Taming the Interoperability Challenges of Complex IoT Systems

Paul Grace, Justan Barbosa, Brian Pickering, Mike Surridge
IT Innovation, University of Southampton, UK
IT Innovation Centre, Gamma House
Enterprise Road, Southampton SO167NS, UK
pjg@it-innovation.soton.ac.uk

ABSTRACT

The Internet of Things is characterised by extreme heterogeneity of communication protocols and data formats; hence ensuring diverse devices can interoperate with one another remains a significant challenge. Model-driven development and testing solutions have been proposed as methods to aid software developers achieve interoperability compliance in the face of this increasing complexity. However, current approaches often involve complicated and domain specific models (e.g. web services described by WSDL). In this paper, we explore a lightweight, middleware independent, model-driven development framework to help developers tame the challenges of composing IoT services that interoperate with one another. The framework is based upon two key contributions: i) patterns of interoperability behaviour, and ii) a software framework to monitor and reason about interoperability success or failure. We show using a case-study from the FI-WARE Future Internet Service domain that this interoperability framework can support non-expert developers address interoperability challenges. We also deployed tools built atop the framework and made them available in the XIFI large-scale FI-PPP test environment.

Categories and Subject Descriptors

D.2.1.12 [**Interoperability**]: Data Mapping

General Terms

Design

Keywords

Internet of Things, model-driven software engineering, interoperability, software testing, architectural patterns

1. INTRODUCTION

Interoperability remains a fundamental challenge for the developers of IoT (Internet of Things) systems and applications. Important characteristics are *heterogeneity* and scale; independently developed devices employ diverse communication protocols and data formats to interact with one another.

Where significant differences in protocols and data formats exist, how can systems be guaranteed to understand each other and interact? Standardisation and middleware are two established methods to achieve interoperability. However, given the diversity of the Internet of Things it is impossible to rely on global standards or a single common middleware platform employed by all. Instead, interoperability is typically tackled in an ad-hoc manor: i) per system/protocol compliance tests, e.g. plug tests for MQTT (MQ Telemetry Transport) implementations[1], ii) published API information for developers to follow (e.g the Hyper/Cat [1] catalogue of IoT services), and iii) development of mappings and adapters to broker system differences on a case-by-case basis (e.g. mappings between data [2], mappings between middleware [3]). While these solutions help in overcoming interoperability problems, there remains a significant burden on developers to understand and identify problems and then implement and test solutions accordingly; hence, interoperability [4] and software service testing [5] are both multi-billion dollar industries.

Model-driven software development offers a principled approach for engineering interoperable solutions through the capture of shared domain knowledge between independent developers, and automated software generation and testing. For example, model-driven testing [6] and model-based interoperability testing [7] have demonstrated the potential of the approach. However, these solutions focus on a single technology (Web Services) and require detailed models of the system's interface syntax (using WSDL) and behaviour (using BPEL) in order to generate automated tests. We propose that model-driven approaches are equally well-suited to addressing interoperability problems in the composition of IoT software; but they must consider the heterogeneity of technologies and the need for simpler quick-to-develop and highly re-usable models.

In this paper we present a Model-driven engineering framework to simplify the composition of IoT services, and support interoperability compliance and testing. For example, where a developer has created a new device to be plugged into an existing publish-subscribe monitoring application, e.g. a vehicle within an intelligent traffic management system. Such a developer can leverage the framework to ensure interoperability is correctly achieved. The framework is built

Workshop on Middleware for Context-Aware Applications in the IoT December 08-12 2014, Bordeaux, France
Copyright 2014 ACM 978-1-4503-3234-7/14/12.
http://dx.doi.org/10.1145/2676743.2676744 ...$15.00.

[1]http://iot.eclipse.org/documents/2014-04-08-MQTT-Interop-test-day-report.html

upon two core contributions:

- *Interoperability patterns* are reusable software artefacts that model the behaviour of IoT services in a lightweight and technology independent manner. These models are used to help developers create and test IoT systems that correctly interoperate. These patterns are a combination of *architecture* specification (i.e. services and interface dependencies) and *behaviour* specification (using state machines and rule-based transitions to evaluate protocol events). Importantly, the models focus only on what is required to interoperate, simplifying the complexity of the model in comparison to approaches that fully model a system's behaviour.

- *The Interoperability framework* captures systems events (REST operations, middleware messages, data transfers, etc.) and transforms them into a model specific format that can be used to evaluate and reason against required interoperability behaviour. The framework tests monitored systems against patterns to evaluate interoperability compliance, reporting where interoperability issues occur, such that the developer can pinpoint and resolve concerns.

To evaluate the framework we utilize a case study based approach. FI-WARE[2] provides a marketplace of independently developed Future Internet Services (approximately 30) that can be composed to build IoT applications; these are loosely-coupled REST services without formal interface or behavioural specifications, and hence achieving interoperability remains a significant task for developers. The FI-PPP[3] provides a large community of these developers, with a large-scale testing environment (XIFI[4]). We show how lightweight interoperability patterns can quickly be created for this domain, and also how the interoperability framework lowers the burden of performing interoperability tests and identifying the causes of interoperability errors.

The remainder of the paper is structured as follow. In Section 2, we present the model-driven interoperability framework focusing on the pattern concept for specifying required interoperability behaviour. Subsequently, we evaluate the framework in Section 3. In Section 4, we analyse the work in comparison to the state of the art; and finally in Section 5 we draw conclusions and highlight future areas of application for the solution.

2. A MODEL-DRIVEN INTEROPERABILITY FRAMEWORK

Figure 1 provides an overview of the interoperability framework; this is a set of components for performing model-driven development of interoperable software. Software engineering tools can then leverage the framework's services to support developers perform specific tasks:

- Software developers *create* new IoT applications and services to be composed with one another. Hence, they wish to engineer interoperable solutions; testing that their software interoperates with other services, and pinpoint the reasons for any interoperability errors that occur. They need to reduce the overall effort required to deliver, test and maintain correctly functioning applications. The framework goes beyond simple interface compliance testing e.g. ensuring that operations have the correct syntax and data format; instead it will also identify application behaviour and data errors e.g. data is not received by system A because system B has not published to broker, or semantic differences e.g. speed in mph and kmph.

- Domain engineers (these may be the same software developers) *model* the interoperability requirements of service compositions; that is they create **interoperability patterns** to specify how IoT applications should behave when composed: what the sequence of messages exchanged between should be (in terms of order and syntax), and what data types and content should form the exchanged information. Importantly, these models are re-usable abstractions that can be edited, shared and composed to lower the barrier towards interoperability engineering, i.e. developing and testing new IoT services that interoperate correctly.

[2]http://www.fi-ware.org/
[3]http://www.fi-ppp.eu/
[4]https://www.fi-xifi.eu

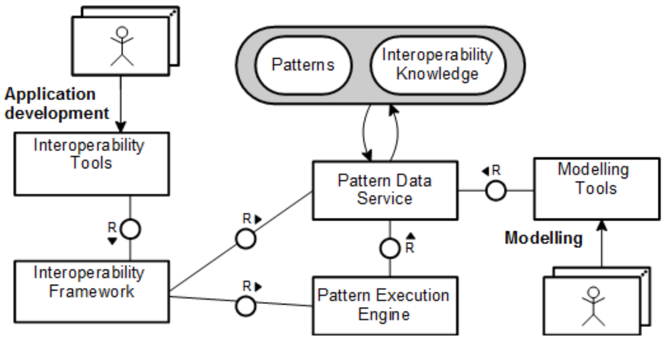

Figure 1: Model-driven interoperability engineering

Note, the software components within the framework are developed as RESTful services such that they can be used to provide a Software as a Service (SaaS) solution. However, the framework can also operate as local software e.g. an IDE plug-in. We now discuss the patterns and framework implementation in greater detail.

2.1 Interoperability Patterns

Distributed services are typically modelled using interface description languages, e.g. WSDL, WADL and IDL, to both describe the operations available and how to execute them (e.g. using a SOAP or IIOP message). These can then be complemented with workflow (e.g. BPEL) and choreography languages to explain the correct sequence of events to achieve particular behaviour. With these models it is then possible to automate the interoperability testing processes [7] and better support service composition. However, these approaches are often tied to a specific technology type e.g. Web Services or CORBA, and hence the approach is not well suited to loosely-coupled IoT services that employ a wide range of technologies and communication protocols. Furthermore, the models themselves are typically complex to write, use, and maintain; WSDL and BPEL require everything to be specified not just aspects related to interoperability. Such detail means these models are not widely deployed; this can already be seen in the Internet Services domain where RESTful APIs (e.g. Twitter, Facebook, and others) provide documentation and SDKs to help developers interoperate without IDLs.

Here, we explore models focusing solely on interoperability; that is, the specification of the exchanges between IoT services with rules defining the required behaviour for interoperability to be guaranteed. An *interoperability pattern* is specified as a finite state machine; the general format is illustrated in Figure 2. A *state* represents a state of a distributed application (not an individual service). A *transition* represents a change in state based upon an observed concrete event (e.g. a HTTP message) matching a set of rules regarding the required behaviour. Hence, the model represents the series of states that a distributed application proceeds through in reaction to discrete events (e.g. a message exchange, a user input, etc.). If the state machine proceeds such that there is a complete trace from a start state to an end state then we can conclude that software within the distributed system interoperates correctly.

If an event occurs and no transition can be made (because the event does not fulfil the rules), then the interoperability pattern identifies a failing condition. Allied with knowledge regarding why this rule failed, the tool can provide preliminary information for either correcting the error or deploying a broker solution to mediate.

In Figure 2 we present a very simple example to illustrate how a pattern is utilised. Here, we have two services (a client requesting the temperature of a room sensor, and a context service providing the sensor data) interacting with each other to complete a single request-response type operation. There are three states: i) the start state, ii) the state when the first request message is received by the sensor service, and iii) the final state where the client received a response message from the service. The interaction is a REST HTTP post operation which can contain either XML or JSON (two transition paths). A number of rules are presented to illustrate how rules are attached to transitions; each transition can specify one or more rules concerning different characteristics of events. These fall into protocol specific or data specific rules:

- *Protocol specific rules.* Evaluate events according to the structure and content of an observed protocol message (not the application/data content). For example, check the IP address of sender of the message to verify which services are interacting with each other. Further, evaluating the protocol type (HTTP, IIOP, AMQP, etc.), and the protocol message type (HTTP GET, HTTP POST or an IIOP request) to ensure that the correct protocol specification is followed. Finally, checking protocol fields (e.g. a HTTP header field exists or contains a required value) to ensure that the message contains the valid protocol content required to interoperate.

- *Application and data specific rules.* Evaluate the data content of protocol messages ensure that services interoperate in terms of their application usage. For example, the data content is of a particular type (e.g. XML or JSON), corresponds to a particular format or schema, contains a particular field unit (e.g. temperature), etc. Furthermore, rules can make constraints on the application message operations e.g. ensuring the operations required are performed in order (e.g. A sends a subscribe message to B, before C sends a publish message to B). Data rules are evaluated using data-specific expression languages; for example we leverage XPATH[5] and JSONPATH[6] tools to extract data fields and evaluate whether a given expression is true (e.g. a rule in the XPATH format: `Data[name(/*)] = queryContext`).

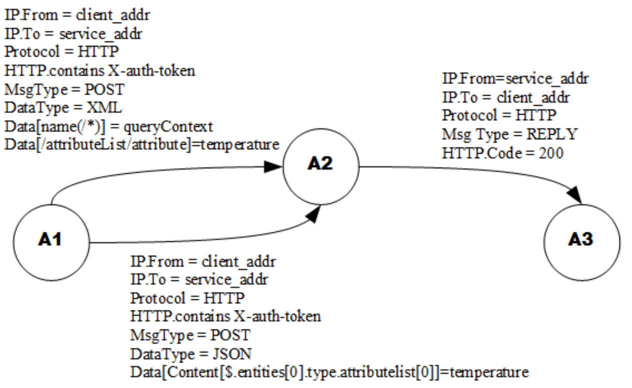

Figure 2: Interoperability Pattern

2.2 Framework Implementation

The interoperability framework has been implemented as a set of software components to monitor and evaluate running distributed applications, focusing solely on interoperability requirements. As illustrated in Figure 1, the framework contains two core elements: i) the Pattern Data service which supports operations to create and edit interoperability patterns, which are finite state machines specified in XML (we are currently developing a GUI tool to all domain modellers to perform this task graphically before generating the XML); ii) the Pattern Execution Engine, which monitors the execution of an application and evaluates it for correct interoperability.

Without going into implementation details beyond the scope of the paper, we can explain the operation of the pattern execution engine in terms of two functions:

- Monitoring deployment; the framework takes an interoperability pattern as input and generates a set of proxy elements that capture message events (these relate to all interface points in the application). Hence, if we observe that a service receives events at a particular URL, we generate a proxy to capture those events–the proxy simply reads the message content before redirecting the request to the actual service. The current implementation is built upon the RESTLET library [7]; each incoming HTTP message is transformed into an abstract representation (all fields of the HTTP message and its data content) that can be evaluated against the rules.

- Pattern evaluator; receives each event and evaluates it against the rules specified by the interoperability

[5]http://www.w3.org/TR/xpath20/
[6]https://code.google.com/p/json-path/
[7]http://restlet.com/

pattern. The pattern evaluator is protocol independent (per protocol plug-ins map concrete messages to the format understandable within the pattern). The evaluator creates a report to identify success or failure to the developer; and where a failure occurs, the framework performs simple reasoning to pinpoint the source of the error. In future work we plan to explore knowledge-based reasoners to provide richer feedback.

The framework is currently made available as a testing tool within the XIFI large-scale testing environment. XIFI establishes a pan-European, open federation comprised of 17 data-center nodes to cope with large trial deployments; there are a broad set of users and experimenters developing solutions within this FI-PPP initiative. The interoperability framework is one of a number of tools to support the development of software using the FI-WARE collection of open, restful services. A browser-based GUI tool allows this community of developers to view and edit interoperability patterns, and then directly evaluate their application software for interoperability issues.

3. EVALUATION

We used a case-study approach to evaluate the ability of the framework to achieve its primary purpose, i.e., to reduce the effort required to develop and test the interoperability of software composed with independently developed IoT services. We utilised FI-WARE software as the domain of our case study. FI-WARE is a growing catalogue of RESTFul services implementing open specifications; importantly, there are no WADL, WSDL specifications on which automated tool support can be based; instead API information is provided in free text. These services include: identity management, context brokering, big data, complex event processing, and media streaming; and have already been leveraged to build commercial IoT applications [8].

We hypothesized that the interoperability framework helps the developers of IoT applications and services during software development and testing phases; discovering problems earlier, reducing the costs, and improving the overall development of the application. We also proposed that the lightweight models offer a suitable abstraction to capture interoperability information that can be reused across multiple applications, e.g. a pattern describing how to interoperate with a context broker being utilised across multiple different applications.

Application. For this case we developed an application to monitor and gather data about traffic and transportation vehicles in Brazil to support safer and optimised payload delivery. Brazil is a location where FI-WARE software has been deployed[9]; and 56% of cargo transportation is carried out by trucks. The application monitors for vehicle and cargo theft (using context information: location, fuel levels, door opening, detour information, etc.); increasing context data collection when transiting through a high-risk area. The application also collects context data from multiple vehicles, and performs off-line analysis of delivery performance to produce optimised routing plans.

Interoperability challenges. The need to integrate multiple devices (e.g. vehicles) and FI-WARE services into

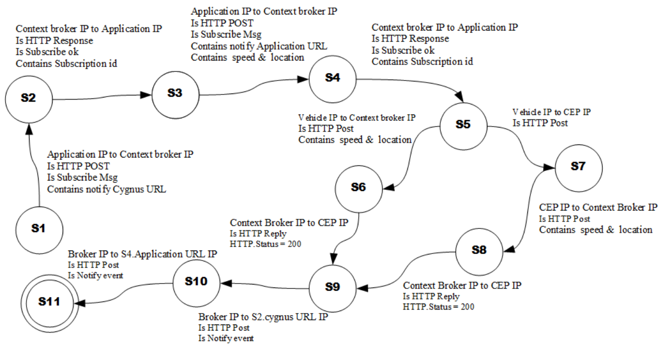
Context broker IP to Application IP
Is HTTP Response
Is Subscribe ok
Contains Subscription id
Application IP to Context broker IP
Is HTTP POST
Is Subscribe Msg
Contains notify Application URL
Contains speed & location
Context broker IP to Application IP
Is HTTP Response
Is Subscribe ok
Contains Subscription id
S2
S3
S4
Vehicle IP to Context broker IP
Is HTTP Post
Contains speed & location
S5
Vehicle IP to CEP IP
Is HTTP Post
S7
Application IP to Context broker IP
Is HTTP POST
Is Subscribe Msg
Contains notify Cygnus URL
S6
CEP IP to Context Broker IP
Is HTTP Post
Contains speed & location
S1
Context Broker IP to CEP IP
Is HTTP Reply
HTTP.Status = 200
S8
Broker IP to S4.Application URL IP
Is HTTP Post
Is Notify event
S10
S9
S11
Context Broker IP to CEP IP
Is HTTP Reply
HTTP.Status = 200
Broker IP to S2.cygnus URL IP
Is HTTP Post
Is Notify event

Figure 3: Pattern for Transport Application

this application domain presented interoperability challenges. This is highlighted by Table 1 which lists a subset of the services and open interfaces that must interoperate. For example: vehicles must interoperate with the NGSI publish-subscribe interface to post events to the context broker; the composition of complex event processing prior to event publications (e.g. events from vehicles being processed to produce speed and location data that can be published to the broker); the use of a Flume connector to persist context events such that they can then be post-processed using big data services; here WebHDFS is the protocol[10]. Overall, there are a number of complex specifications with different behaviour (streaming, publish-subscribe, and request response) that had to be understood and developed against.

To reduce the effort required to understand the challenges of making the software interoperate, we developed an interoperability pattern for this domain; a subset of this pattern is presented in Figure 3. Note, for space reasons we do not include full rules (as in Figure 2) but instead provide an overview of the rules. Within the pattern, example transitions are: i) S1 to S2 where the transport application registers a subscription to the context broker to persist events via the Cygnus flume connector; ii) S3 to S4 where the application registers a subscription to receive events about speed and location context for vehicles. iii) S5 to S6 where a HTTP Post message must be exchanged from the Vehicle to the context broker and the data must have at least speed and location attributes; iv) S5 to S8 where a vehicle publishes an event to the CEP service to process the event before it is published to the broker.

Initial analysis. The software components of the application were developed and tested in-line with the pattern (injecting typical interoperability errors into the software). In each case the tool identified the failure and which state and transition in the application the fault occurred. Hence, with this initial evaluation we believe that the tool has significant value to quickly identify interoperability errors in large-scale complex environments and hence reduce development costs. We plan to evaluate the tool further through community evaluation; we plan to use the FI-PPP community of developers, surveying their use of the tools to evaluate their effectiveness more rigorously.

Additionally, the pattern itself contains a number of subelements that are highly reusable i.e. common composition patterns for utilising the FI-WARE services (e.g. con-

[8]http://www.fi-ware.org/2013/09/19/santander-smart-city-event/
[9]http://www.fi-ware.org/tag/campus-party-brazil/

[10]http://archive.cloudera.com/cdh/3/hadoop/webhdfs.html

Service	Interface	Protocol/Data type
FI-WARE Context broker	Open Mobile Alliance's NGSI-9 and NGSI-10	HTTP Rest/JSON
FI-WARE Complex Event Processor	FI-WARE CEP specification	HTTP Rest/XML
Apache Flume connector (http://flume.apache.org)	In: NSGI; Out: WebHDFS	HTTP Rest/JSON
FI-WARE Big Data Service	WebHDFS	HTTP REST/JSON

Table 1: Heterogenous interface specifications

text broker, big data and CEP). Hence, we also quickly created simple environmental monitoring application types (with different data and behaviour) atop these sub-patterns. We saw that the patterns could be quickly composed and edited (with minimal effort), demonstrating the benefits of modelling both IoT services and applications to transfer knowledge between developers.

4. RELATED WORK

Middleware is typically put forward as an ideal solution to the interoperability problem. Where software is developed on a common middleware, with communication protocols that handle many of the heterogeneity issues e.g. Operating System, Hardware platform and data types differences, certain interoperability guarantees can be made. CORBA, Web Services, REST, and others highlight such ability. However, differences in the way developers use middleware (e.g. data semantics, application behaviour usage such a operation sequences) still result in interoperability issues to address; this is particularly true of the IoT domain with lightweight middleware (to operate on resource constrained devices), transporting highly heterogeneous data; there are a number of IoT middleware solutions, e.g. UbiSOAP [8], Hydra [9], DDS middleware [10], and MQTT [11]. Hence, our interoperability framework provides added value above middleware solutions, allowing multiple technologies to be deployed and then supporting developers address further application and middleware interoperability problems.

Testing languages are an alternative solution to the problem; most notably TTCN [12] used for testing of communication protocols and web services, and RESTAssured[11] for REST services. However, these offer programming solutions rather than a higher-level abstraction; this makes it difficult to quickly perform interoperability testing across a composition of services (indeed the solutions usually target the case of a single piece of software complying with a standard).

The domain of model-driven engineering has also considered similar solutions albeit often targeting different problems. The Motorola case study [13] demonstrated the cost reduction from model-driven practices, largely focusing on code generation and automated testing; it also advocates the need for decoupled models; for example, treating interoperability as a distinct concern. [14] also presents an approach to model adaptive software development for code deployed on heterogeneous software (e.g. sensors), leveraging the use of models to reduce effort and cost. Models have also been leveraged for the development of IoT software [15]; here state machine models are used to support the coding of web service composition, as opposed to the testing of interoperability between independently developed software. All of these solutions offer a clear indication of the benefits of models in the domain of IoT.

[11]https://code.google.com/p/rest-assured/

Finally, model-driven approaches have been put forward to broker between heterogeneous middleware solutions, essentially automating their interoperability [16, 17]. The benefits of modelling interoperability software shows how such abstraction can hide many of the technical challenges from software developers; Starlink's [17] use of state transition automata directly inspired the framework methodology in this paper. However, these solutions focus on brokering between heterogeneous software as opposed to supporting the developers of new software requiring interoperability. Beyond this, Emergent Middleware solutions [18, 19, 20] have been proposed that dynamically broker interoperability between systems; these solutions rely on machine-readable software artefacts, e.g., interface descriptions and ontologies, being available for run-time analysis. Yet, the reality is that systems do not typically publish such information and interoperability remains a significant software development challenge put back in the hands of software developers.

5. CONCLUSIONS AND FUTURE WORK

The increasing scale and complexity of IoT (in terms of devices, users and software) will continue to add to the challenge of composing heterogeneous software that interoperates. Excluding PCs, tablets and smartphones IoT is forecasted to grow to 26 billion units installed in 2020 (representing an almost 30x increase from 0.9 billion in 2009); further, IoT product & service suppliers is estimated to exceed $300 billion, resulting in $1.9 trillion in global economic value-add through sales into diverse end markets. Within this landscape, interoperability will pose significant challenges. There will also be demand for software engineering and middleware solutions to lower the complexity and reduce costs.

In this paper we have presented the challenges that are faced by the developers of IoT applications and services in terms of achieving interoperable software solutions in the face of highly heterogeneous communication protocols and data exchanged between IoT elements. We have advocated and described a lightweight, protocol independent, model-driven development approach to ensure interoperability in heterogeneous IoT services. Our key contributions here are: i) interoperability specific models that are lightweight to create, and are re-usable and composable to support a broad range of applications; ii) an evaluation framework to monitor IoT applications (specifically RESTful interactions in this paper) and evaluate how this software interoperates in accordance with the patterns.

We utilised a case-study approach to perform a preliminary evaluation of the value added to software developers in terms of helping them address the challenges interoperability poses. The FIWARE and FI-PPP domain offers a number of potential users composing open software elements, and we have used this to show the potential benefits of the framework i.e. reducing costs through simplifying interoperability, and capturing and reusing expertise surrounding the interoperability concern. We plan to use the FI-PPP community (with its large developer base) further to perform richer evaluation of the interoperability framework.

We see future work in two key areas. Firstly, the extension of the framework to move beyond REST and web services and also include such technologies and MQQT and XMPP (to increase the applicability of the tools to wider IoT devices). Secondly, to investigate reasoning technologies to infer in greater detail why interoperability has failed. At present, the framework reports where a rule has failed, and hence a developer can correct accordingly. However, in larger-scale systems involving complex patterns, the failure may be much more subtle requiring domain expertise to pinpoint what has gone wrong to make a rule fail. We will also explore the role of semantic rules within the framework e.g. where we test messages for semantic interoperability.

6. ACKNOWLEDGMENTS

This work was carried out as part of the XIFI project (https://fi-xifi.eu). This project received funding from the European Union's Seventh Programme for research, technological development and demonstration under grant agreement No. 604590.

7. REFERENCES

[1] Hyper/Cat. Iot ecosystem demonstrator interoperability action plan. Technical Report Version 1.1, September 2013.

[2] Yaser A. Bishr, Hardy Pundt, and Christoph Rüther. Proceeding on the road of semantic interoperability - design of a semantic mapper based on a case study from transportation. In *Proceedings of the Second International Conference on Interoperating Geographic Information Systems*, pages 203–215, 1999.

[3] Steve Vinoski. It's just a mapping problem. *IEEE Internet Computing*, 7(3):88–90, 2003.

[4] Massimo Paolucci and Bertrand Souville. Data interoperability in the future of middleware. *J. Internet Services and Applications*, 3(1):127–131, 2012.

[5] Satish Chandra, Vibha Singhal Sinha, Saurabh Sinha, and Krishna Ratakonda. Software services: A research roadmap. In *Proceedings of the on Future of Software Engineering*, FOSE 2014, pages 40–54, New York, NY, USA, 2014. ACM.

[6] Antonia Bertolino. Software testing research: Achievements, challenges, dreams. In *FOSE*, pages 85–103, 2007.

[7] Antonia Bertolino and Andrea Polini. The audition framework for testing web services interoperability. In *EUROMICRO-SEAA*, pages 134–142. IEEE Computer Society, 2005.

[8] Mauro Caporuscio, Pierre-Guillaume Raverdy, and Valérie Issarny. ubisoap: A service-oriented middleware for ubiquitous networking. *IEEE T. Services Computing*, 5(1):86–98, 2012.

[9] René Reiners, Andreas Zimmermann, Marc Jentsch, and Yan Zhang. Automizing home environments and supervising patients at home with the hydra middleware: Application scenarios using the hydra middleware for embedded systems. In *1st Workshop on Context-aware Software Technology and Applications*, pages 9–12, 2009.

[10] Gerardo Pardo-Castellote. Omg data-distribution service: Architectural overview. In *Proceedings of the 2003 IEEE Conference on Military Communications - Volume I*, MILCOM'03, pages 242–247, Washington, DC, USA, 2003. IEEE Computer Society.

[11] U. Hunkeler, Hong Linh Truong, and A Stanford-Clark. Mqtt-s; a publish/subscribe protocol for wireless sensor networks. In *COMSWARE 2008*, pages 791–798, Jan 2008.

[12] Ina Schieferdecker. Test automation with ttcn-3 - state of the art and a future perspective. In *Proceedings of the 22Nd IFIP WG 6.1 International Conference on Testing Software and Systems*, ICTSS'10, pages 1–14, Berlin, Heidelberg, 2010. Springer-Verlag.

[13] Paul Baker, Shiou Loh, and Frank Weil. Model-driven engineering in a large industrial context; motorola case study. In *Proceedings of the 8th International Conference on Model Driven Engineering Languages and Systems*, MoDELS'05, pages 476–491, 2005.

[14] Franck Fleurey, Brice Morin, and Arnor Solberg. A model-driven approach to develop adaptive firmwares. In *6th Intl. Symposium on Software Engineering for Adaptive and Self-Managing Systems*, pages 168–177, New York, NY, USA, 2011. ACM.

[15] Nils Glombitza, Dennis Pfisterer, and Stefan Fischer. Using state machines for a model driven development of web service-based sensor network applications. In *ICSE Workshop on Software Engineering for Sensor Network Applications*, pages 2–7, New York, NY, USA, 2010. ACM.

[16] Yérom-David Bromberg, Laurent Réveillère, Julia L. Lawall, and Gilles Muller. Automatic generation of network protocol gateways. In *ACM/IFIP/USENIX 10th International Middleware Conference, Urbana, IL, USA,*, pages 21–41, 2009.

[17] Yérom-David Bromberg, Paul Grace, Laurent Réveillère, and Gordon S. Blair. Bridging the interoperability gap: Overcoming combined application and middleware heterogeneity. In Kon and Kermarrec [21], pages 390–409.

[18] Gordon S. Blair, Amel Bennaceur, Nikolaos Georgantas, Paul Grace, Valérie Issarny, Vatsala Nundloll, and Massimo Paolucci. The role of ontologies in emergent middleware: Supporting interoperability in complex distributed systems. In Kon and Kermarrec [21], pages 410–430.

[19] Valérie Issarny and Amel Bennaceur. Composing distributed systems: Overcoming the interoperability challenge. In *11th International Symposium, FMCO 2012*, pages 168–196, 2012.

[20] Paola Inverardi, Romina Spalazzese, and Massimo Tivoli. Application-layer connector synthesis. In *11th International School on Formal Methods for the Design of Computer, Communication and Software Systems*, pages 148–190, 2011.

[21] Fabio Kon and Anne-Marie Kermarrec, editors. *Middleware 2011 - ACM/IFIP/USENIX 12th International Middleware Conference, Lisbon, Portugal, December 12-16, 2011. Proceedings*, volume 7049 of *Lecture Notes in Computer Science*. Springer, 2011.