Orchestration of Network-accessible Components with Condition-Action Rules

Andreas Harth, Institute AIFB, Karlsruhe Institute of Technology (KIT), Germany

Position paper for the Workshop on IoT Semantic/Hypermedia Interoperability, July 2017, Prague, Czech Republic

Current version available at http://harth.org/andreas/2017/wishi/

Abstract

The paper presents challenges concerning the integrated access to sensors and actuators in networked environments. We draw on experience gained in projects around data integration and system interoperation with both academic and industrial partners, and from lessons learned during the development of several prototypes. We have identified architectural mismatches that require mapping and integration before applications can access and manipulate the state of network-accessible components in a uniform manner. We tackle mismatches along the following dimensions: network protocol, data format and data semantics. Our system architecture builds on ideas from Representational State Transfer and uses standards around Linked Data. For discovery we assume that network-accessible components provide hyperlinks to other components. On top of the standardised interfaces we use an agent application model based on a sense-act cycle; we use a rule-based language to integrating state representations, follow links and specify application behaviour.

Introduction

Modern software systems have to incorporate an increasingly diverse set of components, both in terms of hardware (sensors, actuators) and software (APIs, services). The heterogeneity of the components leads to a high cost for building applications. Performing the integration in a monolithic application requires glue code to access and manipulate the state of each component. Similarly, performing the integration in a distributed application also requires glue code in the form of wrappers to access and manipulate the state of each component. A way to reduce the integration cost per application is to provide common interfaces to components, and to reuse the common interfaces in multiple applications.

When integrating different components, one can employ various strategies for deciding on the features of the common interface. One strategy is to use the union of the feature sets of the source interfaces; another strategy is to use the intersection of the feature sets of the source interfaces; yet another strategy is to pick and choose among the feature sets. However, as the components use different, sometimes inherently incompatible, paradigms for accessing and manipulating component state, the specification of uniform interfaces remains a challenge. In addition, the requirements for interfaces are very broad: interfaces should be simple and easy to use and implement, yet at the same time they should satisfy the requirements of very different scenarios. Often, the requirements of the various scenarios are not made explicit, as they are completely clear to anybody who is part of a particular community. However, once one tries to create applications that access interfaces of devices and components from different communities, the various unspoken assumptions cause problems.

We use a core idea from Web Architecture and Representational State Transfer to keep the uniform interface specification manageable. We dictate constraints to limit the degree of freedom in the way that interfaces can be implemented. Minimal interfaces make cost-effective interoperation possible. At the same time, the interfaces should be sufficiently high-level and be building on a universal model for computation to put as little restrictions as possible on what can be theoretically achieved. There is always a trade-off, and different people have different tastes and styles. Our proposal is to build on standards that have proven to work in the past: internet technologies, as those have been successfully deployed on a global scale. However, we think that the abstraction the core internet technologies provide (basically a channel where packets can be send and received) is too low-level. Web technologies and the resource abstraction with

request/response communication between user agents and origin servers provide a higher level abstraction, including error handling, that has been successfully deployed on the web.

Another useful feature of the web are hyperlinks to allow for decentralised publication and discovery. To leverage the full power of hyperlinks, which allow for applications to discover new things at runtime, the newly discovered things have to be self-described. In such an environment, applications could then follow hyperlinks and use arbitrary data and functionality from hitherto unknown components. At least that is the vision; the realisation of that vision has been proven to be challenging, not the least because developers have to create applications which operate on data with arbitrary schema unknown at design-time of the application. Semantics provides the means for at least some freedom for mapping and integrating data with different schema.

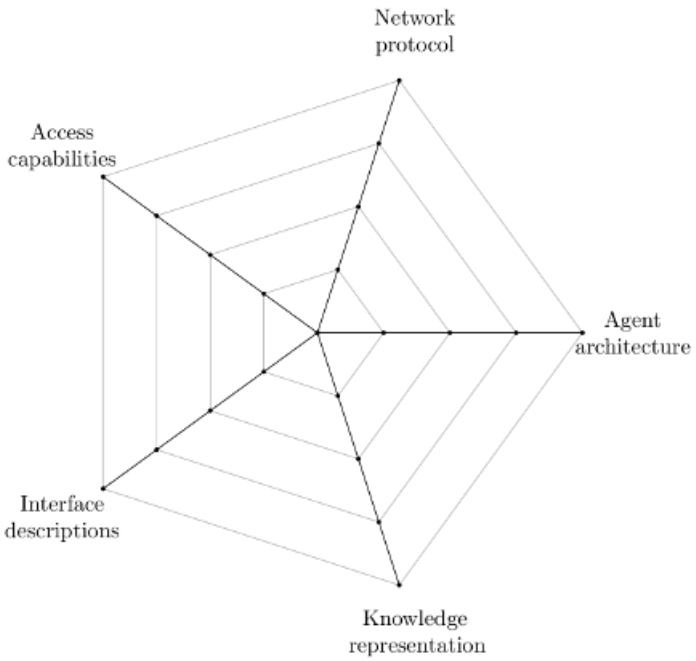
In the following we present the architectural mismatches we have identified in our work on academic and industrial data integration and system interoperation applications. We describe mismatches concerning network protocol, data format and data semantics that preclude applications from accessing data and functionality of components in a uniform manner. To overcome the mismatches, we assume a uniform addressing scheme (URIs), uniform network protocol (HTTP) and uniform data format (RDF), on top of which applications operate. The behaviour of applications is specified using rules. These rule-based applications can access and integrate resource state, follow links to discover new resources, and change resource state to implement application behaviour. While we have an eye on elaborate functionality such as artificial intelligence planning and model checking, the focus on our work so far has been on optimised execution of application behaviour specifications. Due to space constraints, we can only briefly introduce the rule-based language, but we provide pointers to further material.

Overview

We start with describing the constraints on component interfaces, followed by giving an overview of the application model.

Continuum of Choices

The following diagram illustrates the dimensions on which we can make a choice regarding interfaces and application architecture.



- Network protocol: assuming a REST-based protocol such as HTTP (1.1 or 2.0) and COAP, the choice is between supporting read, update, delete, create and observe operations. Yet completely different options are network architectures building on a centralised message bus, which would require an interface to create and manage subscriptions.

- Access capabilities: we can either assume a simple retrieve operation (for GET) and overwrite operation (for PUT), which in each case the message body is the entire resource state. Other choices are transmitting deltas between resource states or patch instructions. Finally, one could assume a general query interface, as for example database mediator systems assume (albeit only on Read operations).
- Interface descriptions: starting with no dedicated interface descriptions and just assuming the HTTP (or COAP) semantics for operations, we could layer additional descriptions on top, starting with the input (request message body) and output (response message body) messages, and adding descriptions related to the resource state (precondition and effects). Finally, assuming query interfaces, the interface descriptions could cover access restrictions related to the shape and structure of queries, e.g., query variables that have to be bound.
- Knowledge representation: assuming an RDF-based knowledge representation format, we could layer ontology languages with progressively more expressive power on top, starting with RDFS, OWL LD and then the more expressive OWL 2 profiles.
- Agent architecture: finally, we could assume different agent architectures (see Artificial Intelligence: A Modern Approach), ranging from simple reflex agents, model-based reflex agents, goal-based agents, utility-based agent to learning agents.

The initial reaction is to go for the most expressive choice in the area (or areas) one is familiar with, while not considering the areas one does not know about or care. For readers interested in approaches that maximise the feature set along almost every dimension we recommend to consult the extensive work on semantic web services.

Constraints for Component Interfaces and Application Model

To reduce the effort for integrated access to component state, we assume the following constraints for the uniform interfaces to components and the application model:

- Network protocol: we assume a REST-based abstraction, where each component provides resources that are identified via URIs. Components provide a HTTP server interface and allow for read (GET) and write (PUT) operations on the provided resources using HTTP. These constraints relate to RMM level 2. RMM level 2 roughly means that we view sensors and actuators as resources and identify the resources with URIs. Reading out sensor state is done via GET, and changing actuator state is done via PUT.
- Access capabilities: we assume a simple state-based read and write interface and do not consider query capabilities.
- Interface descriptions: we do not assume any interface descriptions, but require that the interfaces actually follow the HTTP semantics.
- Knowledge representation: we assume that the resource state is represented in RDF and support RDFS and a small subset of OWL called OWL LD, which works well together with SPARQL, the query language for RDF. We assume that the RDF documents provide hyperlinks to other resources. In addition, we assume that there exists an index resource on each component as entry point, and that the index resource links to other resources on the same component.
- Agent architecture: we assume simple reflex agents. Straightforward should be an extension to model-based reflex agents that know about the semantics of HTTP operations. We do not consider goal-based agents, as those would require expressive interface descriptions as input for the automated planning procedure.

Roughly, for the interfaces to components we assume an interface adhering to the Linked Data principles, modulo the fact that Linked Data is read-only, that is, supports HTTP GET only. Read-Write Linked Data and the Linked Data Platform specification explain how to support full CRUD operations in conjunction with Linked Data.

We call an outgoing HTTP request an action, and an incoming HTTP request an event. For each request/response pair, we say that a user agent emits an action (the outgoing HTTP request), and an origin server receives an event (the incoming HTTP request) [Harth and Käfer 2016].

We limit the number of active components per application to one; that active component is a HTTP user agent that polls resource state. The requirement for one user agent per application could be relaxed, but is useful in the beginning to reduce complexity. Also, one could add a server interface to the controlling component in an application. However, the same argument regarding reduced complexity applies.

Please note that the constraints are not minimal, that is, component providers are free to include support for additional features, for example HTTP 2.0 server push or COAP observe, or expressive OWL 2 knowledge representation.

We require uniform interfaces for two reasons: first, to be able to reuse components between applications and thus drive the overall integration cost down; second, we want to specify application behaviour over different networked components using high-level executable specifications. Applications could be seen as simple reflex agents with behaviour specified in condition-action rules. The applications are structured around a sense-act cycle:

- In the sense step, the interpreter acquires the current state of the relevant resources, including the following of links to discover new resources and fetch their state. How to follow links is specified using condition-action rules. Conditions are evaluated over the current resource state as known by the interpreter (optionally taking into account the semantics of RDFS and OWL LD terms), and actions are HTTP GET requests to fetch new data. Optionally, the interpreter can evaluate a query over the integrated resource state at the end of the sense cycle.
- In the act step, the interpreter applies condition-action rules to decide on which unsafe requests (requests that change the state of resources) should be carried out. Conditions are evaluated over the current resource state as known by the interpreter, and actions are HTTP PUT, POST, DELETE, PATCH requests to manipulate resource state.

The interpreter could run applications in two modes:

- Time-triggered: the sense-act cycle runs at specified times.
- Event-triggered: the sense-act cycle runs whenever a specified event (incoming request) has taken place.

Given that our agents do not provide a server interface, the agents cannot receive events and hence all our applications are time-triggered.

Resolving Mismatches

We now consider how to resolve mismatches that may occur when integrating components from different providers. We distinguish between protocol mismatches, syntax mismatches, and semantic mismatches.

Some of the mismatches occur because the source components support more features than our minimal constraints. However, even if all source components stay within the set out constraints, some mismatches may occur.

Mapping Different Protocols

Because not all components implement the constrained interface, we require wrappers to bring the components to the same level. Next to syntax and semantics of resource representations (covered in the following sections), we might need to map fundamentally different networking protocol paradigms.

The constraint interface assumes a HTTP server interface access, with the components being passive and waiting for incoming requests.

Some networking environments assume active components, i.e., components that emit data at intervals. To be able to include those components in our system architecture, we require a wrapper that provides the current resource state via GET on HTTP URIs. That is, the wrapper has to receive the messages from an active component and store the resource state internally, and make the resource state (passively) accessible via GET on URIs. Analogously, changing resource state (via PUT, POST or DELETE) has to be converted to the appropriate messaging format and then sent on to the final destination.

We have implemented such a protocol mapping between the Robot Operating System (ROS) message bus and our Read-Write Linked Data interface abstraction. The potential drawback is that polling is out of sync with the arrival of events, and that the application might miss state changes due to a polling frequency that is out of sync with the update frequency.

Mapping Different Representation Syntax

Even if all components provide HTTP server interface, the representation of the data might still differ (for example, binary formats, CSV, TSV, JSON or XML). Hence, the wrapper (sometimes also called "administration shell" or "adaptor") needs to lift the data model of the different components to a common data model. We assume the RDF data model. Given that RDF can be serialised into different surface syntaxes, the wrappers can choose to support different serialisations, for example RDF/XML, JSON-LD or Turtle.

Mapping Different Representation Semantics

Even if all sources use the RDF data model consisting of subject/predicate/object triples, data from different providers might be structured and represented differently. We survey the different ways to represent data even within the RDF data model in the following.

Different Terms

However, some ontologies have different assumptions. SOSA, for example, assumes that we want to model a way to represent a journal of sensing and actuating activities. SOSA includes a sosa:Observation and a sosa:Actuation class. Instances of these classes represent results from observation and actuation events. The communication protocol is unspecified. However, the modellers assume implicitly some service-oriented architecture.

SOSA is not directly applicable to RMM level 2 architectures. Performing a GET on a resource that returns an instance of (the latest) sosa:Observation could be seen as adhering to the defined HTTP GET semantics (intuitively, return the current resource state upon GET). The result of a PUT (in SOA terminology the postcondition or effect) is an instance of sosa:Actuation, which could be sent in a response message to a PUT. However, it is unclear what the message body should be for a PUT request (to change the state of an actuator).

Web of Things Thing Description (TD), on the other hand, has a more immediate state-based representation. State representations include the "writable" flag, which indicates whether a representation (such the current temperature reading or the state of a switch) can be written.

If the mental models (concerning the resources) of different vocabulary terms are similar, then different terms can be easily mapped using RDF-based technologies: For example, TD could be mapped to SSN using the following triple:

```
td:Thing rdfs:subClassOf ssn:System .
```

Please note that both SSN/SOSA and TD are still being specified, so any text related to these vocabularies could be outdated.

Different Modelling Granularity

Modelling granularity refers the meaning of resources. Consider two datasets about cities. One dataset uses a resource to identify the city of Berlin, whereas another dataset uses a resource to identify the metropolitan region of Berlin. As the different sources use different modelling granularity, one has to careful when mapping resources from different sources.

Different Assumptions about Time and Aspect

Another mismatch on the level of semantics is that of aspect (related to linguistic aspect). Messages could be represented as current state (for example, lat/long of the position of a person), or using a higher-level description (for example, stating that the person is walking from A to B). Integration of the different aspects (resource-based vs. event-based) is an open challenge.

Building Applications

The goal of our prototypes was to have executable specifications of application behaviour. Given that we wanted to reduce complexity as much as possible for our minimal system architecture,

we wrote application behaviour using directly executable condition-action rules. That is, we implemented simple reflex agents as opposed to goal-based agents.

Some people in our projects were not comfortable with specifying rule-based agents, so they access the component state with code in imperative programming languages. We first briefly discuss such an approach, before introducing the sense-act cycle and finally discussing goal-based agents.

Accessing Component State using Imperative Programming Languages

Another scenario emerged in one of our projects where partners wanted to interface with components in procedural programming languages (rather than using condition-action rules). They needed to generate messages from programming language objects. To that end, the open world assumption in RDFS and OWL turned out to be problematic, as validation of incoming and outgoing messages was not possible (for serialisation and deserialisation). Hence, the vocabulary descriptions in RDFS and OWL were augmented with descriptions in SHACL to specify request (input) and response (output) message bodies.

Safe Requests and Link Following (Sense)

To have resource state available locally, an application has to specify some initial resources that form the basis for further processing. For example, the following two RDF triples encode a HTTP GET request to an index resource of an IoT device.

```
[] http:mthd httpm:GET;
   http:requestURI <http://raspi.example.org/index.ttl> .
```

We can also write rules that follow links. We use condition-action rules encoded in Notation3 syntax. Notation3 is a superset of Turtle, and extends the RDF data model with variables and graph quoting, so that subject and object of triples can be entire graphs.

The following rule specifies that "next" links should be followed (for example, to fetch all data from a paged representation):

```
{ ?x :next ?next . } => { [] http:mthd httpm:GET;
                            http:requestURI ?next . } .
```

Such rules allow for specifying that certain links to URIs should be followed.

URI templates are another way to specify links:

```
{ ?x :lat ?lat ; :long ?long . } =>
         { [] http:mthd httpm:GET;
              http:requestURI "http://geo.example.org/geocode?la={lat}&lo={long}" .
```

In a sense, the input description (which parameters to supply) is in the rule heads (the action part). A description of the parameters is in the rule body (the condition part). With enough applications specifying rules similar to the one above, it would be possible to extract input descriptions from these rule-based programs.

Rule application works as follows. The interpreter starts with carrying out the initial requests. The combined resource state forms the basis over which the conditions in the condition-action rules are evaluated. The interpreter applies these condition-action rules to exhaustion, that is, until now new data (or requests) can be generated anymore and a fixpoint is reached.

After carrying out the GET requests (the sense part of a cycle), the local dataset contains all relevant sources. We can optionally take into account the semantics of RDFS terms and a subset of OWL terms. We can also evaluate a SPARQL query on the local dataset containing the (more or less) current resource state. Depending on the size of the data and response times of the components, we can run 10 to 50 sense procedures per second.

Unsafe Requests (Act)

To issue unsafe requests, we allow for unsafe request templates in the head of rules:

```
{ ?x :temperature ?temp .
  ?temp :greaterThan 20 . } =>
          { [] http:mthd httpm:PUT;
               http:requestURI "http://raspi.example.org/r/heating" ;
               http:body { [] a :State ; :state :Off . } } .
```

As our immediate goal is to provide high-level executable specifications based on rules, we assume that people who write rules know the interface and the required payload and hence do not need descriptions.

The interpreter executes the unsafe requests in the act procedure only after the sense procedure has been concluded. The rule application could be non-deterministic, as there could be multiple rules that overwrite the state of the same resources. If one cannot get rid of non-determinism by changing the condition of the relevant rules, different conflict resolution strategies could be applied.

Why Not Have Goal-based Agents?

The rules specifying application behaviour could also be generated using an AI planning approach, based on IOPE descriptions and a goal specification. However, due to the high modelling effort for providing descriptions that specify input, output, precondition and effect (IOPE descriptions), we have reserved such approaches for future work.

In earlier version of our prototypes we have included descriptions of the input and output of services [Speiser and Harth 2011]. For example, the required parameter for GET requests (encoded in the URI), and the output of the response message. However, for our simple reflex agents we wrote rules directly encoding the parameters in URI templates and did not make use of the descriptions. The descriptions, because they were manually constructed and did not serve a purpose (not even for generating documentation), soon became outdated, as developers changed the API but did not change the descriptions.

Goal-based agents would have required elaborate descriptions of the interfaces to the components, and a goal specification that serves as input to the automated planning algorithm. The automated planning algorithm would then figure out which components to arrange in which way to achieve the goal, and output essentially a close equivalent to simple reflex agents. Please observe that taking the shortcut of directly specifying the simple reflex agent does not preclude us from using a more elaborate goal-based approach in an additional layer.

Conclusion

Data integration and system interoperation are complex problems. We believe that in order to at least solve some of the problems there have to be restrictions in place that (some would say severely) constrain the features of the source components. The idea is to keep things simple. We have presented a system architecture that provides integrated access to networked components following a constrained interface in conjunction with a rule-based condition-action language to access resource state, integrate data and specify application behaviour. We believe that the standardisation of interfaces can benefit from the knowledge of which applications are supposed to make use of the interfaces, and how these applications are going to be developed.

The system architecture synthesises network protocol, knowledge representation and agent architectures into a unified architecture. While each of the parts provide very powerful features, the synthesis requires to reduce the feature set of each part to keep the complexity of the combination manageable. Each additional feature imposes a higher implementation effort. While there is nothing wrong with the vision of having goal-based, utility-based learning agents that get from components real-time push updates encoded in an expressive OWL2 profile, our approach was to try to identify a minimally viable architecture that can form the basis for more elaborate systems.

We have implemented the described interfaces and applications in several prototypes in the areas of geospatial data integration [Harth et al. 2013] as well as industrial applications around product design and validation [Keppmann et al. 2014], [Keppmann2 et al. 2014]. Our prototypes achieve update rates sufficient for industrial requirements. For example, we have achieved update rates of around 30 Hertz in virtual reality/augmented reality applications. We believe that

the encountered problems and the proposed solutions generalise to other application areas in the area of Industry 4.0 [Harth et al. 2016] and the Internet of Things [Käfer et al. 2016].

We have begun to work on a formalism based on state transition systems and mathematical logic to provide a rigorous foundation for accessing components and specifying application behaviour [Stadtmüller et al. 2103], [Harth and Käfer 2016]. Our short-term goal for the formalism is to find a way to specify application behaviour in a form way that is directly executable. On top of the formalism we can layer additional functionality, for example to model systems that detect conditions on integrated resource state described in RDF or to simplify application development for casual users based on a workflow language.

Acknowledgements

The paper is based on joint work with Tobias Käfer, Felix Keppmann, Sebastian Speiser, Steffen Stadtmüller and Rudi Studer. We have to mention Ruben Verborgh who suggested the use of URI templates in rules. We benefitted from discussions with members of the W3C Spatial Data on the Web working group, especially from discussions with Maxime Lefrançois. The presented research has been partially funded by the projects ARVIDA and i-VISION.

References

[Harth and Käfer 2016] Andreas Harth, Tobias Käfer. "Towards Specification and Execution of Linked Systems". 28. GI-Workshop Grundlagen von Datenbanken, May 24 - 27, 2016, Nörten-Hardenberg, Germany.

[Harth et al. 2013] Andreas Harth, Craig Knoblock, Steffen Stadtmüller, Rudi Studer and Pedro Szekely. "On-the-fly Integration of Static and Dynamic Linked Data". Fourth International Workshop on Consuming Linked Data (COLD 2013). Co-located with ISWC 2013, Sydney, Australia.

[Harth et al. 2016] Andreas Harth, Tobias Käfer, Felix Leif Keppmann, Dimitri Rubinstein, René Schubotz, Christian Vogelgesang. "Flexible industrielle VT-Anwendungen auf Basis von Webtechnologien". VDE Kongress 2016, Internet der Dinge, Nov 7-8, 2016, Mannheim, Germany.

[Käfer et al. 2016] Tobias Käfer, Sebastian Bader, Lars Heling, Raphael Manke and Andreas Harth. "Exposing Internet of Things Devices on REST and Linked Data Interfaces". 2nd International Workshop on Interoperability & Open Source Solutions for the Internet of Things. Co-located with 6th International Conference on the Internet of Things (IoT 2016). Nov 7, 2016, Stuttgart, Germany.

[Keppmann et al. 2014] Felix Leif Keppmann, Tobias Käfer, Steffen Stadtmüller, René Schubotz and Andreas Harth. "High Performance Linked Data Processing for Virtual Reality Environments". International Semantic Web Conference (Posters & Demos). ISWC 2014, Riva del Garda, Italy.

[Keppmann2 et al. 2014] Felix Leif Keppmann, Tobias Käfer, Steffen Stadtmüller, René Schubotz and Andreas Harth. "Integrating Highly Dynamic RESTful Linked Data APIs in a Virtual Reality Environment". International Symposium on Mixed and Augmented Reality (Posters & Demos). ISMAR 2014, Munich, Germany.

[Speiser and Harth 2011] Sebastian Speiser, Andreas Harth. "Integrating Linked Data and Services with Linked Data Services". 8th Extended Semantic Web Conference (ESWC 2011), Heraklion, Greece.

[Stadtmüller et al. 2014] Steffen Stadtmüller, Sebastian Speiser, Andreas Harth, Rudi Studer. "Data-Fu: A Language and an Interpreter for Interaction with Read/Write Linked Data". WWW 2013, Rio de Janerio, Brasil. Please note that we have renamed the system to "Linked Data-Fu" to avoid name clashes with other projects.

2017-07-01