# Practical Semantics for the Internet of Things
## Physical States, Device Mashups, and Open Questions

Matthias Kovatsch
Department of Computer Science
ETH Zurich, Switzerland
Email: kovatsch@inf.ethz.ch

Yassin N. Hassan
Department of Computer Science
ETH Zurich, Switzerland
Email: yhassan@student.ethz.ch

Simon Mayer
Siemens Corporate Technology
Berkeley, USA
Email: simonmayer@siemens.com

***Abstract*—The Internet of Things (IoT) envisions cross-domain applications that combine digital services with services provided by resource-constrained embedded devices that connect to the physical world. Such smart environments can comprise a large number of devices from various different vendors. This requires a high degree of decoupling and neither devices nor user agents can rely on a priori knowledge of service APIs. Semantic service descriptions are applicable to heterogeneous application domains due to their high level of abstraction and can enable automatic service composition. This paper shows how the RESTdesc description format and semantic reasoning can be applied to create Web-like mashups in smart environments. Our approach supports highly dynamic environments with resource-constrained IoT devices where services can become unavailable due to device mobility, limited energy, or network disruptions. The concepts are backed by a concrete system architecture whose implementation is publicly available. It is used to evaluate the semantics-based approach in a realistic IoT-related scenario. The results show that current reasoners are able to produce medium-sized IoT mashups, but struggle with state space explosion when physical states become part of the proofing process.**

## I. Introduction

The Internet of Things (IoT) is expected to turn our surroundings into smart environments by combining sensing and actuation with digital services. For this, tiny communicating computing devices are embedded into everyday objects. They usually have limited processing power (megahertz MCUs), memory (kilobytes of RAM and Flash), and communication bandwidth (low-power wireless transceivers). Furthermore, they will often miss proper user interfaces, in particular input devices and displays, which makes configuration cumbersome. The overall challenge, however, is service composition in smart environments due to the high number of IoT devices from various vendors and different application domains. Because of this heterogeneity, classic approaches with device and service IDs or API documentation and manual composition do not scale. Service descriptions need to be abstract enough to cover multiple application domains, but still allow for automatic service composition. Moreover, smart environments can be highly dynamic with devices joining, moving around, and temporally failing. Thus, service providers and consumers must be loosely coupled and able to evolve with a changing environment [17]. To remedy this situation, we present new concepts to facilitate the self-configuration of IoT applications in such environments and provide a quantitative evaluation.

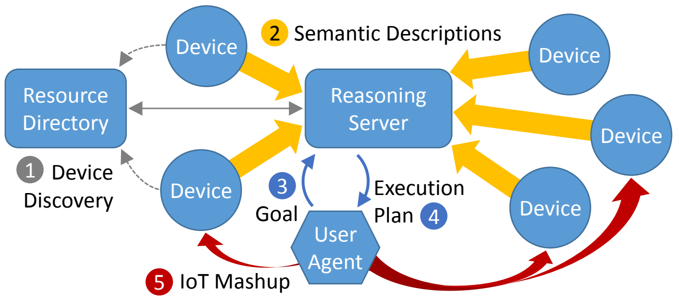


Fig. 1. By providing a goal, user agents or machine clients can use a semantic reasoner to automatically create Web-like mashups among resource-constrained IoT devices and services.

### A. The Web of Things

Our approach follows the Web of Things initiative by applying patterns from the World Wide Web to realize the application layer of the IoT [6]. We expect all IoT devices and services to provide RESTful APIs [4] that can be combined into Web-like IoT *mashups*. REST enables loose service coupling and defines the low-level protocol semantics through URIs, standard methods, and well-defined representations of the resource state that is exchanged. This enables wide interoperability among different IoT devices—even across different application domains. Yet the WoT is missing the hypermedia controls (i.e., representation formats and link relation types) that would allow machines to learn the high-level application semantics. Thus, IoT mashups are usually defined manually using a scripting language [10] or visual tools [2].

The Semantic Web community has been working on approaches to make the Web machine-readable [1], [12]. The *Resource Description Framework (RDF)* defines a metadata model to describe the *data semantics* independently from the application domain. For this, the model uses three data types: *Resources* are the central entities in RDF and are identified by Internationalized Resource Identifiers (IRIs)—a superset of URIs that can also include abstract resources that are not part of the Web. *Properties* describe specific aspects of a Resource such as an attribute or the relationship between two resources. *Statements* are triples that consist of a Resource ("subject"), a Property ("predicate"), and the value of the Property ("object"), where this value can be another Resource or a literal.

```
{  # Antecedent: set of preconditions for the implication
  ?degreesFahrenheit a dbpedia:Temperature;
      ex:hasValue ?fahrenheitValue;
      ex:hasUnit "Fahrenheit".
}
=> # Implication
{  # Conclusion: set of postcondition Statements
  ?degreesCelsius a dbpedia:Temperature;
    ex:hasValue ?celsiusValue;
    ex:hasUnit "Celsius";
    ex:derivedFrom ?degreesFahrenheit.

  # Request description added by RESTdesc
  _:request http:methodName "POST";
    http:requestURI ("http://conv.example.com/degf2cel");
    http:reqBody ?fahrenheitValue;
    http:resp ?celsiusValue.
}.
```

Listing 1. RESTdesc description of a Fahrenheit to Celsius conversion service. Variables that are substituted with matching Resources from the knowledge base start with a **?**. The prefix definitions for the Resources are omitted.

## B. Reasoning and RESTdesc

Given a set of RDF triples, machines can only understand Statements that are explicitly given. The *Notation3 (N3)* syntax[1] adds support for logical formulas and quantification. With this expressiveness, a *semantic reasoner* (a.k.a. proof engine) can derive new Statements through first-order logic (see Lst. 1): If the preconditions in the *antecedent* are true for a specific substitution of the variables, then the postconditions in the *conclusion* also apply and Statements with the same substitutions may be added to the knowledge base.

The *RESTdesc* format [18] leverages these inference rules to describe the *functional semantics* of RESTful APIs. To do this, it adds additional *blank nodes* (_: namespace) that describe the REST requests required to realize the postconditions in the conclusion of a rule. The request arguments (e.g., URI or request body) are configurable through the substitutions given in the antecedent. The response usually provides a literal that is substituted in one of the postconditions (e.g., **?celsiusValue** in Lst. 1).

By combining RESTdesc and semantic reasoning, the reachability of specific Statements can be proven. Hence, users can define a *goal* they want to achieve as Statements: if the goal can indeed be reached, the reasoner outputs all necessary requests to reach it. There can be dependencies between requests when a precondition includes a variable that is quantified in the conclusion of a different rule. The resulting ordered list of requests including their parameterization is called *execution plan*. It tells a client how to use RESTful APIs to achieve a certain goal.

## C. The Constrained Application Protocol (CoAP)

Resource-constrained IoT devices require lightweight protocols that fulfill the requirements of limited memory and processing power as well as low-power networking. Based on the success of low-power IP for the IoT, the *Internet Engineering Task Force (IETF)* has standardized a new Web protocol for constrained RESTful environments: the *Constrained Application Protocol (CoAP)* [15]. Following the REST architectural style, CoAP was designed from scratch with the above-mentioned requirements in mind. It uses URIs to address resources, enables interaction through the uniform methods GET, POST, PUT, and DELETE, and uses standard Internet Media Types to transfer data (i.e., resource representations and action results). Additional features allow to *observe* resources [7] (i.e., servers can push notifications when the resource state changes) and to use IP multicast for group communication.

## D. Contributions

Due to the similarities between CoAP and HTTP, we successfully transferred and extended the results from current research in semantics-based service composition for unconstrained WoT devices [13] to constrained RESTful environments. In particular, we contribute:

- Extensions for RESTdesc to better handle physical states (Sec. II-A) and IoT mashups (Sec. II-B), and the concept of *open questions* that enable interactive feedback from the user or other devices (Sec. II-C).
- A working CoAP-based system that applies this technology to constrained environments (Sec. III).
- The *Semantic IDE*, which fosters the study and development of semantics for RESTful IoT devices (Sec. III-D).
- An IoT-related evaluation of the semantics-based approach to automatic service composition (Sec. IV).

Our implementations are publicly available on GitHub.[2]

## II. Practical Semantics for the IoT

RDF and RESTdesc have been developed for machine-readable linked data and classic Web services. These deal with knowledge bases of mainly static information. By contrast, IoT systems must be able to handle frequently changing information and services on devices in the physical world whose availability is much less stable. In this section, we present the current body of related work together with our extensions to practically apply semantic reasoning in the IoT.

## A. Physical States

IoT devices provide a lot of information that is primarily about real-time states of the physical world. This includes sensor readings, but also the state of actuators or the devices themselves. The challenge here is not the frequency with which the states can change, since the reasoner is always invoked with a momentary snapshot of the environment. Creating an execution plan that changes the environment requires that different Statements can be true during the same reasoning process, that is, before and after a state change. In first-order logic, however, reasoners cannot invalidate once proven facts, and hence are not able to produce a proof for execution plans that include state changes [5]. To remedy this situation, Mayer et al. introduced an ontology that enables RESTdesc to handle

[1]http://www.w3.org/TeamSubmission/n3/

[2]https://github.com/mkovatsc/iot-semantics

state transitions [13]. The value of a state Resource is modeled as a container, a *parent state*, in which transitions can be chained to describe state changes during reasoning. The last element of this chain contains the currently valid Statements.

The disadvantage of the available ontology is that each transition must fully match the previous Statements, and hence must include the 'old' environment state to be chained. This adds complexity to the goal definition and in some cases the values can even be undefined, making it impossible to define a valid transition. Thus, we extended the state ontology by introducing a new state transition Property called `replaced`. It defines a list of Statements that can always be added to the parent state, while all previous Statements with the same subject and predicate are invalidated (i.e., all related Statements are replaced with the new ones). As an example, the RESTdesc description in Lst. 2 replaces any "old" temperature values from the parent state with the `?new` value; the given PUT request applies this transition to the physical environment.

```
{
  ?new a dbpedia:Temperature;
       ex:hasValue ?tempValue;
       ex:hasUnit "Celcius".
  # there is a modifiable state
  ?state a st:State.
} => {
  # transition replacing the temperature for the location
  [ a st:StateChange;
    st:replaced { ?location ex:hasTemperature ?new. };
    st:parent ?state ].

  _:request http:methodName "PUT";
    http:requestURI ("http://ac.example.com/setpoint");
    http:reqBody ?tempValue.
}.
```

Listing 2. State transition that changes the current temperature using our extended state ontology.

Statements about physical states can also be used as preconditions, which helps to include the current device state into the reasoning process. Here, the frequent changes in the knowledge base can pose a problem, since the reasoner always requires a fresh snapshot to produce valid execution plans. We propose to split the semantic description of a device into a static description of its services and a dynamic description of its current state. Furthermore, we use the CoAP observe mechanism to keep the snapshot at the reasoner in-sync with all device states. Sec. III describes this feature in more detail based on our system implementation.

## *B. Device Mashups*

Real-world settings such as smart home and office environments usually have multiple devices that provide the same functionality but in different contexts (e.g., multiple ambient lights or air conditioners in different rooms). Their semantic descriptions can be identical and defined Resources could overlap. However, devices need to provide individual definitions to model the different context. Therefore, we introduce a device-local namespace, which can be used to define device-specific Statements. The `local:` namespace is similar to the empty namespace, which is the local namespace of a document, but it spans all description documents of the same device.

We use an extension at the reasoner to manage the device-local namespaces. It uses the dynamic network addresses of devices to guarantee uniqueness. This also allows devices to use relative URIs in the request definitions. The semantic documents can then be defined as a static file (i.e., served from ROM), simplifying the implementation for resource-constrained devices. The absolute URI that includes the dynamic network address is generated at the reasoner. It provides a statement to substitute a device-local `?uri` variable, the *base URI*, which is then concatenated with the relative URI of the request definitions (e.g., `_:request http:requestURI (?uri "/set/temperature")`).

The HTTP ontology used in RESTdesc is a basic model for requests and responses. Since CoAP is also an implementation of REST, we were able to apply the RESTdesc request statements unchanged. The base URI (`?uri` variable) simply uses the `coap` scheme to identify CoAP-based services (e.g., `coap://ac2.example.com`). This allows to apply the semantic approach to resource-constrained IoT devices.

Instead of having the reasoner perform the requests in the execution plan, the plan is transmitted to the client, which has to execute it itself. This allows for a fine-grained security model: Instead of a central entity that requires access to all devices in an environment, the client that defines the goal requires the right authorization to manipulate the environment. This can be enforced through the `coaps` scheme using DTLS with client-side authentication and the upcoming mechanisms of the *Authentication and authorization in Constrained Environments (ACE)* working group within the IETF.

## *C. Open Questions*

To simplify the commissioning process in smart environments, we introduce a uniform device configuration interface. This allows each device to define a series of *open questions* as part of its (static) semantic description. The open questions can either be answered directly by the end user or by combining information from other devices using inference rules.

A semantic question is defined by creating an RDF Resource of the type `:question`. If the answer to the question differs for each instance of the device, the question must be defined in a device-local namespace. Each `:question` Resource must define a human-readable version of the question using the predicate `:text`, which can displayed to the end user (see Lst. 3). In addition, each question must define an answer type using the predicate `:replyType`. This is used to provide the end user with suggested answers based on answers to other questions with the same type. For example, if a question defines the answer type `:location`, a list of previously used locations to choose from can be provided to the end user.

```
local:devicelocation a :question;
  :text "Where is the air conditioner located?";
  :replyType :location.
```

Listing 3. Defining a question.

Questions can then be answered by POSTing corresponding Statements about an `:answer` Resource to the REST API of the reasoner. The triples must contain the Property `:answers` with the corresponding question as value. An answer is typed according to the `:replyType` of the question and contains further Statements to define the answer such as the location `:name` in Lst. 4.

```
config:ans1 a :answer;
  :answers device_1:devicelocation;
  a :location;
  :name "Living Room".
```

Listing 4. Statements that answer the question from Lst.3: here, the name of the requested location is given. Depending on the definition of :location, providing other information (e.g., coordinates) is also possible.

Each question must come along with an inference rule that allows to properly match its answer within the reasoning process. In the case of the location example, a rule defining the `:locatedAt` Property for the device Resource with the answer as value can then be used to determine the room controlled by the air conditioner in question (see Lst. 5).

```
{
  local:devicelocation :hasAnswer ?a.
} => {
  local:airconditioner :locatedAt ?a.
}.
```

Listing 5. Derive the `:locatedAt` Property from an answer.

In some cases, the knowledge base might contain Statements that define these Properties directly. For instance, there might be a localization system that can identify devices automatically and defines the triple `local:airconditioner :locatedAt ?a`. To mark the related questions as answered, we must also define a rule that derives the `:hasAnswer` Statement from this precondition (cf. Lst. 5 with swapped antecedent and conclusion).

To retrieve unanswered questions for dynamic answering, we introduce the semantic type `:openquestion`. It is assigned to all questions that have an empty answer set. This can be achieved using the logic rule `eye:findall`, which is built into the reasoner. In the precondition of the rule, we match the subject `?q` with the semantic type `:question` and then find all `?a` that satisfy the pattern `?q :hasAnswer ?a`. If the substitution of `eye:findall` returns an empty set `()`, we can conclude that the question is unanswered and assign the semantic type `:openquestion` to the subject `?q`.

```
@prefix eye: <http://eulersharp.sourceforge.net/2003/03
    swap/log-rules#>.
{
  ?q a :question.
  ?SCOPE eye:findall ( ?ANY
    { ?q :hasAnswer ?ANY }
    ()
  ).
} => {
  ?q a :openquestion.
}.
```

Listing 6. This inference rule marks all unanswered questions with the type `:openquestion`

The semantic reasoner can retrieve all open questions for a client similar to an execution plan query. For this, we again use the `eye:findall` rule, but also include all the possible answers that match the reply type. This way, the open questions query returns all questions together with a list of answers from which a human user can choose. Machine clients usually rely on the `:replyType` directly.

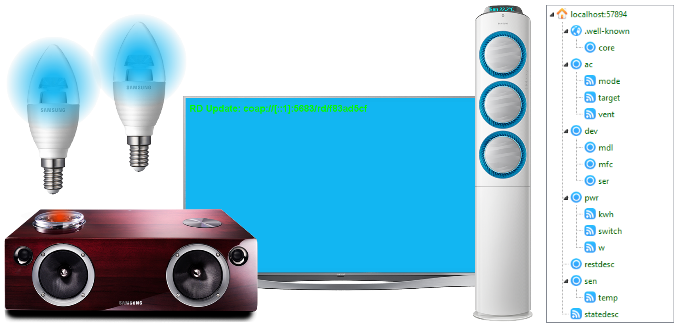


Fig. 2. We use emulated IoT based on Californium to focus on the design of the APIs and their semantic descriptions. Besides simple, constrained devices such as lightbulbs, we have a couple of more complex devices such as the air conditioner, whose RESTful API is shown on the right.

## III. System Architecture for IoT Semantics

In this work, we focus on the design of RESTful APIs for IoT devices and the modeling of their semantic descriptions. We use a set of emulated CoAP devices that provide a good mixture of simple and complex APIs. We do not follow a specific API style to demonstrate the semantic interoperability. Rather, each device simulates internal processes such as power consumption or sensor readings and visualizes its current state through a graphical representation as shown in Fig. 2. The devices are implemented using *Californium* [11], which allows for rapid prototyping and easy deployment based on Java. Based on our experience with actual IoT hardware, we can confirm that the results learned for the emulated devices can easily be transferred to constrained implementations such as *Erbium* [9]. The overall architecture is depicted in Fig. 1.

### A. CoRE Resource Directory (RD)

CoAP-based systems usually use a CoRE resource directory (RD) for service discovery [16]. We use the default Cf-RD provided in the *californium.tools* repository[3] to be aware of all devices or services in our smart environment. Each device registers with the RD by POSTing a list of their provided Web resources as CoRE Link Format [14] to a standard interface. The Link Format contains Web linking attributes for each resource, which allows to identify specific resources such as the semantic descriptions and the dynamic state descriptions (Fig. 2 `/restdesc` and `/statedesc`, respectively). An update mechanism, where devices regularly need to send POST requests to a registration handle URI at the RD, ensures freshness of the discovered services.

### B. Reasoning Server

In this work, we use the *Euler Yet another proof Engine (EYE)*.[4] EYE is a semi-backward reasoning engine enhanced with Euler path detection. As stated earlier, it is based on first-order logic. We encapsulated the Prolog-based EYE in a Californium-based server that provides several features to use the reasoner within smart environments.

[3]https://github.com/eclipse/californium.tools
[4]http://eulersharp.sourceforge.net/2003/03swap/

*1) Notation3 Parser:* All interaction with the EYE reasoner is based on N3 documents. While previous projects used a combination of regular expressions and string matching strategies to parse N3 files containing the reasoner proof, we implemented a new parser covering the entire Notation3 grammar. It is based on ANTLR v4[5] and extends the Turtle grammar created by Alejandro Medrano,[6] which is a subset of Notation3. We also provide an abstract syntax tree (AST) visitor to modify N3 documents and store them back as files.

*2) Knowledge Base:* The reasoning server caches all semantic descriptions of the smart environment locally. For that, it periodically checks the RD, retrieves the descriptions of new devices, and removes those of devices that disappeared. Furthermore, it observes all dynamic descriptions of the current device states (e.g., `/statedesc` in Fig. 2). This way, the reasoning server always has an up-to-date snapshot of the environment to provide valid execution plans. Finally, the knowledge base automatically manages the device-local namespaces and the dynamic network address of each device to provide absolute URIs in the execution plans.

*3) Query Interface:* The query interface allows clients to access the knowledge base, for instance to retrieve the list of open questions or to query the location of a specific device. Queries are defined like goals, that is, a list of preconditions to define the resources the client is looking for and postconditions to define the output (see Lst. 7). The semantic reasoner is used to process the query given the knowledge base of the server and the resulting proof is parsed to return an N3 document that contains the Statements specified in the postconditions. For now, interacting with the knowledge base requires clients (these could also be IoT devices) to be able to parse N3.

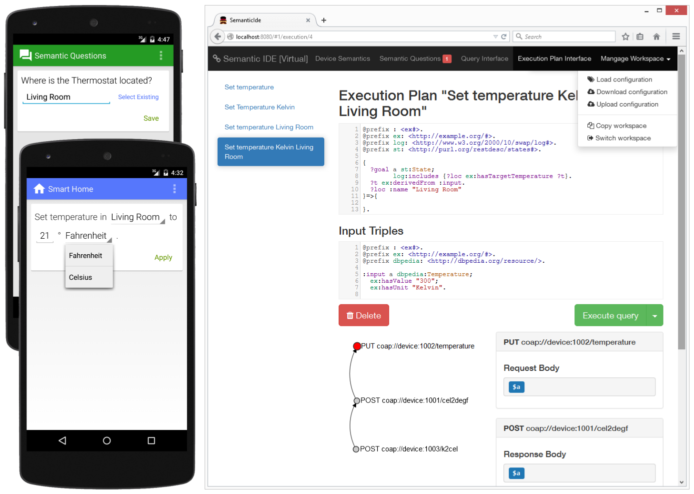
```
{ # look for devices in the "Living Room"
  ?device :locatedAt ?location.
  ?location :name "Living Room".
  ?device :name ?deviceName.
} => { # output the device names
  ?device :name ?deviceName.
}.
```

Listing 7. Query that retrieves all the devices in the living room and a triple containing the device resource and name.

*4) Execution Plan Interface:* Clients use this interface to request an execution plan for their user-defined goal. A goal definition is an implication (=>) that has the desired Statements as preconditions and an empty consequent, as the preconditions simply need to hold true without any specific conditions. In addition, the reasoner requires the definitions of the goal values as input (e.g., the value of the desired temperature), which must be given in a separate N3 document. At this point, we use a delimiter (a line containing at least ten number signs) to implement a very simple multipart request body that contains both goal and input (see Lst. 8). Given the goal, the input, and the knowledge base, the EYE reasoner is called to produce a proof, which is encoded in N3. The lemmas of the proof include the necessary requests defined by RESTdesc as well as dependency information. Our reasoning

[5]http://www.antlr.org/

[6]https://github.com/antlr/grammars-v4/blob/master/turtle/TURTLE.g4



Execution Plan "Set temperature Kel
Living Room"

```
@prefix : <ex#>.
@prefix ex: <http://example.org/#>.
@prefix log: <http://www.w3.org/2000/10/swap/log#>.
@prefix st: <http://purl.org/restdesc/states#>.

{
  ?goal a st:State;
        log:includes {?loc ex:hasTargetTemperature ?t}.
  ?t ex:derivedFrom :input.
  ?loc :name "Living Room"
}=>{

}.
```

Input Triples

```
@prefix : <ex#>.
@prefix ex: <http://example.org/#>.
@prefix dbpedia: <http://dbpedia.org/resource/>.

:input a dbpedia:Temperature;
  ex:hasValue "300";
  ex:hasUnit "Kelvin".

```

PUT coap://device:1002/temperature
POST coap://device:1001/cel2degf
POST coap://device:1003/k2cel

**PUT** coap://device:1002/temperature

Request Body

$a

**POST** coap://device:1001/cel2degf

Response Body

$a

(a) User Agents (b) Semantic IDE

Fig. 3. (a) The Android-based user-agents demonstrate how the interaction with a smart environment works when using our system architecture.
(b) The screenshot of our Semantic IDE gives an idea of our Web-based tool to develop semantic descriptions. The IDE supports multiple workspaces, allows to store and load device and goal configurations, provides syntax-highlighting for Notation3, and can visualize execution plans.

server parses the output and the interface returns the execution plan. The plan is a JSON-encoded list of request definitions that uses references to link responses of one request to the arguments of another request.

```
{
  ?goal a st:State;
          log:includes {?s ex:hasTargetTemperature ?t}.
  ?t ex:derivedFrom :input.
} => { }.
##############################
:input a dbpedia:Temperature;
        ex:hasValue "23";
        ex:hasUnit "Celcius".
```

Listing 8. A simple multipart request for the execution plan interface that includes the goal definition and the input triples separated by a delimiter.

## C. User Agents

To better demonstrate our practical approach, we implemented two Android user-agent applications that allow to interact with the reasoning server and our IoT devices. The green user agent depicted in Fig. 3(a) uses the query interface to retrieve the list of open questions and provides the list of suggested answers. When the user saves his answer, it is sent to the reasoning server and stored in the knowledge base. The blue user agent in Fig. 3(a) provides a front end to define goals and automatically runs the execution plan when the reasoning server finds a proof.

## D. Semantic IDE

The Semantic IDE provides a development environment for semantic device descriptions. It consists of an HTML5 front end that communicates with a RESTful back end using HTTP. The IDE provides a Notation3 editor that supports syntax-highlighting and code completion, a dialog to answer open questions, form-based access to the query interface, and access to the execution plan interface where the resulting requests are also visualized. There are two modes:

*1) Connected:* By providing the base URI of the reasoning server, the workspace attaches to a real system with its standalone reasoning server and available IoT devices. The editor can be used to browse and debug the RESTdesc descriptions of the devices. When an execution plan is found, it can also be executed directly from the IDE, that is, it performs the requests on the actual devices.

*2) Virtual:* The IDE can also simulate environments by instantiating an internal reasoning server and feeding the knowledge base directly from the Notation3 editor. The descriptions are stored as virtual devices that can be enabled and disabled dynamically in the environment. Furthermore, the RESTdesc format can be edited during runtime, which fosters easy experimentation with RESTdesc and rapid prototyping of semantic mashups.

In both modes, the workspace configuration including defined queries and goals can be stored and loaded again later. The screenshot in Fig. 3(b) gives an idea of our Semantic IDE front end.

## IV. Evaluation

The flexibility of mashup creation is the central motivation for using semantics. For this, execution plans must be calculated dynamically to take into account all available services as well as the current physical state of the environment. The practical feasibility of this approach thus depends on the time required to calculate an execution plan: not only do users expect their smart environments to be responsive, we also need the input states to still be valid upon termination of the reasoning process (i.e., upon execution of the mashup).

In this section, we evaluate how properties of the environment affect the reasoning times when creating execution plans. In the long run, we assume about 250 smart devices to be present in a typical environment, which is a reasonable assumption for future smart environments according to the literature [3], [8], [19]. Each device can provide multiple services such as *power-on/off*, *set-parameter*, *read-sensor*, and *actuate*. We expect many simple devices that only provide a single service such as motion sensors or light bulbs, but also a few complex devices with many services such as infotainment systems or cleaning robots. To dimension our experiments, we use scenarios with about 250 devices and four services each on average (i.e., about 1,000 services in total). Note, however, that 250 devices is not an upper bound and does not represent a technical limitation for the approach put forward in this paper.

We intentionally neglect the time required for communication, e.g., to transmit execution plans or execute requests within the resulting mashups. While these times can be significant in constrained environments, they are mostly independent from the concrete approach used for composing services in a smart environment: even statically configured mashups experience a similar overhead when invoking service requests.

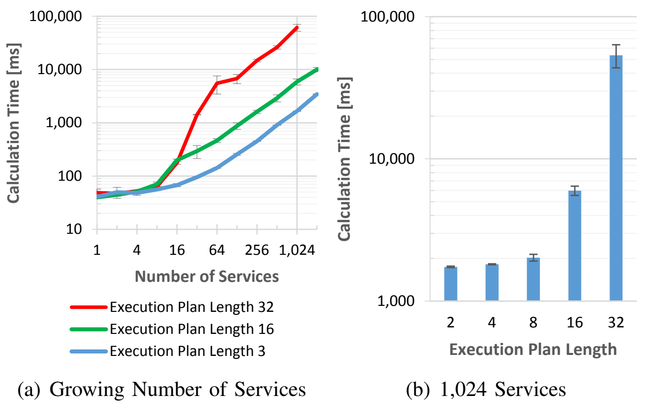


Fig. 4. The reasoning time grows exponentially with the number of services. The execution plan length describes the complexity of a mashup, that is, how many services need to be chained to meet the given goal.

### A. Experiment Setup

In our evaluation, we measure the time required to derive an execution plan in different smart environments while varying different parameters. As described in Sec. III, the reasoning server caches all service descriptions locally and updates the environment state in real time by observing the dynamic descriptions. The experiments are done on a Ubuntu-14.10 laptop with Intel(R) Core(TM) i7-4600U@2.10GHz using SWI-ProLog 6.6.6-amd64[7] to run the EYE reasoner 7433/2014-09-30 (which only runs on a single core) and Oracle Java 1.8.0_45-b14 to execute the reasoning server and our evaluation wrapper. Each experiment series is repeated 20 times and error bars indicate $\pm 1$ standard deviation.

### B. Number of Services

As a baseline, we show how the semantic service composition approach scales for a growing number of abstract services.[8] We calculate execution plans for up to 2,048 available services (i.e., 512 devices) to see how the approach performs when environments go beyond our assumption of 250 devices. In addition, we show results for a stepwise increase in the complexity of the execution plan: the reasoner has to chain more and more services that depend on each other to achieve the given goal.

Fig. 4(a) shows a log–log plot of the reasoning time over a growing number of services. The plotted series represent different execution plan lengths. For up to about eight services, we see the startup cost of the EYE reasoner, which dominates the runtime independently of the number of services. From there onward, the curves are approximately linear, indicating that the heuristics of the proofing tactics employed by EYE run in polynomial time.

Fig. 4(b) shows how the length of derived execution plans affects the reasoning time. For this, we fix the number of

[7]http://www.swi-prolog.org/

[8]Since the total mashup derivation time is the most relevant metric for end users, we report that number in all our graphs. This is different from [13] and [18], who differentiate between the time required for parsing and the reasoning itself.

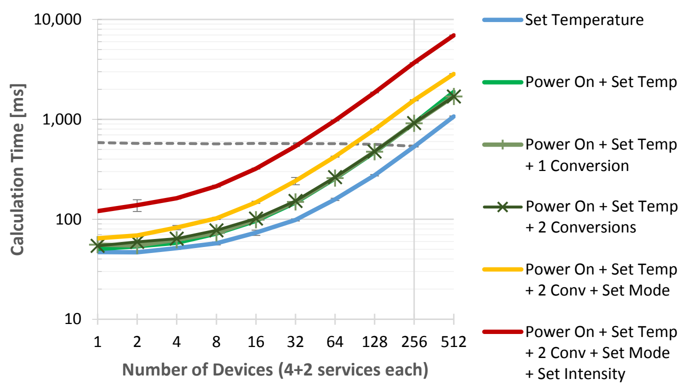


Fig. 5. These are results for a realistic IoT scenario where the goal requires a mashup of up to four stateful and two stateless services to set the temperature. We increase the execution plan length from one to six for each series). Classic stateless services such as the conversion services (series with cross markers) affect the reasoning time only marginally. The dashed gray line shows the reasoning time for 256 devices with a varying ratio of ($x$) relevant and ($256 - x$) non-relevant services for the goal (see Sec. IV-D).

services to 1,024 and gradually double the required service chain length through the goal definition. Increasing execution plan lengths exhibit the exponential growth of the underlying satisfiability problem. This impact can also be seen in the different slopes of the series in Fig. 4(a).

In general, the numbers are much higher than in previous work [13], [18], where the range of seconds is reached at about 4,096 services. The reason is that we use fully functional descriptions of conversion services (similar to Lst. 1), while [13] and [18] only use single triples as service description.

We conclude that the primary challenge for semantic mashup composition lies in the amount of service dependencies. For a medium execution plan length of ten service interactions (requests), the response times enter the critical order of seconds at about 256 services in the environment. For 1,024 services, corresponding to our assumption of about 250 devices, the reasoning time is already in the order of multiple seconds on a relatively strong CPU (i.e., not feasible for Raspberry Pi types of devices).

## C. Mashup Complexity

In this experiment, we define a realistic IoT scenario with devices that host multiple services and goals that require complex mashups out of different stateful services. For this, we use the air conditioner (AC) device presented in Fig. 2 that provides four services in its semantic description (*power-on/off*, *set-temperature*, *set-intensity*, and *set-mode*). To include stateless services in the experiment, we added two additional conversion services to each AC to convert between degrees Fahrenheit, degrees Celsius, and Kelvin, resulting in six services in total.

We measure the reasoning times for six different goals, each requiring a more complex execution plan than the previous:

a) Set the temperature at one specific location (length 1)
b) Like a), but the AC must be powered on first (length 2)
c) Like b), but the input must be converted once (length 3)
d) Like b), but the input must be converted twice (length 4)
e) Like d), but the goal also specifies the mode (length 5)
f) Like e), but the goal also specifies a specific fan intensity (execution plan length 6)

The different series in Fig. 5 show how the linearly increasing mashup complexity impacts the reasoning time. In this case, the growing offset of the curves is not caused by the execution plan length (which is relatively small), but by the state ontology. With stateful IoT services, a proof must consider the possible environment states which significantly increases the state space for the reasoner, leading to longer execution times. In contrast, the stateless conversion services have almost no impact on the reasoning time: the two series with the cross markers behave almost identical to goal b) without the conversions.

For 250 devices, the reasoner can compute execution plans with up to three stateful services in reasonable time. With four stateful services, the calculation time is already at 3.7 seconds for 250 devices. This requires significant improvement considering that smart environments in the IoT are primarily about services to manipulate the state of the physical world through actuators. Sensing, however, does not require the state ontology (in case no dynamic configuration is required). Hence, typical mashups with a mixture of sensor and actuator services can already be generated in reasonable time using the semantic composition approach.

## D. Service Diversity

Verborgh et al. state that the EYE reasoner can “discriminate between relevant and non-relevant descriptions,” so that a large number of non-relevant services do not significantly affect the reasoning time [18]. The experiments in [18] indicate this for stateless services. We examine this feature in our IoT setting with four stateful services identical to Sec. IV-C. The experiment fixes the total number of devices to 256 and gradually change the ratio between relevant services (number of ACs) and non-relevenat services (non-trivial random dummy devices with the same number of services) from one AC to 256 ACs out of 256 devices.

The dashed gray line in Fig. 5 shows an almost constant calculation time over the ratios. When the state ontology is involved, the reasoner is thus not anymore able to discriminate between relevant and non-relevant descriptions. We explain the slight decrease in the calculation time for a growing number of ACs through synergy effects when more descriptions become identical (except for the location Property).

## E. Open Questions

Finally, we evaluate the overhead of our open question configuration feature. So far, the device locations were given through an explicit Statement in the device descriptions. In this experiment, we compare this with device descriptions that define an open question for the location. When calculating the execution plan, the answers are already given and stored in the knowledge base.

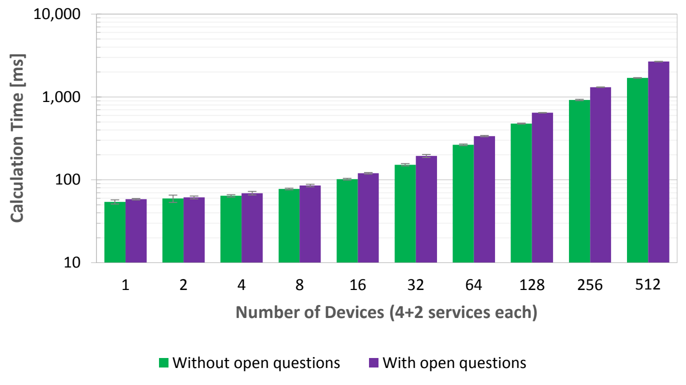


Fig. 6. With our open questions, context information such as the location can be configured dynamically through user feedback or other IoT devices, e.g., through a localization system. This indirection requires additional inference steps for each device and causes an overhead in the execution plan calculation.

Fig. 6 shows a comparison for the "Power On + Set Temp + 2 Conversions" goal from Sec. IV-C. At 256 devices, the overhead for open questions in relation to the static configuration, which takes 920 ms to compute the proof, is 43% or 391 ms. While this mechanism thus enables more dynamic configuration through interactive feedback, the reasoner requires additional inference steps to produce the necessary Statements. This could be optimized by adding explicit Statements to the knowledge base when a questions is answered—at the cost of losing flexibility in dynamic environments.

## V. Conclusions

In this paper, we show multiple extensions to semantics-based service composition in smart environments. In particular, we build on the approach presented in [13] by facilitating the management of stateful services in physical mashups and demonstrate the general feasibility of RESTdesc and semantic reasoning in resource-constrained environments using CoAP. We also introduced the concept of *open questions*, which allows for dynamic configuration and interactive composition by enabling the reasoner to fetch additional information from (potentially human) users of the system.

Since providing semantic descriptions for RESTful APIs—in particular in the context of the IoT—is relatively new, we developed a tool that fosters the experimentation and design of RESTdesc descriptions for IoT devices. Our Semantic IDE can attach to live environments to help debugging the distributed system and can simulate devices for rapid prototyping. It supports developers with a comprehensive RESTdesc editor, Web-based reasoner interfaces, execution plan visualizations, and configuration management.

The power of using a semantic reasoner is that service composition becomes automatic, dynamic, and more fault-tolerant in smart environments. Our evaluation results show that the time required by the reasoner to calculate an execution plan represents a potential obstacle to applying this approach. [13] and [18] demonstrated that parsing the service descriptions is indeed a major overhead. On top of this, we found that the time required for a proof is significantly increased when including stateful services, which are natural to physical mashups. At present, semantics-based service composition is feasible for smart environments with about 250 devices when the goal is reachable through medium-sized execution plans with around ten requests involving a low number of stateful services.

There are still open research questions to improve the heuristics of reasoners to deal with the state space explosion for complex proofs. Optimized implementations that can also leverage multiple cores could further improve execution plan calculation times. There are, however, further possible improvements at the system level: for instance, the knowledge base could cache results from inference steps, which can in particular improve our dynamic configuration approach through open questions. Furthermore, our reasoning server could leverage additional service descriptions such as the CoRE Link Format to pre-select the relevant services, and hence reduce the state space. Solving these issues will enable a dynamic approach for service composition that is applicable across different application as envisioned in the IoT.

## References

[1] T. Berners-Lee, J. Hendler, and O. Lassila. The Semantic Web. *Scientific American*, 284(5):34–43, 2001.

[2] M. Blackstock and R. Lea. Toward a Distributed Data Flow Platform for the Web of Things (Distributed Node-RED). In *Proc. WoT*, Cambridge, MA, USA, 2014.

[3] A. Brandt, J. Buron, and G. Porcu. Home Automation Routing Requirements in Low-Power and Lossy Networks. RFC 5826, 2010.

[4] R. T. Fielding. *Architectural Styles and the Design of Network-based Software Architectures*. PhD thesis, UC Irvine, 2000.

[5] R. Goldblatt. *Logics of time and computation*. Center for the Study of Language and Information Stanford, 1992.

[6] D. Guinard, V. Trifa, and E. Wilde. A Resource Oriented Architecture for the Web of Things. In *Proc. IoT*, Tokyo, Japan, 2010.

[7] K. Hartke. Observing Resources in CoAP. draft-ietf-core-observe-16, 2014.

[8] ITU-T. Short range narrow-band digital radiocommunication transceivers - PHY and MAC layer specifications. Recommendation G.9959, Jan. 2015.

[9] M. Kovatsch, S. Duquennoy, and A. Dunkels. A Low-Power CoAP for Contiki. In *Proc. MASS*, Valencia, Spain, 2011.

[10] M. Kovatsch, M. Lanter, and S. Duquennoy. Actinium: A RESTful Runtime Container for Scriptable Internet of Things Applications. In *Proc. IoT*, Wuxi, China, 2012.

[11] M. Kovatsch, M. Lanter, and Z. Shelby. Californium: Scalable Cloud Services for the Internet of Things with CoAP. In *Proc. IoT*, Cambridge, MA, USA, 2014.

[12] M. Maleshkova, C. Pedrinaci, and J. Domingue. Investigating Web APIs on the World Wide Web. In *Proc. ECOWS*, Ayia Napa, Cyprus, 2010.

[13] 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 *Proc. IoT*, Cambridge, MA, USA, 2014.

[14] Z. Shelby. Constrained RESTful Environments (CoRE) Link Format. RFC 6690, 2012.

[15] Z. Shelby, K. Hartke, and C. Bormann. The Constrained Application Protocol (CoAP). RFC 7252, 2014.

[16] Z. Shelby, M. Koster, C. Bormann, and P. van der Stok. CoRE Resource Directory. draft-ietf-core-resource-directory-04, 2015.

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

[18] 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 *Proc. SSN*, Boston, USA, 2012.

[19] ZigBee Alliance. Home Automation Public Application Profile. ZigBee Document 053520r26, Feb. 2010.