Towards Specification and Execution of Linked Systems

Andreas Harth
Institute AIFB
Karlsruhe Institute of Technology (KIT)
Kaiserstr.12, 76131 Karlsruhe, Germany
harth@kit.edu

Tobias Käfer
Institute AIFB
Karlsruhe Institute of Technology (KIT)
Kaiserstr.12, 76131 Karlsruhe, Germany
tobias.kaefer@kit.edu

ABSTRACT

We introduce the formalism of Linked Systems for specifying and executing dynamical systems that operate over Read-Write Linked Data. Linked Systems cover user agents (components that emit HTTP requests) and servers (components that receive HTTP requests). The formalisation is inspired by automata theory and the concepts of state transition systems and state machines. For the proposed formalism to scale to the web, we try to minimise the burden on component providers. We thus assume a Read-Write Linked Data interface that offers only a few operations: resources identified via URIs can be created and deleted, and the state of resources is expressed in RDF and can be read and updated. Our near-term goal is to provide executable specifications of autonomous behaviour expressed in a rule-based language, without requiring formal service descriptions of the operations on resources. In the long term, we plan to use our formalism as basis for applying techniques from the fields of formal verification and artificial intelligence (AI) planning.

1. INTRODUCTION

Users can gain access to arbitrary content and functionality on the web with a browser in one unified user interface. On mobile devices, however, the dominant user interface to content and functionality are apps. Each tailored app communicates with a backend server, often via web protocols. Universal user interfaces are desirable so that users can comfortably access any network-accessible component or combinations of components. Potential future universal user interfaces could be dialogue systems, such as chatbots and virtual assistants, or virtual reality environments.

A fundamental research challenge has been to find the appropriate abstractions for interfaces to networked components to facilitate the interaction with components and enable the interoperation between them. Realising the overall goal of enabling flexible access to content and functionality from a large number of sources requires at least the following from component providers:

1. A uniform protocol to decentralised components, to be able to manipulate and interact with components without writing adaptors.

2. A uniform data model, a suitable knowledge representation language and an associated query language, to be able to represent, integrate and query data on a global scale.

3. A uniform description of the behaviour of components, to be able to apply techniques from the fields of formal verification and AI planning.

With these uniform interfaces we could specify and execute systems that operate autonomously; perform static analysis, verification and simulation of these systems; and automatically generate these systems, given component descriptions and a goal. While these uniform interfaces been available in closed systems (for example, in the field of artificial intelligence), such a level of elaboration has been elusive on the web, which is an open decentralised system with many contributors. Different communities, however, work towards providing (parts of) elaborate uniform interfaces in open environments:

1. The followers of the architectural style Representational State Transfer (REST) [7] encourage people to use the abstraction of resources, on which a constrained set of operations (e.g. reading and writing, creating and deleting) can be executed. Hyperlinks provide connections between resources that provide data and functionality. The Richardson Maturity Model (RMM)[1] introduces different levels of adherence to the REST ideas. RESTful interfaces, however, do not require a certain syntax for the representation of resource state.

2. The Linked Data community follows a different (but similar) set of principles [3] to publish and access data on the web that mandate the Resource Description Framework (RDF) as a graph-structured data model for representing resource state. The Linked Data principles, however, only cover read-access to resource state. The combination of the higher RMM levels and Linked Data leads to Read-Write Linked Data [4], in which resources are linked and resource state represented in RDF can be manipulated.

28^{th} GI-Workshop on Foundations of Databases (Grundlagen von Datenbanken), 24.05.2016 – 27.05.2016, Nörten-Hardenberg, Germany.

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

3. Finally, different communities concerned with dynamical systems (for instance, control systems and cyber-physical systems) abstract from specific protocols and often use a simple data model. Formal descriptions of services or actions – the "laws" of a system – are central to the applied verification methods, such as model checking or simulation. The Semantic Web community had attempted to create a respectable amount of fully-described services, and the REST community is working towards having standardised input and output descriptions for APIs widely available.

Currently, we cannot assume that elaborate uniform interfaces to components are commonplace on the open web. Today, many APIs provide a resource-oriented modelling with constrained operations (REST). Linked Data is popular for read-access to RDF data. Read-Write Linked Data, as the combination of the two popular paradigms, could provide access to a sizable amount of components with a limited additional burden for component providers, for example as specified in the W3C Linked Data Platform recommendation[2] with editors from IBM and Oracle.

Thus, in this paper, we focus on manual specification of systems that operate on components which have a Read-Write Linked Data interface, but do not provide formal service descriptions. The goal is to have a formal model of systems that are able to use and provide Read-Write Linked Data resources. We present an approach for executable formal specifications. The approach can be extended to support formal verification and AI planning in case a sufficient amount of components with formal service descriptions should become available.

We have applied prototypes that implement the proposed formalism for specifying interactive systems based on a Read-Write Linked Data interface in several settings. In the German ARVIDA project[3], we break up monolithic industrial Virtual and Augmented Reality (VR and AR) systems into components with a Read-Write Linked Data interface, and specify and execute VR applications based on these interfaces. In the European i-VISION project[4], we provide means to connect a workflow analysis software with a flight simulator to perform Human Factors analysis of aircraft cockpits in a VR environment at run-time; the components are also based on Read-Write Linked Data interfaces. We have demonstrated that the execution of the system specifications achieved update rates sufficient for an immersive experience in the VR environment [11, 12].

We begin the remainder of the paper with related work in Section 2, introduce necessary definitions in Section 3, formalise the notion of a Linked System in Section 4, and conclude with a summary and a list of open issues in Section 5.

2. RELATED WORK

Dynamical systems are fundamental to many fields. Fields concerned with the "real" world – such as physics, control theory, and cyber-physical systems – study primarily continuous-time systems. In the digital world of computation, discrete-time systems are common, with a monotonically increasing sequence of integers denoting time (often written as t). Given the enormous breadth of dynamical systems, we can only provide a subjective selection of related work of the discrete-time variants.

In computational systems, Harel and Pneuli [9] introduce the dichotomy between a transformational system "that accepts inputs, performs transformations on them and produces output" and reactive systems that in contrast "are repeatedly prompted by the outside world and their role is to continuously respond to external inputs". The goal of our work is to provide a formalism to specify reactive systems[5] that combine both user agents (systems that emit requests) and servers (systems that accept requests) into a single representation.

Formal models are popular for describing reactive systems in closed environments [5, 2, 15, 1]. These approaches require full descriptions of the "laws" of the systems, to be able to generate the entire state space for a given vocabulary if the systems have finite domains, or to use simulation for infinite domains. However, often the data representation is limited: either variables with values in the case of the cyber-physical systems community, or propositional logic in the case of the model checking and automata communities.

McCarthy's situation calculus is an early formalism to describe dynamical systems in the area of artificial intelligence. The situation calculus includes actions that can be performed in the world and fluents that describe the (changing) state of the world. As we conceptually operate on a single ternary *triple* predicate for representing state, we do not have the possibility to identify fluents (predicates that change over time). Also, we assume the open web as task environment rather than a closed system, and distinguish between user agents and servers.

Abstract State Machines (ASMs) [8] use first-order structures (functions and relations) over a fixed vocabulary to represent state. ASMs heavily inspired our approach, but our formalism is targeted for the web: we use RDF instead of first-order structures. We further do not make the assumption of a fixed vocabulary, as on the web we want to follow links during runtime, which may lead to hitherto unknown URIs. Instead of arbitrary external functions to interact with the environment, we limit the external functions to HTTP requests with create-read-update-delete (CRUD) methods.

Approaches for Web Service Composition [13] are based on XML-based WS-* standards, such as WSDL for interface description to remote procedure calls [14]. We use a resource-based abstraction with CRUD operations instead. These approaches use BPEL (Web Services Business Process Execution Language), an industry standard for specifying service compositions. We rather use a formal model based on state transition systems as the foundation.

The Guard-Stage-Milestone (GSM) framework for artifact-centric workflows [6] provides a business process view on dynamical systems. Instead of a fixed data schema (that is, the information model as attribute/value pairs) partitioned into data (static) and status attributes (fluent), we use the semistructured RDF triples data model without the distinction of fixed and fluent partitions. GSM systems operate on incoming events and outgoing events. In contrast to GSM and also our earlier work [16], we distinguish between ac-

[2] `http://www.w3.org/TR/ldp`
[3] `http://www.arvida.de/`
[4] `http://www.ivision-project.eu/`

[5] Or interactive systems, that in contrast to newer definitions of reactive systems do not have the strict timing requirement to react at the pace of the environment.

tions and events, and user agents and servers. While GSM assumes a fixed representation of different steps and milestones in the system, we do not assume a fixed model but provide the possibility to model such representations in RDF triples. In GSM, updates to the world state are visible immediately. In our model, updates to the world state are only visible in the next step once a new sensing round accessing the resource state has been carried out.

Semantic Web Services [17] assume a fully described task environment, and use expressive first-order logic, similar to ASMs, to represent the state. In Semantic Web Services, AI planning based on descriptions is central for combining arbitrary network-accessible functions (via SOAP) into an executable plan. Instead of arbitrary functions, we assume a resource-oriented CRUD interface. We share the long-term vision of Semantic Web Services and Agents in the Semantic Web [10]. But we focus on the part that we feel is widely deployable today, namely specifying and executing systems combining comparably simple Read-Write Linked Data components without descriptions.

3. PRELIMINARIES

In the following, we provide definitions for web interfaces based on a common access protocol and a common knowledge representation language. We assume Read-Write Linked Data [4], which informally can be alternatively viewed as the combination of Linked Data [3] and HTTP CRUD operations, or as Web APIs on an RMM level of at least 2 that serve RDF in one of its various syntaxes.

3.1 Resources and URIs

The following definitions build on the notions of a resource a URI, an identifier for a resource.

Definition 1 (Resource, URI). *A resource is an abstract notion for things of discourse, be they abstract or concrete, physical or virtual (e.g., a document on the web, a car, or the set of the natural numbers). A Uniform Resource Identifier (URI) is a character string that identifies a resource.*

A URI has a scheme, which is the beginning of the character string until the first colon. The scheme specifies how to interpret the rest of the URI. We focus on the `http` scheme (the considerations analogously apply to the `https` scheme). Moreover, we provide an analogy for `http` URIs with the scheme `file`. Collections (e. g. lists, sets) are a particular kind of resources, treated later.

3.2 Hypertext Transfer Protocol

While a URI with the scheme `http` first is to identify a resource, the scheme also indicates that second we may be able to interact with the resource using the Hypertext Transfer Protocol (HTTP)[6]. HTTP is the prototypical variant of REST.

Definition 2 (HTTP Request, HTTP Response). *A HTTP message is a tuple $\langle S, H, B\rangle$, where S is the mandatory start line, H is an optional list of header name/value pairs, and B is the message body, also optional. A HTTP request is a HTTP message in which the start line S consists of a request line (with the HTTP method and the request URI and the version information). In a HTTP response message, the start line S consists of a HTTP status code and information on the used HTTP version.*

Interaction with HTTP URIs is done in request/response pairs. The message body B of a HTTP message contains a representation of the current state of the resource identified via the request URI in the start line. Representations of state can vary over time, just like the resource can change.

A HTTP request has a HTTP method in its status line. The HTTP methods include `GET`, `PUT`, `POST`, and `DELETE`. Less popular methods include `PATCH` and `OPTIONS`[7]. There exists a rough mapping from the HTTP methods to the CRUD operations, create, read, update, delete – the basic operations for persistent storage, which we present as we describe the HTTP methods. We call methods that are free of side effects "safe". The safe HTTP methods are:

- A `GET` request retrieves the representation of the current resource state (corresponding to "read" in CRUD).
- An `OPTIONS` request results in the methods that are allowed on a resource. The body of response messages to `OPTIONS` requests could contain formal service descriptions in a possible extension of our approach. We currently do not use `OPTIONS`.

Methods that change the state of a resource are called "unsafe" (e.g., `PUT`, `POST`, `DELETE`, `PATCH`). The unsafe requests (those are for "create", "update" and "delete" in CRUD) are:

- A `PUT` request with message body b assigns b as the representation of the resource state. `PUT` can be used to create a resource with a URI set by the client.
- A `PATCH` request with body b updates a resource representation (remove and add data) based on the sent patch specification b.
- A `POST` request can serve different purposes:
  - append data to an existing representation of a resource;
  - create a new resource with a URI determined by the server; or
  - perform RPC-style arbitrary data-processing (not possible on `file` URIs).

  The `POST` request allows for data processing in a remote procedure call (RPC) fashion. Therefore, `POST` allows for invoking arbitrary operations (functions). To benefit from the uniform interface of REST, in this paper we only consider state updates, in-line with RMM level 2.
- A `DELETE` request removes a resource.

If multiple applications of the same method yield the same result, we call these methods "idempotent" (e.g., `GET` because it is safe, `PUT` because if the resource state is overwritten multiple times with the same data, the resulting resource state is still the same).

Using the HTTP methods, we operate with resources identified by URIs with the `http` scheme. Alternatively, we

[6] `http://tools.ietf.org/html/rfc7230`

[7] `http://tools.ietf.org/html/rfc7231`

can define the HTTP methods also for URIs with the `file` scheme, i.e. URIs that identify files and directories. For example, `GET` on such a URI would retrieve the content of the file; `PUT` on a `file` URI would create the file with the payload; `PUT` on a `file` URI ending in a slash character would create a directory.

3.3 Resource Description Framework

As indicated by the Linked Data principles, we assume that the representation of the state of resources is given using the Resource Description Framework (RDF) in an RDF graph.

Definition 3 (RDF Term, Triple, Graph). *The set of RDF terms consists of the set of URIs U, the set of blank nodes B and the set of RDF literals L, all being pairwise disjoint. A tuple $\langle s, p, o\rangle \in (U \cup B) \times U \times (U \cup B \cup L)$ is called an RDF triple, where s is the subject, p is the predicate, and o is the object of the triple. A set of triples is called RDF graph.*

To be able to talk about the state representations retrieved from multiple resources, i.e. multiple RDF graphs, we introduce the notion of an RDF dataset.

Definition 4 (Named Graph, RDF Dataset). *Let G be the set of RDF graphs and let U be the set of URIs. A pair $\langle g, u\rangle \in G \times U$ is called a named graph. An RDF dataset consists of a (possibly empty) set of named graphs (with distinct names) and a default graph $g \in G$ without a name.*

To talk about the state of resources at different times, we introduce an index t, which denotes the point in time to which an RDF dataset D refers, thus yielding D_t. We assume discrete time represented as monotonically increasing integers.

4. LINKED SYSTEMS

In the following, we first introduce the general notion for dynamics in Linked Data which we call Linked Data Transition System. We then define user agents and servers, and next outline Linked Systems.

The Linked Data Platform (LDP) recommendation poses restrictions on how to interact with web resources, particularly collection resources, in a Linked Data context. Our Read-Write Linked Data interface as described in the following definitions adheres to LDP where it concerns HTTP-based interaction with collection resources. We omit the various header fields that are part of LDP.

On the web, we operate in a decentralised system in which we cannot control each participant. REST provides a limited set of constraints on the components that allow users to assume a certain behaviour. The use of resource-based CRUD operations is one such constraint. However, certain behaviours are under-specified. For instance, if we overwrite a resource state with `PUT`, the resulting state of the resource may differ from what has been sent in the message body of the `PUT` request. Also, changing the state of resource A may affect the state of resource B. In our definitions of the semantics of HTTP operations that follow, we allow such side effects only to occur between collection resources and element resources.

4.1 Semantics of HTTP Operations

Let S be an RDF dataset. We write S_t for RDF dataset S at time $_t$. With `GET`, we are able to obtain the resource state, so that we can process the resource state as part of our world model. Let u be a URI from U identifying not a collection resource. Let r a request/response pair involving u which relates to a named graph in S.

We contrast the current state t of an RDF dataset S before the request/response pair r with the state of S at $t+1$ after the request/response pair r. Now we can formally define how `PUT`, `POST` and `DELETE` behave. We denote a "don't-care" as a dot ("·").

- `PUT`: Let r be a request/response pair $\langle$`PUT` $u, \cdot, B\rangle$, $\langle$`200 OK`$, \cdot, \cdot\rangle$. Applying r to S_t yields S_{t+1}, where in S_{t+1} the triples belonging to u are B.
- `POST`: Let r be a request/response pair $\langle$`POST` $u, \cdot, B\rangle$, $\langle$`200 OK`$, \cdot, \cdot\rangle$. Applying r to S_t yields S_{t+1}, where in S_{t+1} there are additional triples (related to those in B, if $B \neq \emptyset$) related to u.
- `DELETE`: Let r be a request/response pair $\langle$`DELETE` $u, \cdot, \cdot\rangle$, $\langle$`204 No Content`$, \cdot, \cdot\rangle$. Applying r to S_t yields S_{t+1}, where in S_{t+1} there is no representation available at u.

Now, let uc be a URI from U identifying a collection resource.

- `PUT`: `PUT` is not possible on collection resources (we assume collections are managed by the server). LDP does not require the `PUT` request to be supported in the context of collections either.
- `POST`: Let r be a request/response pair $\langle$`POST` $uc, \cdot, B\rangle$, $\langle$`201 Created`$, H, \cdot\rangle$. Applying r to S_t yields S_{t+1}, where in S_{t+1} there are additional triples (those in B) related to a newly created[8] URI ue that is linked to uc. Also, uc is different now, as the link to the newly created resource identified via URI ue is part of the state of the collection resource. The set of headers H contains ue as the value of the `Location` header. In compliance with LDP, we assume no representation in the body of the response.
- `DELETE`: Let r be a request/response pair $\langle$`DELETE` $uc, \cdot, \cdot\rangle$, $\langle$`204 No Content`$, \cdot, \cdot\rangle$. Applying r to S_t yields S_{t+1}, where in S_{t+1} there is no representation available for uc. Whether deleting the collection resource uc also affects its associated element resources depends on the type of the relation between the collection and its elements (cf. the container types specified in LDP).

4.2 Linked Data Transition System

Having covered the semantics of one transition from t to $t+1$, we now can define a transition system that describes resource states (represented as RDF datasets) over multiple transitions. That is, we can we say what happens with the resource state at $t+1$ relative to resource state at t.

Definition 5 (Linked Data Transition System). *Let $Req, Resp$ be the set of all request/response pairs with unsafe operations. A Linked Data Transition System is a pair $(S, \rightarrow)$:*

[8]Or taken from a pre-filled reservoir of URIs, for example as defined in ASMs.

- *A set of RDF datasets S representing resource states.*
- *The transition relation $\rightarrow$ over $S \times 2^{Req,Resp} \times S$. We can contrast the current state s_t with the next state s_{t+1}, given a transition occurs. A transition consists of a set of request/response pairs.*

For Linked Data Transition Systems we assume an omniscient view, where we assume we can readily observe all request/response pairs in the system and all resource states. The states S are the states of all resources in the system. We write $s_0, s_1 \ldots s_n$ to denote datasets at different time points. We write $(s_o, r_o, s_1) \in \rightarrow$ as $s_o \xrightarrow{r_0} s_1$. We can define the history of a system by a sequence $s_o \xrightarrow{r_0} s_1 \xrightarrow{r_1} s_2 \ldots$, where s_o, s_1, s_2 are states, and r_0, r_1 are sets of request/response pairs.

Next, we introduce a limited point of view, where we distinguish the direction of requests relative to a user agent or server.

4.3 Actions and Events

We now can define the notion of a Linked Data User Agents and Linked Data Servers. A HTTP message exchange involves two parties: user agents and servers. The user agent creates the request and the server creates the response.

Definition 6 (User Agent, Action). *Let Req be a HTTP request message, and Resp be a HTTP response message. The function send$(Req, Resp)$ denotes a system emitting Req and receiving Resp. An action is an outgoing request/response pair. User agents are systems that emit actions.*

We can group the actions into safe actions (`GET`) and unsafe actions (`PUT`, `POST`, `DELETE` and `PATCH`).

Definition 7 (Server, Event). *Let Req be a HTTP request message, and Resp be a HTTP response message. The function receive$(Req, Resp)$ denotes a system receiving Req and sending Resp. An event is an incoming request/response pair. Servers are systems that receive events.*

We can group the events into safe events (`GET`) and unsafe events (`PUT`, `POST`, `DELETE` and `PATCH`).

The same request/response pair is regarded as an action in the user agent and as an event on the server. Informally, the difference between user agents and servers is similar to the difference between generators and recognisers in state machines.

4.4 Linked Systems

We can now define the notion of a Linked System.

Definition 8 (Linked System). *Let A be the set of all (outgoing) unsafe actions and let E be the set of all (incoming) unsafe events. A Linked System consists of the quadruple $(S, s_0, F, \rightarrow)$:*

- *A set of RDF datasets S representing resource states.*
- *The initial state, $s_0 \in S$.*
- *A set of the final states, $F \subseteq S$.*
- *The transition relation $\rightarrow$ over $S \times 2^E \times 2^A \times S$. The transition consists of a set of events from E and a set of actions from A.*

Linked Systems cover both user agents and servers. In case E is the empty set, we arrive at the special case of a Linked Data User Agent. In case A is the empty set, we arrive at the special case of a Linked Data Server.

We now can define the execution of a Linked System.

Definition 9 (Run, Step). *A run of a Linked System M is a sequence of states, unsafe events, and unsafe actions. A run has multiple steps. One transition is a step. A successful run starts in s_0 and ends in a state $s_n \in F$, where n denotes the number of steps carried out during the run.*

An example of a one-step run would be $s_o \xrightarrow{e_o, a_0} s_1$, where $s_1 \in F$.

In case the Linked System is a user agent only (and thus only emits actions), we can run the system from the command line. In case the Linked System is a server, we have to provide a HTTP interface to have the ability to receive events.

We start the execution of a Linked Data User Agent from the command line. We start with the initial state s_o. As the user agent does not receive events, we successively execute steps until a final state $f \in F$ is reached. Each step t consists of the following:

- Collect the resource state s_t by dereferencing the graph names in the RDF dataset s_t. We provide link traversal specifications using rules that yield the graph names to be dereferenced in a fixpoint procedure. We can also provide deduction rules to encode different semantics (such as RDFS or subsets of OWL). The result is a materialised version of the resource state s_t.
- After computing the fixpoint to yield the overall resource state, carry out the unsafe requests as specified in the transition relation. The responses of the unsafe requests become part of s_{t+1}.
- We also provide means to register queries that are continuously evaluated on the current internal resource state. To arrive at a fully streaming model, we are only supporting SPARQL basic graph pattern queries, which can be implemented in non-blocking operators.

As the default, a new step $t + 1$ immediately starts once step t has been finished. We are able to align the start of each step at specified interval boundaries, or specify a wait time between each step. We use the time-triggered execution in our VR demonstrators at 30 Hertz (one step each 33ms).

The execution of a Linked Data Server has to take into account events from the external environment. We thus arrive at an event-triggered execution model: the system runs once the incoming request (an event) arrives. The run proceeds as in the case of Linked Data User Agents. In case the run succeeds, the response to the event includes a HTTP status code in the `2xx` range, denoting a successful run. Otherwise, a server error (HTTP status code `500`) is returned. Per default, the union of the triples in the state s_t is serialised as RDF triples in the body of the response. Optionally, we can filter the triples with a query to reduce the amount of returned data.

5. CONCLUSION

We have provided a formal account of Read-Write Linked Data as a transition system. We have identified the two roles of user agents (components that emit actions) and servers (components that receive events), and have shown how both roles can be understood in terms of the transition system. We have outlined the notion of Linked Systems, which can be instantiated in user agent or server roles, or both.

We base the presented formalism on our experiences collected with a prototype [16] that we have been using in a variety of projects. In the implementation, we represent the state transition relation in a variant of event-condition-action rules, where the condition serves as a guard that can trigger the execution of an action. We currently only implement non-blocking operators to enable stream processing. Future work includes the finalisation of the syntax for rules and a representation of traces of runs. We also consider adding support for state representations other than RDF. Our plan is to provide the system as open source.

Our formalism can serve as the basis for parallel execution. Given that Linked Systems are based on fundamental notions common to many dynamical systems, the formalism could be extended to incorporate further functionality. For example, we could add invariants, which are observed and checked during execution. In case a substantial amount of descriptions for components become available, we could use AI planning to generate the transition relation in a Linked System from the descriptions of the components, given an initial and a final state. To be able to use verification techniques, we would likely define a reduced subset of the presented Linked System, for instance providing finite domains or reducing the flexibility of the RDF triples data model.

Acknowledgements

We thank Dieter Fensel for pointing out the possible connection between REST and Abstract State Machines, Martin Junghans for explaining IOPE descriptions and process calculi, and Aidan Hogan for fruitful discussions related to dynamics in Linked Data. We acknowledge support from the BMBF ARVIDA project (FKZ 01IM13001G) and AFAP, a BMBF Software Campus project (FKZ 01IS12051).

6. REFERENCES

[1] R. Alur. *Principles of Cyber-Physical Systems*. MIT Press, 2015.

[2] C. Baier and J.-P. Katoen. *Principles of Model Checking*. MIT Press, 2008.

[3] T. Berners-Lee. Linked Data. Design Issues, 2006. `http://www.w3.org/DesignIssues/LinkedData.html`.

[4] T. Berners-Lee. Read-Write Linked Data. Design Issues, 2009. `http://www.w3.org/DesignIssues/ReadWriteLinkedData.html`.

[5] E. M. Clarke, Jr., O. Grumberg, and D. A. Peled. *Model Checking*. MIT Press, 1999.

[6] E. Damaggio, R. Hull, and R. Vaculín. On the Equivalence of Incremental and Fixpoint Semantics for Business Artifacts with Guard-Stage-Milestone Lifecycles. *Information Systems*, 38(4):561 – 584, 2013.

[7] R. T. Fielding and R. N. Taylor. Principled Design of the Modern Web Architecture. *ACM Transactions on Internet Technology (TOIT)*, 2(2):115–150, May 2002.

[8] 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. Springer, 2004.

[9] D. Harel and A. Pnueli. *Logics and Models of Concurrent Systems*, chapter On the Development of Reactive Systems, pages 477–498. Springer, 1985.

[10] J. Hendler. Agents and the Semantic Web. *IEEE Intelligent Systems*, 16(2):30–37, Mar. 2001.

[11] 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 13th IEEE International Symposium on Mixed and Augmented Reality*, pages 347–348, 2014.

[12] F. L. Keppmann, T. Käfer, S. Stadtmüller, R. Schubotz, and A. Harth. High Performance Linked Data Processing for Virtual Reality Environments. In *Poster & Demo Proceedings of the 13th International Semantic Web Conference*, 2014.

[13] N. Milanovic and M. Malek. Current Solutions for Web Service Composition. *IEEE Internet Computing*, 8(6):51–59, Nov 2004.

[14] C. Pautasso, O. Zimmermann, and F. Leymann. RESTful Web Services vs. "Big" Web Services: Making the Right Architectural Decision. In *Proceedings of the 17th International Conference on World Wide Web*, pages 805–814. ACM, 2008.

[15] K. Schneider. *Verification of Reactive Systems: Formal Methods and Algorithms*. Springer, 2004.

[16] 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*, pages 1225–1236. ACM, 2013.

[17] R. Studer, S. Grimm, and A. Abecker. *Semantic Web Services: Concepts, Technologies, and Applications*. Springer, 2007.