to specify their names, input types, and appropriate icons to represent the added properties, which requires only a few lines of JavaScript code.



(a)

(b)



(c)

Fig. 1: (a) With our extensions, ClickScript users can create semantic goals graphically. In this example, the user configures the ambient temperature, sets an alarm, and chooses a song for the room "Office." (b) The connected "Target" instructs ClickScript to display the goal that corresponds to the modeled environment, where we omit the definitions of the parameters `temp0`, `song1`, and `time2`. (c) ClickScript can also display the necessary requests to achieve the goal in a human-readable way, using a simple verbalization heuristic.

V. Discussion

To evaluate our proposed approach to automatic service composition in smart environments, we draw on earlier studies that assess service composition systems with respect to a set of qualities these systems should exhibit (see [7]). By its very nature, our system achieves a *high degree of automation* with respect to the selection of individual services: given that these services are annotated appropriately, service selection and composition is done fully automatically. It is also *highly adaptable* with respect to the dynamic availability of specific services in a smart environment, since the ability to bind services dynamically at runtime lies at the core of our system.

#services	1024	4096	16384	65536	131072
parsing	276 ms	1001 ms	3916 ms	17127 ms	34526 ms
reasoning	*12 ms*	*18 ms*	*107 ms*	*122 ms*	*228 ms*
total	289 ms	1019 ms	4023 ms	17249 ms	34754 ms

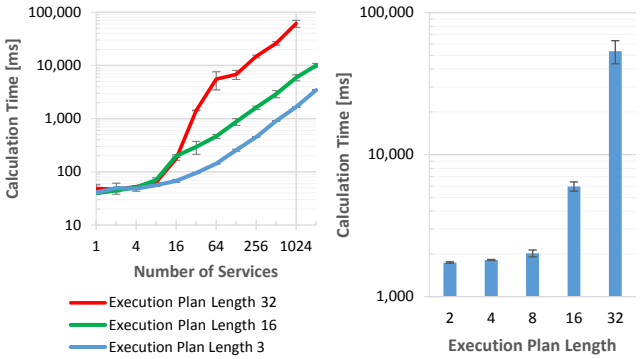
TABLE I: Delays incurred by parsing and reasoning over many services.

Compared to others, our approach allows to obtain execution plans *very rapidly*, and hence clients may even choose to re-query the reasoner in the middle of executing a service mashup for maximal adaptation. Our system also features a *high level of personalization*: user preferences and context characteristics (e.g., device locations) that are available to the reasoner as logical facts are automatically considered during the service composition phase. Since clients execute all requests to individual services themselves, our system can also be considered to be *simple to monitor* by the client. This is closely tied to our system's *reliability*: our approach does not automatically handle exceptional behavior, but the client is explicitly informed about incidents via REST status codes, which are returned by the individual services. The concrete recovery mechanism must, however, be implemented by the client itself. However, if the reason for a failure was a transient fault in the system, for instance related to bad connectivity, it might be sufficient to execute the requests once more, given the idempotent service design that is common with REST. Alternatively, if the reason for the fault was a component of the system that became unavailable, the reasoner should be asked again for a new service execution plan. Both these resolution strategies are generic and do not depend on an explicit fault handling mechanism. In the following, we discuss our semantic service composition approach with respect to the remaining desirable qualities of such systems set forth by [7]: *scalability*, *expressibility*, *correctness*, and *selectability*.

1) Scalability of the Reasoning: One concern with respect to all service composition systems is how they scale with an increasing number of individual services. We explored the scalability of our approach with two experiments: The first demonstrates that it is very lightweight compared to other reasoning-based approaches *in principle* while the second focuses on showing that it is usable to control smart environments in *realistic* contexts.

Being grounded in pure first-order logic, our system scales better than other approaches that employ more heavyweight technologies [20]. To demonstrate this, we conducted an evaluation to see how fast the reasoner we use in our system – the Euler Yet another proof Engine (EYE) [29] – can process service descriptions when the number of available services grows. In this test, that is described in detail in [20], the total composition length was fixed to 32 simple, stateless, services (which is a lot for the context we consider), and the number of individual services that are considered during the reasoning step was increased to up to 2^{17}. Our results (see Table I) show that the *reasoning time* remains under a few hundred milliseconds on an average consumer computer even for very high numbers of considered service descriptions.[3]

[3]The code to run this experiment is available at https://github.com/RubenVerborgh/RESTdesc-Composition-Benchmark



(a) Growing Number of Services (b) 1,024 Services

Fig. 2: Reasoning time under realistic conditions (reproduced from [32]).

The time required for *downloading and parsing* the rules does significantly increase, but this effect can easily be mitigated by caching service descriptions locally at the reasoner.

In contrast to our system, service composition approaches that are based on heavier technologies, such as OWL-S XPlan, require processing time in the order of seconds for planning interactions in settings with under 100 services [30]. Demonstrating that our system scales better than other reasoning-based approaches is, however, insufficient to prove its capability of composing IoT services – some even challenge the ability of current reasoners to accomplish this feat in principle, due to architectural and performance issues [31]. For this reason, we performed more extensive testing of our system under realistic IoT conditions: we simulated an environment with up to 1000 services, which corresponds to about 250 devices, a reasonable assumption for typical smart environments according to several studies (see [32] for a discussion of these estimates). Our tests show that the proof calculation time in such an environment that also includes stateful services (which are natural to physical mashups) is in the order of a few seconds on a laptop computer for execution plan lenghts of up to sixteen service interactions (see Fig. 2). Although the performance of our reasoner is thus lower in IoT settings with stateful services, we conclude that our system can be used to control medium-sized smart environments that contain up to about 250 devices when the goal is reachable through execution plans with around ten requests and only few stateful services are required to reach it. More optimization is necessary for larger settings and applications with real-time constraints – several initial ideas of how this could be accomplished are discussed in [32].

2) Expressibility of RESTdesc: Being based on rules in the N3 format, the RESTdesc language is in principle limited in its expressiveness to implications in monotonic first-order logic. Although its *explicit* expressiveness thus does not rival that of planning languages or business process definition languages, we found that it is suitable for describing services that we encounter in WoT scenarios. The only capability that we added to the language is an explicit state handling mechanism to remove inconsistencies that could arise from state changes in smart environments. With this modification, which we combined with a pragmatic approach of handling temporal dependencies, we

```
{ _:req a sreq:None. } => { _:iReq a sreq:Secure. }.
{ _:req a sreq:None. } => { _:iReq a sreq:HTTP. }.
{ _:req a sreq:Secure. } => { _:iReq a sreq:HTTPS. }.
```

Listing 5: A RESTdesc description of a temperature conversion service.

found the system to be applicable in typical smart environments. We successfully used it to specify services in a home automation context, and in an industrial manufacturing scenario, where we defined capabilities of robotic devices with respect to the transportation of items between manufacturing stations. Others have demonstrated that the RESTdesc language can also be used in the context of multimedia, mathematics, and medical imaging analysis as well as diagnosis assistance [29].

3) Correctness of Service Compositions: In principle, the reasoning component in our approach guarantees the correctness of any composite service it generates. This, however, assumes that the underlying RESTdesc documents clearly and unambiguously capture the functionality of the described services, and that the user goal correctly specifies the desired state of the smart environment using semantic concepts that are compatible to the service descriptions. While we believe that both these challenges can be overcome in limited scenarios with full agreement on the underlying semantic concepts, they give rise to a challenge at the heart of the Semantic Web, especially when third-party services and ontologies are incorporated in the reasoning: the issue of conflicting semantic information. This is perhaps the prime reason for many researchers to remain skeptical regarding the fitness of semantic technologies for real-world applications [33] and to question whether they are actually able to achieve the promised interoperability between services.

In the context of service composition in smart environments, conflicts in the semantic information could lead to situations where services that *should be interoperable* cannot be combined by the system and to situations where services that *should not interact* are utilized within a service mashup. To mitigate these issues in our system, we added the option of visualizing suggested composite applications prior to executing them, but our system does not provide a universal remedy to these issues.

4) Selectability of Service Mashups and Usability Considerations: The property of *selectability* refers to the selection of the most appropriate service among a number of functionally similar or equivalent options, based on non-functional characteristics such as QoS parameters [7]. Ideally, a service composition system would allow users to formulate non-functional preferences with respect to individual services or the service mashup as a whole, either directly within their goals or within accompanying input documents to the reasoner that express these desired characteristics. In fact, the RESTdesc language permits the encoding of such properties within descriptions by extending the *preconditions* of a service description with clauses that describe non-functional characteristics – similar information could also be provided using a separate semantic meta-model of devices and their services.

As an example, we show how our system can enable users to formulate basic security requirements (e.g., *confidentiality* of data exchanged within a mashup) that are considered by the reasoner when composing the service mashup. For illustration, we again use our smart thermostat example, and define basic rules that specify the relationships between different security- and privacy-related concepts in a Web service environment: In Listing 5, the first rule expresses that *Secure* is a "stronger" requirement than *None*. The second and third rules express the relationships between the two security requirements `None` and `Secure`, and the HTTP and HTTPS protocols, respectively.[4] These additional rules allow users to express that they want the communication to happen confidentially, by supplying the fact `userRequirement a sreq:Secure.` in their goal: in this case, the reasoner will only infer the fact that an entity of type `sreq:HTTPS` exists (third rule in Listing 5) which will prevent any service that contains a security specification of `sreq:HTTP` in its precondition from being instantiated by the reasoner. This mechanism also works for composite mashups, e.g., ensuring that *all* communication that happens as part of a mashup happens confidentially, and also applies to other non-functional aspects that can be modeled ontologically.

VI. Conclusions

To facilitate the configuration of smart environments for end users by fully automating the service composition step, we propose a goal-driven approach where users express their needs using a graphical configuration environment. In our system, users define the desired state of their smart environment in the form of a semantic goal that is used by a reasoner to deduce the REST requests necessary to reach that goal. The requests can be executed using HTTP, or using CoAP for resource-constrained devices such as battery-powered sensors or embedded automation components. We are able to satisfy complex demands using only first-order logic, which makes our system flexible yet fast. To overcome the complexity of the goal formulation step for end users, we integrated our system with a graphical editor that enables users to easily create a model of the desired state of their environment. This editor then translates the graphical model into a goal in the N3 format, thereby hiding the complexity of the underlying semantics and mitigating the fragility of manual goal formulation.

The main advantage of using semantic technologies to deduce service mashups is the flexibility of this approach: because the services are combined at runtime, the system can flexibly react to individual services becoming unavailable by finding alternative paths that also serve to reach the user's goal. Furthermore, this allows to derive context-adaptive, personalized mashups by taking into account more information about the system context and users' preferences.

Standards – if honored by all relevant stakeholders – could also accomplish the use cases that we put forward in this paper. However, while standardization can improve interoperability among standard-compliant components, it impedes or complicates the integration of elements that were out of scope at the time the standard was designed. Ontologies have been shown to be more flexible with respect to adding additional concepts to deployed systems [5]. In the context of smart environments,

[4]The same mechanism can be used in constrained environments via CoAPS which uses DTLS, the datagram version of TLS.

we thus consider semantic technologies as a very flexible form of standardization: using semantics within service descriptions represents a lightweight approach to support new services in an evolving way – even when considering their shortcomings with respect to conflicting information.

In the future, we plan to experiment more with multi-user environments and mixed interaction scenarios where the reasoner can interactively ask users for more instructions or to clarify inputs that are required for the reasoning. We expect that this would increase the robustness of the system and that the additional feedback would 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] S. Mayer, N. Inhelder, R. Verborgh, R. V. de Walle, and F. Mattern, "Configuration of Smart Environments Made Simple – Combining Visual Modeling with Semantic Metadata and Reasoning," in *Proceedings of the 4th International Conference on the Internet of Things (IoT)*, IEEE Computer Society, 2014, pp. 61–66.

[2] A. J. Brush, B. Lee, R. Mahajan, S. Agarwal, S. Saroiu, and C. Dixon, "Home Automation in the Wild: Challenges and Opportunities," in *Proceedings of the ACM CHI Conference on Human Factors in Computing Systems (CHI)*, ACM, 2011, pp. 2115–2124.

[3] V. Issarny, N. Georgantas, S. Hachem, A. Zarras, P. Vassiliadis, M. Autili, M. A. Gerosa, and A. B. Hamida, "Service-Oriented Middleware for the Future Internet: State of the Art and Research Directions," *Journal of Internet Services and Applications*, vol. 2, no. 1, pp. 23–45, 2011.

[4] W. Kleiminger, C. Beckel, and S. Santini, "Opportunistic Sensing for Efficient Energy Usage in Private Households," in *Proceedings of the Smart Energy Strategies Conference (SES)*, 2011.

[5] J. L. M. Lastra and I. M. Delamer, "Semantic Web Services in Factory Automation: Fundamental Insights and Research Roadmap," *IEEE Transactions on Industrial Informatics*, vol. 2, no. 1, pp. 1–11, 2006.

[6] C. Peltz, "Web Services Orchestration and Choreography," *IEEE Computer*, vol. 36, no. 10, pp. 46–52, 2003.

[7] 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.

[8] N. Milanovic and M. Malek, "Current Solutions for Web Service Composition," *IEEE Internet Computing*, vol. 8, no. 6, pp. 51–59, 2004.

[9] K. Baïna, B. Benatallah, F. Casati, and F. Toumani, "Model-Driven Web Service Development," in *Proceedings of the 16th International Conference on Advanced Information Systems Engineering (CAiSE)*, 2004, pp. 290–306.

[10] M. Rietzler, J. Greim, M. Walch, F. Schaub, B. Wiedersheim, and M. Weber, "HomeBLOX: Introducing Process-Driven Home Automation," in *Adjunct Proceedings of the 2013 ACM Conference on Pervasive and Ubiquitous Computing (UbiComp)*, ACM, 2013, pp. 801–808.

[11] F. Lécué, Y. Gorronogoitia, R. Gonzalez, M. Radzimski, and M. Villa, "SOA4All: An Innovative Integrated Approach to Services Composition," in *Proceedings of the IEEE International Conference on Web Services (ICWS)*, IEEE Computer Society, 2010, pp. 58–67.

[12] G. Vanderhulst, K. Luyten, and K. Coninx, "Pervasive Maps: Explore and Interact with Pervasive Environments," in *Proceedings of the 8th Annual IEEE International Conference on Pervasive Computing and Communications (PerCom)*, IEEE Computer Society, 2010, pp. 227–234.

[13] L. Naef, "ClickScript: Easy to Use Visual Programming Language," 2011. [Online]. Available: http://clickscript.ch/site/home.php

[14] B. Medjahed and A. Bouguettaya, "A Multilevel Composability Model for Semantic Web Services," *IEEE Transactions on Knowledge and Data Engineering*, vol. 17, no. 7, pp. 954–968, 2005.

[15] P. Stephan, M. Eich, J. Neidig, M. Rosjat, and R. Hengst, "Applying Digital Product Memories in Industrial Production," in *SemProM: Foundations of Semantic Product Memories for the Internet of Things*, W. Wahlster, Ed., Springer, 2013, pp. 283–304.

[16] S. Sohrabi, N. Prokoshyna, and S. A. McIlraith, "Web Service Composition via the Customization of Golog Programs with User Preferences," in *Conceptual Modeling: Foundations and Applications*, ser. Lecture Notes in Computer Science, A. T. Borgida, V. K. Chaudhri, P. Giorgini, and E. S. Yu, Eds., Springer, 2009, vol. 5600, pp. 319–334.

[17] A. Charfi, T. Dinkelaker, and M. Mezini, "A Plug-in Architecture for Self-Adaptive Web Service Compositions," in *IEEE International Conference on Web Services (ICWS)*, IEEE Computer Society, 2009, pp. 35–42.

[18] H. Zhao and P. Doshi, "Towards Automated RESTful Web Service Composition," in *Proceedings of the IEEE International Conference on Web Services (ICWS)*, IEEE Computer Society, 2009, pp. 189–196.

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

[20] R. Verborgh, V. Haerinck, T. Steiner, D. Van Deursen, S. Van Hoecke, J. De Roo, R. Van de Walle, and J. G. Vallés, "Functional Composition of Sensor Web APIs," in *Proceedings of the 5th International Workshop on Semantic Sensor Networks (SSN)*, ser. CEUR Workshop Proceedings, no. 904, 2012, pp. 65–80.

[21] Z. Shelby, "CoRE Resource Directory," Internet-Draft, 2014. [Online]. Available: http://tools.ietf.org/html/draft-ietf-core-resource-directory-02

[22] M. Kovatsch, "CoAP for the Web of Things: From Tiny Resource-constrained Devices to the Web Browser," in *Adjunct Proceedings of the 2013 ACM International Joint Conference on Pervasive and Ubiquitous Computing (UbiComp)*, ACM, 2013, pp. 1495–1504.

[23] Z. Shelby, K. Hartke, and C. Bormann, "The Constrained Application Protocol (CoAP)," RFC 7252, 2014. [Online]. Available: http://tools.ietf.org/html/rfc7252

[24] B. Ostermaier, K. Römer, F. Mattern, M. Fahrmair, and W. Kellerer, "A Real-Time Search Engine for the Web of Things," in *Proceedings of the 2nd IEEE International Conference on the Internet of Things (IoT)*, IEEE Computer Society, 2010.

[25] T. Berners-Lee and D. Connolly, "Notation3: A readable RDF syntax," 2011. [Online]. Available: http://www.w3.org/TeamSubmission/n3/

[26] M. Nottingham, "Web Linking," RFC 5988, 2010. [Online]. Available: http://tools.ietf.org/html/rfc5988

[27] Z. Shelby, "Constrained RESTful Environments (CoRE) Link Format," RFC 6690, 2012. [Online]. Available: http://tools.ietf.org/html/rfc6690

[28] World Wide Web Consortium, "HTTP Vocabulary in RDF 1.0," 2011. [Online]. Available: http://www.w3.org/TR/HTTP-in-RDF10/

[29] R. Verborgh and J. De Roo, "Drawing Conclusions from Linked Data on the Web," *IEEE Software*, vol. 32, no. 5, pp. 23–27, 2015.

[30] M. Klusch and A. Gerber, "Fast Composition Planning of OWL-S Services and Application," in *Proceedings of the 4th IEEE European Conference on Web Services (ECOWS)*, IEEE Computer Society, 2006, pp. 181–190.

[31] S. De, B. Christophe, and K. Moessner, "Semantic enablers for dynamic digital-physical object associations in a federated node architecture for the Internet of Things," *Ad Hoc Networks*, vol. 18, pp. 102–120, 2014.

[32] M. Kovatsch, Y. N. Hassan, and S. Mayer, "Practical Semantics for the Internet of Things," in *Proceedings of the 5th International Conference on the Internet of Things (IoT)*, IEEE Computer Society, 2015, pp. 54–61.

[33] N. Shadbolt, T. Berners-Lee, and W. Hall, "The Semantic Web Revisited," *IEEE Intelligent Systems*, vol. 21, no. 3, pp. 96–101, 2006.