Specifying and Executing Application Behaviour with Condition-Request Rules[*]

Andreas Harth Tobias Käfer
Institute AIFB
Karlsruhe Institute of Technology (KIT)
Germany

Abstract

The paper outlines a method for writing applications operating on components that are linked in a decentralised fashion. Our aspiration is to simplify data integration and system interoperation at scale. In projects we have routinely encountered obstacles for integration and interoperation due to architectural mismatches along several dimensions: network protocol, data format and data semantics. We argue for a uniform interface to components based on a combination of the Representational State Transfer architectural style and the Linked Data principles, as only an uncluttered component interface allows for a concise specification of application behaviour. To specify applications, we present the syntax of a small language consisting of condition-request rules and sketch an operational semantics for the language based on an agent architecture with a sense-act cycle.

1 Introduction

Modern software applications – think of virtual assistants, augmented reality applications or autonomous systems – have to incorporate an increasingly diverse set of hardware and software components. Diversity on the hardware level is due to the fact that the vendors of sensing and actuation devices favour competing network protocols. On the software level, various components use different data formats for communication (e.g., binary vs. XML vs. JSON). Even if all components use the same network protocol and the same message syntax, the semantics of the messages can differ. Although known for decades, these problems associated with heterogeneity in information systems still persist today. A consequence of the heterogeneity of components is a high cost for building applications over networked components. Our goal is to reduce the cost for building applications.

Wiederhold's mediators in information systems [14] are a technical approach for resolving heterogeneity. But mediators are costly to create and maintain. One way of reducing the integration cost per application is to reuse mediators in multiple applications. Still, somebody has to create and maintain the mediators. Also, to avoid an explosion in the number of required mediators, the mediators have to assume one single target interface. A mediator approach works for brown-field projects, where existing components have to be

[*]A preliminary version of this paper was presented at the Workshop on IoT Semantic/Hypermedia Interoperability, July 15-16, 2017 in Prague. Discussions and debates at the STI Summit, Sep 4-5, 2017 in Heraklion helped to shape the introduction. We thank Carsten Bormann, elf Pavlik and Juan L Reutter for constructive comments and reviews.

adapted for interoperation. In green-field projects, a way of reducing the integration cost is to do without mediators in the first place, which again requires agreement on uniform component interfaces.

Various strategies can be employed for deciding on the features of such a uniform 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 components can use sometimes inherently incompatible paradigms for accessing and manipulating component state, the decision on a uniform interface remains a challenge. In addition, the requirements for interfaces are difficult to reconcile: interfaces should be simple and easy to use and implement, so that applications can be built easily; yet at the same time, interfaces should be very flexible and feature-rich. There is always a trade-off, and different people have different tastes and styles.

Our decision regarding uniform component interfaces is to rely on popular technologies around web architecture, which has proven to work on a global scale. A tenet of web architecture are hyperlinks used for decentralised publication and serendipitous browsing. But to allow for applications to browse, i.e., discover new components at runtime, the components have to be self-described. Then, applications could follow hyperlinks and use arbitrary data and functionality from hitherto unknown components. At least that is the vision; the realisation of that vision has turned out to be challenging, not the least because developers have to create applications that are able to operate on components with data semantics unknown at the design-time of the application. Semantic technologies help to mitigate the problem by providing the means for expressing mappings in a logic-based formalism. With such mappings, applications can transparently integrate data with different schema.

The paper provides the following contributions:

- We present the architectural mismatches we have identified in our work on academic and industrial data integration and system interoperation applications[1,2]. 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 network protocol mismatch, we assume a uniform addressing scheme (URIs[3]) and a uniform network protocol (HTTP[4]). To overcome the data format mismatch, we assume uniform data format (RDF[5]) – in other words, Linked Data[6] (Section 3). To overcome the semantics mismatch, we use logic formalisms to be able to formally express the semantics of terms.

- We report on the difficulties we have encountered in bringing existing components to the uniform Linked Data interface. Even with all components using the same interface, mismatches can occur that hamper interoperation. The goal is to resolve mismatches independently of the application that is built on top of networked components (Section 4).

- We introduce a method for specifying the behaviour of applications using rules in Notation3 syntax. Notation3 is a superset of Turtle[7], and extends the RDF data model with variables and graph quoting, so that subject and object of triples can be entire graphs. The rule-based applications can access and

[1]`http://www.arvida.de/en/`, BMBF FKZ 01IM13001A and FKZ 01IM13001G

[2]`http://www.ivision-project.eu/`, EU FP7 GA #605550

[3]`https://tools.ietf.org/html/rfc3986`

[4]`https://tools.ietf.org/html/rfc7230`

[5]`https://www.w3.org/TR/rdf11-concepts/`

[6]`https://www.w3.org/DesignIssues/LinkedData.html`

[7]Because Turtle is defined in a W3C recommendation, and N3 is not, making definite statements about N3 syntax or semantics is difficult.

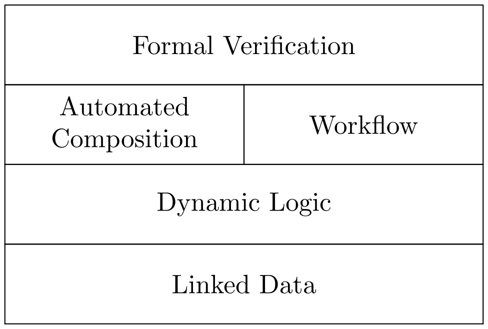

Figure 1: Technology layers for Linked Systems.

integrate resource state, follow links to discover new resources and change resource state to implement application behaviour. Due to space constraints, we can only briefly introduce the rule-based language, but we provide pointers to further material (Section 5).

While we keep an eye on elaborate functionality such as artificial intelligence planning, workflows and model checking, which could be layered on top of a uniform Linked Data interface, the focus of our work so far has been on the optimised execution of logic-based application behaviour specifications. Figure 1 illustrates the technology layers for Linked Systems [3] – methods and associated technologies operating on Read-Write Linked Data – to enable scenarios surrounding virtual assistants, augmented reality applications or autonomous systems in decentralised environments. In terms of the figure, this paper mainly belongs to the "Linked Data" and "Dynamic Logic" layers.

Before we cover the contributions in Sections 3 to 5, we define some terms in Section 2. We cover future topics in Section 6 and conclude with Section 7.

2 Terminology

We now define the core terms used in this paper, starting with Fielding's definition of a component. "A component is an abstract unit of software instructions and internal state that provides a transformation of data via its interface." [1]. An application consists of one or more components.

An agent is an application that acts in an environment. To be able to act in a networked environment of components, we require the notion of a connector. "A connector is an abstract mechanism that mediates communication, coordination, or cooperation among components." [1]. Components can have a server connector for handling incoming requests or a client connector for handling outgoing requests.

We use the term "server" to mean "component with an HTTP server connector". We use the term "user agent" to mean "component with an HTTP client connector that operates on behalf of a user in an environment consisting of servers".

3 Architectural Choices

We start with describing the dimensions of the choices for the system architecture and then we constrain the dimensions, leading to Linked Data.

3.1 Towards Using Web Architecture for Applications

In our experience, the least contentious architectural choice in projects is to rely on internet protocols at the transport layer. Internet protocols have been successfully deployed on a global scale, and the majority of existing component with a network interface already rely on TCP; UDP is used to a lesser extent. Most developers of networked applications today choose TCP without much deliberation. However, we think that the abstraction that TCP and UDP provide – basically a channel where packets can be sent and received – is too low-level, and requires a lot of complexity on the side of applications.

The concepts and technologies underlying the web, namely URIs and HTTP, provide a higher-level abstraction. However, in our experience, the decision for URIs and HTTP for the uniform interface can be contentious. Firstly, because HTTP precludes UDP, and secondly because of the strict separation between user agent – components that can issue requests – and server – components that receive requests. Such a separation is a big constraint. Protocols such as WebSocket allow for bidirectional communication (as, by the way, do UDP and TCP); a similar functionality could be enabled in HTTP with "servients" – components with both a server and a client connector. The question of whether a component should use a client or server connector is related to the choice between pull-based and push-based communication [10]. Finally, one can imagine completely different architectures (and some people do), for example, architectures building on a centralised message bus.

Because our goal is a minimal language for writing applications that combine components, we require big constraints on the component interfaces. In the following, we rule out a message bus architecture and assume that the choice has been made for the resource abstraction with request/response communication between user agents and servers. These constraints relate to RMM level 2[8], which roughly means that we regard things as resources and identify the resources with URIs. Communication on RMM level 2 is done using HTTP requests preferring HTTP methods that match the kind of communication act (e. g. reading with GET and updating with PUT, instead of using POST for both). These constraints are thoroughly understood and are well-documented [6, 1].

In particular, the resource and state manipulation abstractions behind URIs and HTTP provide constraints that limit the degree of freedom for providing and accessing component interfaces. The decision for using web architecture goes a long way towards cost-effective interoperation, however, even within web architecture, there are many degrees of freedom, which leads to mismatches concerning data format and data semantics that still have to be addressed.

3.2 Dimensions

In the following, we list the dimensions on which one can make a choice regarding how the interface looks and behaves. Remember that the goal is to reduce the many interface variations that make building applications cumbersome.

- Network interface: assuming a REST-based protocol such as HTTP/1.1, HTTP/2 or COAP, the choice is between supporting read, update, delete, create and observe operations. In addition, one could assume a general query interface, as for example database mediator systems assume (albeit only on read operations).

- Message semantics: 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

[8] `http://martinfowler.com/articles/richardsonMaturityModel.html`

transmitting patch instructions or arbitrary commands. Although it is conceivable to just send values or objects in some syntax, we assume a more elaborate approach for encoding messages.

- Knowledge representation: assuming an RDF-based knowledge representation format, we could layer ontology languages with progressively more expressive power on top, starting with RDFS, moving to OWL LD and then to the more expressive OWL 2 profiles.
- Interface descriptions: starting with no dedicated interface descriptions and just assuming the HTTP (or COAP) semantics for state manipulation, 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.

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. Again, the reason that we go for a minimal component interface is the ability to specify application behaviour on distributed components in a simple and formally clean way based on mathematical logic.

3.3 Constraints on the Dimensions

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

1. Network interface: we assume a REST-based abstraction, where each component provides resources that are identified via URIs. Components provide an HTTP server connector and allow for read (GET) and write (PUT) operations on the provided resources using HTTP.
2. Message semantics: we assume information about the state of resources to be transferred in successful GET and PUT requests.
3. 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 an entry point, and that the index resource links to other resources on the same component.
4. Interface descriptions: we do not assume any interface descriptions, but simply require that the interfaces actually follow the HTTP semantics[9]. We briefly return to the question of interface descriptions in Section 6.2.

In summary, for the uniform access to components, we assume an interface adhering to the Linked Data principles. Due to the fact that Linked Data is read-only, that is, only supports HTTP GET, we extend

[9]`https://tools.ietf.org/html/rfc7231`

the interface to include write capabilities with HTTP PUT. Read-Write Linked Data[10] and the Linked Data Platform specification[11] explain how to support full CRUD operations in conjunction with Linked Data.

Please note that the constraints are not minimal, that is, component providers are free to include support for additional features, for example HTTP POST/DELETE, HTTP/2 server push or COAP observe, or expressive OWL 2 knowledge representation. However, each additional feature of the server components requires support on the side of user agents that also need to support the server features, which complicates the development of user agents.

4 Resolving Mismatches

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

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

4.1 Mapping Different Protocols

Because not all components implement the constrained interface, we require wrappers (a.k.a. shims, administration shells, lifting/lowering specifications) to bring the components to the same interface. Next to syntax and semantics of resource representations (covered in the following sections), we might need to map fundamentally different networking protocol paradigms.

The constrained interface assumes an HTTP server connector, with the components being passive and waiting for incoming requests. Some networking environments assume active components, i.e., components that emit data at intervals, which would in the HTTP world require a client connector. 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 (i.e., as server). 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 passed on to the final destination.

We have implemented such a protocol mapping between the Robot Operating System[12] (ROS) message bus and our Read-Write Linked Data interface abstraction [9]. The potential drawback is that polling is out of sync with the arrival of events and that the application might, therefore, miss state changes. Also, increased latency can occur due to a polling frequency that is out of sync with the update frequency.

4.2 Mapping Different Representation Syntax

Even if all components provide access to resources via an HTTP server connector, the representation of resource state might still differ (for example, binary formats, CSV, TSV, JSON or XML). Hence, the wrapper 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.

[10] `https://www.w3.org/DesignIssues/ReadWriteLinkedData.html`
[11] `https://www.w3.org/TR/ldp/`
[12] `http://www.ros.org/`

4.3 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[13].

4.3.1 Different Terminology

If the conceptual models (concerning the resources) of different vocabulary terms are similar, then different terms can be easily mapped using RDF-based technologies. For example, the resource "Thing" of the Thing Description (TD) ontology[14] could be mapped to the resource "System" of the Semantic Sensor Network (SSN) ontology[15] using the following triple:

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

An RDF processor that knows about the RDFS semantics would then draw the right conclusions and resolve the different terminology transparently for the user. Instead of using RDFS terms with associated semantics, we could also assume a derivation rule encoded in Notation3 syntax[16] to partially express the "subclass" relation:

```
{ ?x rdf:type td:Thing . } => { ?x rdf:type ssn:System . } .
```

Intuitively[17], the rule states that every instance of `td:Thing` is also an instance of `ssn:System`. Please note that the rule only partially encodes the semantics of the "subclass" relation; missing is transitivity, which we would have to encode in a separate derivation rule.

Mapping different terminology requires a similar conceptual view. We cover diverging conceptual views in the following.

4.3.2 Different Modelling Granularity

Modelling granularity refers to the scope of resources. Consider two components that use the concept of a city. Assume that one component uses a resource to identify the city of Berlin, whereas another component uses a resource to identify the metropolitan region of Berlin. For some cases, it would be ok to equate the city of Berlin with the metropolitan region of Berlin, while for other cases such a mapping would be wrong. As the different components use different modelling granularity, one has to careful when mapping resources.

4.3.3 Different Assumptions about Aspect and Time

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

Yet another mismatch can occur if ontologies have been modelled based on different assumptions. For example, an implicit assumption behind the SOSA (Sensor, Observation, Sample, and Actuator) ontology

[13]The examples assume prefix declarations as defined at `http://prefix.cc/`.
[14]`https://w3c.github.io/wot-thing-description/`
[15]`https://w3c.github.io/sdw/ssn/`
[16]`https://www.w3.org/DesignIssues/Notation3.html`
[17]For a quick introduction to derivation rules see `http://n3.restdesc.org/`.

(which is part of an updated version of SSN) is that users want to represent a journal of sensing and actuating activities. SOSA includes a `sosa:Observation` and a `sosa:Actuation` class to represent results from past observation and actuation events. Triggering observation or actuation events within a state manipulation architecture such as RMM level 2 is not straightforward. The WoT TD ontology, on the other hand, has a more immediate state-based view. For example, a temperature reading in TD could be done via accessing the current state of a thermometer, whereas in SOSA an observation has to be triggered somehow (e.g., via a service call), so that the value of the observation then can be read. State representations in TD include the "writable" flag, which indicates whether a representation (such as the state of a switch) can be written. In SOSA, actually carrying out actuation is outside the modelling (again, one could assume a service call of some sort).

5 Applications Using Uniform Component Interfaces

We finally arrive at the topic of developing applications that require access to many different components. 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, to be able to specify application behaviour over different networked components using high-level executable specifications.

In the following, we first discuss how to access components using imperative programming languages, and then outline the first step towards high-level descriptions of application behaviour with executable specifications of simple reflex agents on Linked Data.

5.1 Writing Applications in Imperative Programming Languages

A uniform interface to components simplifies the development of applications in imperative programming languages. Some people in our projects access the component state with code in imperative programming languages, as they were not comfortable with specifying applications using rules.

When dealing with RDF messages within imperative programming languages, developers need to generate and handle messages as objects (with static typing) in the programming language. To that end, validation of incoming and outgoing messages for serialisation and deserialisation was not possible, due to the open world assumption in RDFS and OWL. Hence, the developers augmented the existing vocabulary descriptions in RDFS and OWL with descriptions in SHACL[18] to specify request (input) and response (output) message bodies. Based on the SHACL descriptions, the programmers could then validate messages and thus make sure that the RDF messages contain all fields required for parsing into objects in typed programming languages such as C/C++ or Java.

5.2 Writing Applications in a Rule-based Language

We now present an approach for specifying rule-based user agents operating on components via the uniform interface (extending earlier work [12]). In the following, we present an application model following the outlined constraints. 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 the outgoing HTTP request (an action), and a server receives the incoming HTTP request (an event) [3].

[18]`https://www.w3.org/TR/shacl/`

In terms of agent architectures, we assume simple reflex agents. Straightforward should be an extension to model-based reflex agents that know about the semantics of HTTP operations.

We limit the number of active components per application to one, similar to a composition in web services; the active component is an HTTP user agent that polls resource state and issues unsafe requests where applicable. 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 connector to the controlling component in an application. However, the same argument regarding reduced complexity applies.

Applications could be seen as simple reflex agents with behaviour specified in condition-request 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-request 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-request 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 user agents in two modes: first, in time-triggered mode, in which the sense-act cycle runs at specified times; second, in event-triggered mode, in which the sense-act cycle runs whenever a specified event (incoming request) has taken place. Given that our agents only use a client connector, the agents cannot receive events (which would require a server connector) and hence all our user agents currently are time-triggered. We touch on an extension of the syntax and semantics that allows for rule-based specifications to take into account incoming requests in Section 6.

5.3 Safe Requests and Link Following (Sense)

To have resource state available locally, a user agent application has to specify some initial resources that form the basis for further processing. For example, the following two RDF triples encode an 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-request rules encoded in Notation3 syntax with request templates using the HTTP vocabulary[19] in the rule heads. 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 . } .
```

[19]https://www.w3.org/TR/HTTP-in-RDF10/

Such rules allow for specifying that certain links to URIs should be followed. Please note that the semantics of request rules is different to the semantics of derivation rules. With derivation rules, we assume that the graph pattern in the rule head is used to create new triples that are added to the knowledge base. With request rules, we assume that the request template in the rule head is used to create new HTTP requests that are executed, yielding HTTP responses with a response body that is parsed and added to the knowledge base.

URI templates are another way to specify links. Assume a service that returns triples with nearby locations, given the URI, latitude and longitude of a location. We can write the following request rule:

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

The rule instructs the interpreter to access the service for each location (with latitude and longitude) in the dataset. Such request rules with URI templates are a variant of Linked Data Service (LIDS) descriptions [11].

The evaluation of condition-request rules 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-request rules are evaluated. The interpreter applies these condition-request 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 encoded in derivation rules. 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.

5.4 Unsafe Requests (Act)

For a user agent to be able 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 { [] :state :Off . } .
} .
```

The rule instructs the interpreter to switch off the heating in case the temperature is above 20. As our immediate goal is to provide high-level executable specifications based on rules, we assume that people who write rules know both the uniform interface to the components and the required payload and hence do not need component 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.

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.

6 Future Directions

We now touch on future topics. We briefly sketch a model of computation for our architecture and finally we discuss the relation between the reflex agents (condition-request rule-based user agents) and goal-based agents.

6.1 Towards A Uniform Model of Computation

For a formal grounding for our rule-based user agent specifications, we seek a model of computation to quantify the expressive power of our approach, and to formally align the application architecture with other behaviour description works such as programming and workflow languages. In theoretical computer science, the notion of Abstract State Machines [2] has been developed as a formal approach to specify semantics in the context of computation. Abstract State Machines are defined using model theoretic structures, where the interpretation of symbols changes over time governed by transitions given in rules. Hence, Abstract State Machines is concerned with the dynamics of interpretations. In RDF model theory however, static structures are in the focus, with elaborate semantic conditions on the interpretations. Currently, we are investigating the combination of technologies in our architecture, namely of RDF, Linked Data, and condition-request rules on the one side, with the theoretical foundation in Abstract State Machines on the other side, consists in network-accessible data using elaborate knowledge representation with a Turing-complete model of computation.

6.2 The High Cost of Goal-based Agents

The specifications of user agent behaviour in rules could also be automatically generated, instead of getting manually crafted. For instance, AI planning or techniques based on mathematical proofs make use of goal specifications and elaborate Input/Output/Precondition/Effect (IOPE) descriptions[20] of the components to derive user agent behaviour.

In earlier versions of our prototypes, we have included descriptions of the input and output of services. With LIDS [11], we described the required input parameters for GET requests (encoded in the URI), and the graph shape of the outputs of the corresponding response. However, when specifying user agents that operated on the components, we wrote rules that directly encoded 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 put in the effort to also change the descriptions.

[20]https://scholar.google.com/scholar?hl=en&q=iope+descriptions

When starting with specifications that are immediately executable, the users can more quickly create applications, without having to spend significant effort for providing descriptions. Descriptions could be provided later. In a sense, one can find very basic descriptions in the request rules. Both safe and unsafe requests in our examples contain descriptions (which parameters/payloads to supply) in the rule heads as part of the request. A description of the parameters is in the rule body (the condition part). With enough applications specifying request rules, one could extract descriptions from request rules.

Approaches such as RESTdesc [13] provide a way to encode component descriptions in N3 syntax, and, in conjunction with a suitable reasoner and a goal description, can execute user agent behaviour. In the case of RESTdesc, the user agent specification is an HTTP request to be executed immediately and a list of possible HTTP requests that could be executed in the future. Superficially, RESTdesc rules and condition-request rules look the same, given both are given in N3 syntax and use the HTTP vocabulary in the head of rules. But they serve opposing purposes: while RESTdesc rules describe the potential behaviour of components, the condition-request define the actual behaviour of user agents.

RESTdesc rules follow the form `{ precondition } => { request effect } .`, where the precondition states under what circumstances a user agent may want to issue the defined request that in turn leads to the described effect. A more classical way to describe the behaviour of server components, closer to IOPE, would be with rules in the form `{ precondition request } => { response effect } .`, where `precondition` and `effect` refer to the state of resources on the server, `request` refers to the incoming HTTP request (an event), and `response` refers to the HTTP response the server issues. Such precondition/request-response/effect rules would directly lead to executable specifications for server behaviour, analogous to how the condition-request rules lead to executable specifications of user agent behaviour.

7 Conclusion

Data integration and system interoperation are complex problems. We believe that in order to solve at least some of the problems, there have to be restrictions in place that – some would say severely – constrain the interfaces of the components that should interoperate. The idea is to keep things simple. We have presented a system architecture that provides integrated access to networked decentralised components following a constrained interface in conjunction with a rule-based condition-request language to access resource state, integrate data and specify behaviour. We have applied the system architecture to geospatial data integration [5] and industrial applications around product design and validation [8][7][4].

The presented system architecture unifies network protocol, knowledge representation and agent architectures. While each of the parts provide very powerful features, the synthesis requires a reduction of 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 receive real-time push updates encoded in an expressive OWL2 profile from components, our approach was to try to identify a minimally viable architecture that provides execution as an immediate payoff. We believe that our architecture and rule-language can represent a clean foundation, on top of which more elaborate functionality can be layered, such as automated composition, workflows and formal verification.

References

[1] R. Fielding. *Architectural Styles and the Design of Network-based Software Architectures.* PhD thesis, 2000. `https://www.ics.uci.edu/~fielding/pubs/dissertation/rest_arch_style.htm`.

[2] Y. Gurevich. Abstract State Machines: An Overview of the Project. In *Proceedings of the Third International Symposium on Foundations of Information and Knowledge Systems*, pages 6–13, 2004.

[3] A. Harth and T. Käfer. Towards specification and execution of Linked Systems. In *Proceedings of the 28th GI-Workshop Grundlagen von Datenbanken (GvD)*, pages 62–67, 2016.

[4] A. Harth, T. Käfer, F. L. Keppmann, D. Rubinstein, R. Schubotz, and C. Vogelgesang. Flexible industrielle VT-Anwendungen auf Basis von Webtechnologien. In *VDE Kongress 2016, Internet der Dinge*, 2016.

[5] A. Harth, C. A. Knoblock, S. Stadtmüller, R. Studer, and P. A. Szekely. On-the-fly integration of static and dynamic sources. In *Proceedings of the Fourth International Workshop on Consuming Linked Data*, 2013.

[6] I. Jacobs and N. Walsh. Architecture of the World Wide Web, Volume One. Recommendation, W3C, Dec. 2004. `http://www.w3.org/TR/webarch/`.

[7] T. Käfer, A. Harth, and S. Mamessier. Towards declarative programming and querying in a distributed Cyber-Physical System: The i-VISION case. In *Proceedings of the 2nd International Workshop on Modelling, Analysis, and Control of Complex CPS (CPSData) at the 9th CPS week*, pages 1–6, 2016.

[8] F. L. Keppmann, T. Käfer, S. Stadtmüller, R. Schubotz, and A. Harth. Integrating highly dynamic RESTful Linked Data APIs in a Virtual Reality environment (demo). In *Proceedings of the 14th International Symposium on Mixed and Augmented Reality (ISMAR)*, pages 347–348, 2014.

[9] F. L. Keppmann, M. Maleshkova, and A. Harth. Building rest apis for the robot operating system - mapping concepts and interaction. In *Proceedings of the Workshop on Services and Applications over Linked APIs and Data*, 2015.

[10] F. L. Keppmann, M. Maleshkova, and A. Harth. Towards optimising the data flow in distributed applications. In *Proceedings of the Workshop on Web APIs and RESTful Design (WS-REST)*, 2015.

[11] S. Speiser and A. Harth. Integrating Linked Data and services with Linked Data Services. In *Proceedings of 8th Extended Semantic Web Conference (ESWC)*, 2011.

[12] S. Stadtmüller, S. Speiser, A. Harth, and R. Studer. Data-fu: A language and an interpreter for interaction with Read/Write Linked Data. In *Proceedings of the 22nd International Conference on World Wide Web (WWW)*, pages 1225–1236, 2013. Please note that we have renamed the system to "Linked Data-Fu" to avoid name clashes with other projects.

[13] R. Verborgh, T. Steiner, D. Van Deursen, S. Coppens, J. G. Vallés, and R. Van de Walle. Functional descriptions as the bridge between hypermedia apis and the semantic web. In *Proceedings of the Third International Workshop on RESTful Design*, pages 33–40, 2012.

[14] G. Wiederhold. Mediators in the architecture of future information systems. *Computer*, 25(3):38–49, Mar. 1992.