View metadata, citation and similar papers at core.ac.uk
brought to you by CORE
provided by Aston Publications Explorer

Ann. Telecommun. (0000) 71:141–150
DOI 10.1007/s12243-015-0487-2

Model-driven interoperability: engineering heterogeneous IoT systems

Paul Grace[1] **· Brian Pickering**[1] **· Mike Surridge**[1]

Received: 31 March 2015 / Accepted: 30 October 2015 / Published online: 25 November 2015
© The Author(s) 2015. This article is published with open access at Springerlink.com

Abstract Interoperability remains a significant burden to the developers of Internet of Things systems. This is because resources and APIs are dynamically composed; they are highly heterogeneous in terms of their underlying communication technologies, protocols and data formats, and interoperability tools remain limited to enforcing standards-based approaches. In this paper, we propose model-based engineering methods to reduce the development effort towards ensuring that complex software systems interoperate with one another. Lightweight interoperability models can be specified in order to monitor and test the execution of running software so that interoperability problems can be quickly identified, and solutions put in place. A graphical model editor and testing tool are also presented to highlight how a visual model improves upon textual specifications. We show using case-studies from the FIWARE Future Internet Service domain that the software framework can support non-expert developers to address interoperability challenges.

Keywords Model driven engineering · Interoperability · Cloud computing · Internet of things

✉ Paul Grace
pjg@it-innovation.soton.ac.uk

[1] IT Innovation Centre, University of Southampton, Southampton, UK

1 Introduction

The Internet of Things (IoT) and Cloud Computing naturally go hand-in-hand for developing large-scale, data-oriented distributed systems. There is a growing need to collect and analyse data (in significant quantities) from an IoT world. However, this requirement brings a common problem to the fore, namely *Interoperability*. *Things* are highly heterogeneous networked devices employing different protocols (e.g. HTTP, MQTT, DDS and CoA) and different data formats (e.g. binary, XML, JSON and GIOP). These can then be composed with cloud-based infrastructure services (e.g. OpenStack and Amazon Web Services) or Cloud Platform services (e.g. Hadoop offerings) for further processing, analysis and storage. Where there are significant differences in technologies, how can systems be guaranteed to understand each other and interact? This is particularly true where there is the need to migrate IoT applications between Cloud providers.

This growing heterogeneity means that established methods to achieve interoperability are no longer appropriate. Complying with common standards and leveraging common middleware platforms reduces interoperability problems, but aiming for global consensus is unrealistic. Instead, interoperability is managed in an ad hoc manor: (i) specification compliance tests (of protocols and APIs), e.g. plug tests for MQTT (MQ Telemetry Transport) implementations,[1] (ii) published API documentation for developers to

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

follow (e.g. the Hyper/Cat [14] 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 [4] and mappings between middleware [20]). These solutions help, but there remains a significant burden on developers to understand and identify interoperability problems and then implement and test solutions accordingly. In this paper, we seek to reduce this burden using model-driven development tools and techniques.

Model-driven software development offers a principled approach to engineer interoperable solutions through: the capture of shared domain knowledge between independent developers and the automated generation and testing of software. For example, model-driven testing [2] and model-based interoperability testing [3] highlight the potential. However, these solutions focus on Web Services and require detailed models of the system's interface syntax (using WSDL) and behavior (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.

We present a *model-driven engineering tool* to simplify the engineering of interoperable systems. This paper highlights three key contributions of this work:

- *Interoperability models* are reusable, visual software artifacts that model the behavior of services in a lightweight and technology independent manner. These models are used to help developers create and test systems that correctly interoperate. These models 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). These models are based upon Finite State Machines (FSM); there are a number of active testing solutions based upon FSM [9]. Importantly, our 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 behavior.
- *A Graphical development tool* to allow the developer to create and edit interoperability models and to also execute tests to report interoperability issues. This tool aims to further reduce the development process by making it easier to understand and develop the models themselves; this is in contrast to textual, heavyweight and disjoint distributed systems models such as BPEL and WSDL.
- *The Interoperability monitoring and testing 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 behavior. The framework tests monitored systems against interoperability models to evaluate compliance, reporting where interoperability issues occur, such that the developer can pinpoint and resolve concerns.

Hence, the tool allows the developer to create, use and re-use "models of interoperability" to reduce development complexity in line with the following requirements to ensure interoperability is correctly achieved:

- *Specification compliance*; to check that systems comply with particular specifications, e.g. an IoT sensor produces event data according to the NSGI specification,[2] or streamed data content complies with a data format specification uploaded to the HyperCAT catalogue.
- *Interoperability testing*; monitors the interaction between multiple systems to test whether they interoperate with one another, identifying the specific issues to be resolved where there is failure.

To evaluate the tool, we utilize a case-study based approach. FIWARE[3] provides a marketplace of independently developed Future Internet Services (approximately 30) that can be composed to build IoT and cloud applications; these are loosely coupled REST services without formal interface or behavioral specifications, and hence achieving interoperability remains a significant task for developers. In the first case, we show how lightweight interoperability models 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. In the second case, we illustrate how multiple implementations of a specification across a federation can be tested for compliance to support application migration.

In Section 2, we present the model-driven engineering methodology; then in Section 3, we introduce the developer tool for model-driven engineering. Subsequently, we evaluate the framework in Section 4. In Section 5, we analyse the work in comparison to the state of the art, and finally, in Section 6, we draw conclusions and highlight future areas of application for the solution.

[2] http://technical.openmobilealliance.org/Technical/technical-information/release-program/current-releases/ngsi-v1-0

[3] http://www.fiware.org/

2 Model-driven interoperability

2.1 Interoperability engineering methodology

Figure 1 provides an overview of the engineering methodology; here different stakeholders use the interoperability modeling and testing tools to achieve interoperability between independently developed services and applications. These developers utilise the tool described in Section 3 to perform the *model, compose, edit, reuse* and *test* functions seen in the figure.

- Interoperability testers *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. Therefore, reducing the overall effort required to deliver, test and maintain correctly functioning distributed applications. The framework will identify application behavior and data errors, e.g. data is not received by system A because system B has not correctly published information.
- Application developers (these may be the same as interoperability tester) *model* the interoperability requirements of service compositions; that is, they create *interoperability models* 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.
- Service or API developers *model* the compliance requirements of their new service API, that is, they create *compliance models* to specify how applications must interact with their services, such that tests can be generated to ensure that an implementation of this model is compliant.
- Specification compliance testers *test compliance* of their specification implementation against the model of a service API in order to guarantee future interoperability with other parties conforming with this standard.

2.2 Interoperability and compliance models

Distributed services are typically modeled 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 [3] and better support service composition. However, these approaches are often tied to a specific technology type, e.g. Web Services and CORBA being clear technology silos, and hence the approach is not well suited to loosely-coupled IoT and cloud services that employ a wide range of technologies and communication protocols. Furthermore, the models themselves are typically complex to write, use and maintain which in turn means they 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 the need for separately maintained IDLs.

Our approach explores models that focus solely on interoperability; that is, the specification of the exchanges between IoT services with rules defining the required behavior for interoperability to be guaranteed. There are two types of model: (i) the interoperability model used by application developers and interoperability testers and (ii) specification models and compliance testers.

An *interoperability model* is specified as a finite state machine; the general format is illustrated in Fig. 2. A *state* represents a state of a distributed application (not an

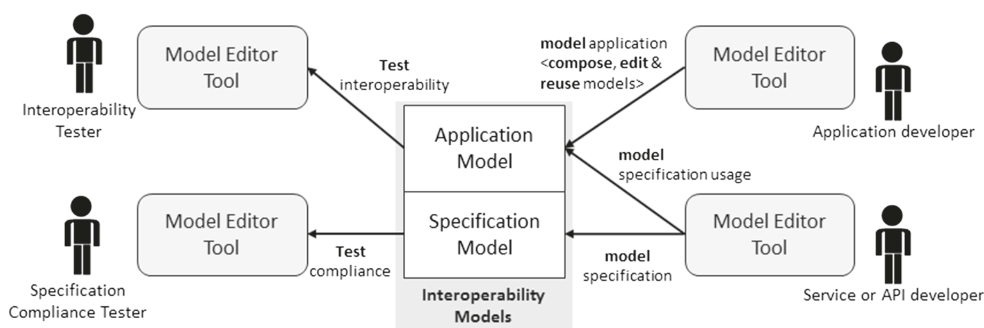

Fig. 1 Model-driven interoperability engineering

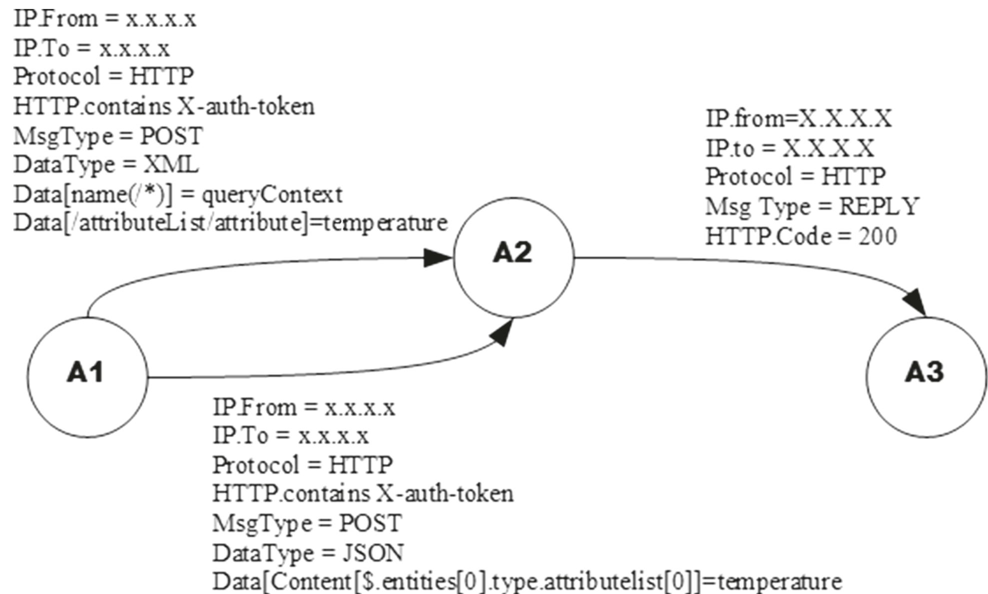

Fig. 2 Simple interoperability model

individual service) waiting to observe an event. A *transition* represents a change in state based upon an observed event matching a set of rules regarding the required behavior. 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 interoperate correctly.

If an event occurs and no transition can be made (because the event does not fulfill the rules), then the interoperability model identifies a failing condition. Aligned 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. Discrete events are captured messages (e.g. a HTTP message in Fig. 2), which are evaluated against the model, i.e., transition rules can be evaluated. Where all rules evaluate to true, the state machine transitions to the corresponding labeled state (e.g. from state A1 to A2 in the diagram).

In Fig. 2, we present a very simple example to illustrate how a model is used in practice. Here, we have a client requesting the temperature of a room sensor, and a context service providing the sensor data. They interact 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 alternative 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. Currently, the tool evaluates HTTP protocol rules.
- *Application and data-specific rules*. Evaluate the data content of protocol messages to 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, 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[4] and JSONPATH[5] tools to extract data fields

[4]http://www.w3.org/TR/xpath20/

[5]https://code.google.com/p/json-path/

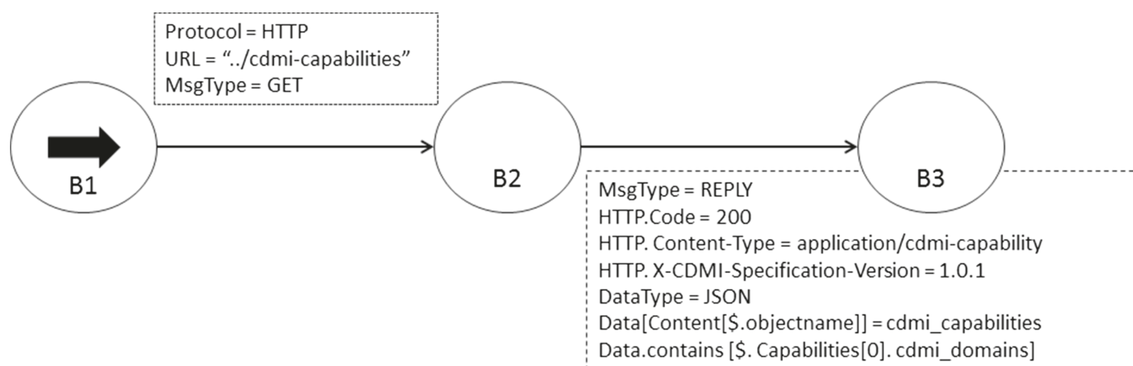

Fig. 3 Simple compliance model

and evaluate whether a given expression is true (e.g. a rule in the XPATH format: Data[name(/*)] = queryContext).

A *compliance model* is specified as a finite state machine using the same elements as the interoperability model above. However, two extra elements are now added in order to allow the framework executing the model to inject messages into the system in order to evaluate specification compliance:

– *Trigger state* (B1 in Fig. 3). This is an active state as opposed to an observing state, i.e., it does not monitor for events, rather it triggers the sending of a new event described in the out transition. A trigger state can only have one outgoing transition.
– *Trigger transition* (Transition from B1 to B2 in Fig. 3). This is a transition from one state of the distributed system caused by the sending of a new message. This message is a HTTP message that is described in the attributes of the transition.

A simple compliance model is illustrated in Fig. 3. Note, a trigger state has an arrow in the circle to distinguish it from a normal observing state. This is a simple model of part of the Cloud Data Management Interface[6] (CDMI api) specification for cloud storage. Here, we are testing if the service correctly implements the `discoverCapabilities` operations to view the technical capabilities and installed features of a CDMI deployment. The first state is a trigger state, this means that the tool creates and sends a HTTP GET message to the `cdmi_capabilities` URL. The system being tested for compliance should understand this message and send back a response. Hence, the second state transition evaluates a rule set against this received response to ensure that the data in the HTTP response matches the required data format of the api. Again, we see rules to test the structure of the HTTP message and that the data has fields *equal to* specific values and *contains* required fields.

3 Interoperability modeling and testing tool

The Modeling tool illustrated by the screen-shot in Fig. 4 has two core elements. First, the graphical editor providing drag and drop functionality to create the models described in the previous section. Second, the monitoring and testing framework that evaluates running distributed applications against the model visualised in the editor and evaluates them for correct interoperability. When the test command is selected in the editor—the models are converted to XML (this also allows them to be permanently stored) and they are input to the *Model Evaluation Engine*.

Without going into implementation details beyond the scope of the paper (more detailed information is found in [12]), we can explain the operation of the model evaluation engine in terms of two functions:

– Monitoring deployment; the framework takes an interoperability model as input and generates a set of proxy elements that capture REST 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 information before redirecting the request to the actual service. The implementation is built upon the RESTLET framework.[7]
– Model evaluator receives events and evaluates them against the rules specified in the transitions. The evaluator is protocol independent (per protocol plug-ins map concrete messages to the format of the model rules); hence, at present the framework parses HTTP messages, but is extensible to other data protocols. The evaluator creates a report to identify success or

[6] http://www.snia.org/cdmi

[7] http://restlet.com

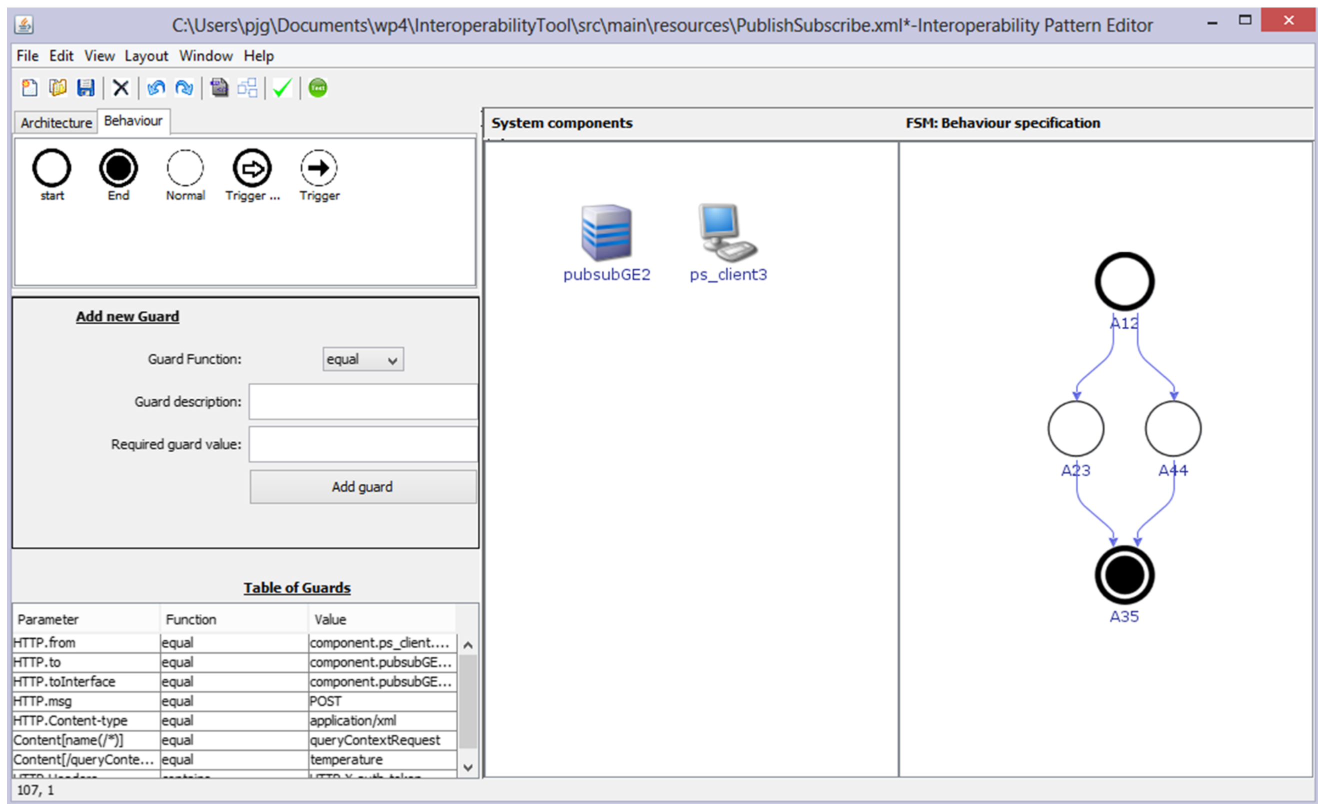

Fig. 4 Interoperability modeling and testing tool

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 software usable within the XIFI project[8] facilities. The source code is available at.[9] XIFI establishes a pan-European, open federation comprised of 17 data-center nodes to cope with large trial deployments and can serve the various needs of a broad set of FI users and experimenters. The interoperability framework is one of a number of tools to support the development of software using the FIWARE collection of open, restful services. Developers within this software domain can use the tool to view and edit models and then directly evaluate their application by executing the framework against this model.

4 Evaluation

We use a case-study approach to evaluate the usage of the interoperability model and associated tool to achieve its primary contribution, i.e. to reduce the effort required to develop and test the interoperability of software composed with independently developed IoT and cloud-based services. We hypothesize that the framework can monitor running services and identify where they do and do not interoperate; we also propose that the lightweight models offer an abstraction to capture interoperability information that can be reused across multiple applications, e.g. a model describing how to interoperate with a context broker being utilised across multiple different applications.

We utilise FIWARE software as the domain of our case study. FIWARE is a catalogue of approximately 30 RESTFul services implementing open specifications documented using free text (there are no WADL, WSDL specifications on which automated tool support can be based). 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.[10]

We hypothesize that the interoperability framework helps the developers of IoT applications and services during software development and testing phases; discovering problems

[8] http://wiki.fi-xifi.eu/Public:Interoperability_Tool

[9] https://github.com/pjgrace/connect-iot

[10] http://www.fiware.org/2013/09/19/santander-smart-city-event/

earlier, reducing the costs and improving the overall development of the application.

4.1 Case one: developing an application to interoperate with cloud and IoT services

We developed an application to monitor and gather data about traffic and transportation vehicles across multiple countries to support logistic reporting and analysis. The need to integrate new devices (e.g. vehicles) and services (e.g. reporting applications) into this application domain presents interoperability challenges. This is highlighted in Table 1 which list a subset of the services and open interfaces to interoperate with. For example, vehicles interoperating with the NGSI publish-subscribe interface; the composition of complex event processing prior to event publications; and choreographing post-processing of data using big data services. Here, there are a number of complex specifications with different behavior (streaming, publish-subscribe and request response) and data to understand and develop towards.

A model of interoperability was created for this application domain; a subset of the model is highlighted Fig. 5. Example transitions are vehicle to broker: HTTP POST message with JSON content to register new context, where the data must have at least attributes speed and fuel levels. Transport management application to broker: HTTP POST message to subscribe with URL for notifications, and then corresponding publication from broker to this endpoint. Interaction between broker and big data services to persist events, and then subsequent transport reporting application to create big data jobs to analyse the data. The full model consists of six interacting components, modeled by 38 states and 45 transitions.

Analysis The software components of the application were developed and tested in-line with the model (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. As the system grows in size, the visualisation ability allows the system to be tested without having to understand 1000s of lines of code. Additionally, the model itself contains a number of sub-elements that are highly reusable, i.e. common composition model for utilising the FIWARE services (e.g. context broker, big data and CEP). Hence, we also quickly created simple environmental monitoring application types (with different data and behavior) atop these sub-models. We saw that the models could be quickly composed and edited (with minimal effort), demonstrating the benefits of modelling both IoT services and applications to transfer knowledge between developers.

4.2 Case two: migrating between service providers

The XIFI federation consists of 17 geographically distributed nodes (where each node provides interfaces to utilise cloud and IoT resources at that location). Hence, the federation contains multiple deployments of service functionality useful to the application in case 1, e.g. publish-subscribe brokers (NGSI apis) and data storage services (CDMI apis). We hypothesize that the interoperability tool can model application compliance to evaluate if an application can be migrated between service providers, i.e. we can test that different service interfaces provide the functionality required by an application, identifying where compliance fails such that the application can then be edited to interoperate.

A compliance model was created for the NGSI publish-subscribe interface and CDMI data storage interface (as used in the transport application). This was then tested against the API deployments across each of the 17 XIFI nodes, reporting success in each case (as expected where standard specifications are deployed). The application code was then executed using the different nodes to verify that compliance testing success was equivalent to interoperability once the application was migrated. We finally created a mismatching api by changing the implementation of a test NGSI service; the compliance tool identified all points of

Table 1 Heterogeneous interface specifications

Service	Interface	Protocol
Context broker	Open Mobile Alliance's NGSI9[a]	HTTP Rest/JSON
Complex Event Processor	FIWARE CEP specification	HTTP Rest/XML
Big Data Adaptor	Apache Flume connector[b]	Binary
Big Data Service	FIWARE Big Data specification	Rest/XML
Object Storage	CDMI API specification	

[a] FIWARE open specifications
[b] flume.apache.org

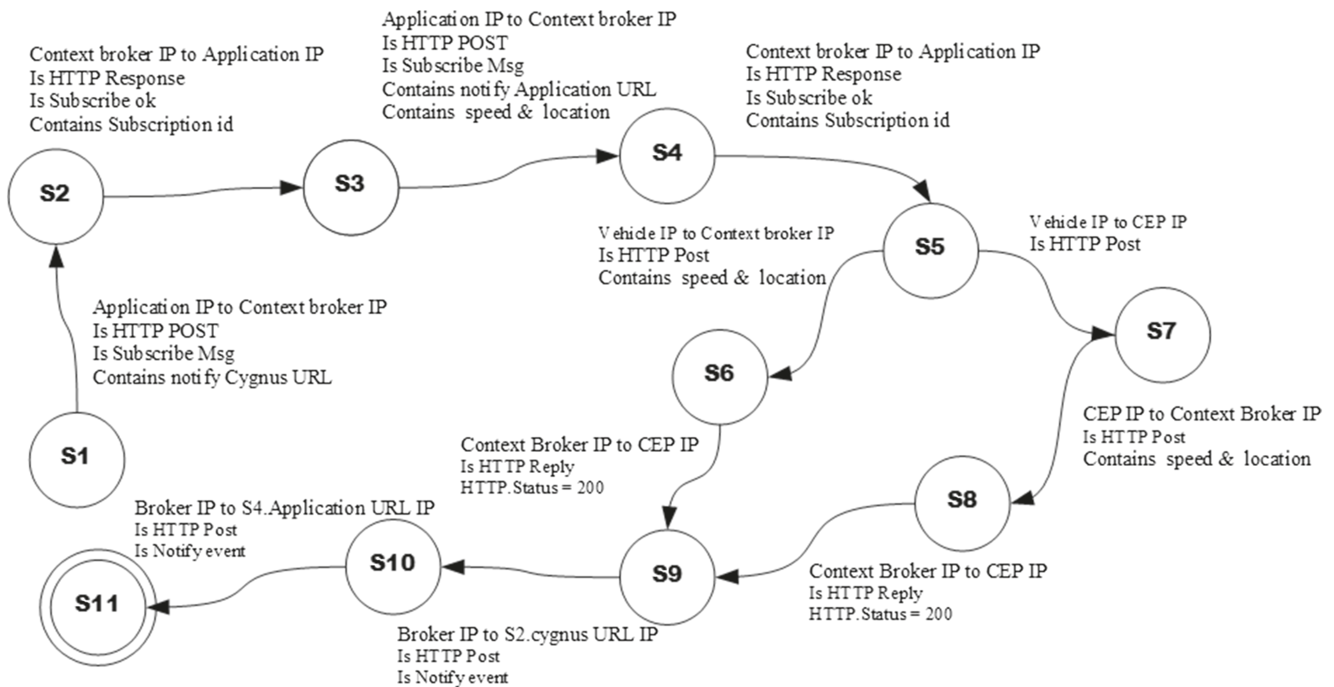

Fig. 5 Model of a transport application

mismatch correctly, and the application code was adapted to work with this new version of the API.

Analysis The use of lightweight compliance models can be used to establish how easy it is to migrate a given application to a new service. This can reduce the migration effort, reducing the need to analyse the new apis in detail, and then reimplementing code or writing brokers or adaptors.

5 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 complex 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 [18], DDS middleware [17] and MQTT [13]. 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 [19] 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.

The domain of model-driven engineering has also considered similar solutions albeit often targeting different problems. The Motorola case study [1] 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. Fleurey et al. [10] 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 [11]; 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; however, there is a clear indication of the benefits of models in the domain of IoT and cloud computing.

[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 [6, 7]. The benefits of modelling interoperability software shows how such abstraction can hide many of the technical challenges from software developers; Starlink's [6] 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 [5, 15, 16] have been proposed that dynamically broker interoperability between systems using semantics; 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.

6 Conclusions and future work

In this paper, we have presented the challenges that are faced by the developers of IoT applications and cloud 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. 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) a graphical tool to support visual development and testing and (iii) an evaluation framework to monitor application behaviour (specifically RESTful interactions in this paper) and evaluate how this software interoperates in accordance with the models.

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 when integrating and migrating systems. The FIWARE and XIFI domain offers a number of potential users composing open software elements, and we have shown the potential benefits of the tool, i.e. reducing costs through simplifying interoperability, and capturing and reusing expertise surrounding the interoperability concern.

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 MQTT 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 models, the failure may be much more subtle requiring domain expertise to pinpoint exactly what has gone wrong.

Acknowledgments This work was carried out as part of the XIFI project (https://fi-xifi.eu). This project received funding from the EU under grant agreement No. 604590. We also acknowledge Justan Barbosa's contribution to the case study applications.

Open Access This article is distributed under the terms of the Creative Commons Attribution 4.0 International License (http://creativecommons.org/licenses/by/4.0/), which permits unrestricted use, distribution, and reproduction in any medium, provided you give appropriate credit to the original author(s) and the source, provide a link to the Creative Commons license, and indicate if changes were made.

References

1. Baker P, Loh S, Weil F (2005) 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, pp 476–491
2. Bertolino A (2007) Software testing research: achievements, challenges, dreams. In: Briand LC, Wolf AL (eds) FOSE, pp 85–103
3. Bertolino A, Polini A (2005) The audition framework for testing web services interoperability. In: EUROMICRO-SEAA. IEEE Computer Society, pp 134–142
4. Bishr YA, Pundt H, Rüther C (1999) 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, pp 203–215
5. Blair GS, Bennaceur A, Georgantas N, Grace P, Issarny V, Nundloll V, Paolucci M (2011) The role of ontologies in emergent middleware: supporting interoperability in complex distributed systems. In: Kon F, Kermarrec A-M (eds) 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, pp 410–430
6. Bromberg Y, Grace P, Réveillère L, Blair GS (2011) Bridging the interoperability gap: overcoming combined application and middleware heterogeneity. In: Kon F, Kermarrec A-M (eds) Middleware 2011 - ACM/IFIP/USENIX 12th international middleware conference, Lisbon, Portugal, December 12-16, 2011. Pro- ceedings, volume 7049 of lecture notes in computer science. Springer, pp 390–409
7. Bromberg Y, Réveillère L, Lawall JL, Muller G (2009) Automatic generation of network protocol gateways. In: ACM/IFIP/USENIX 10th international middleware conference. Urbana, pp 21–41
8. Caporuscio M, Raverdy P, Issarny V (2012) ubisoap: a service-oriented middleware for ubiquitous networking. IEEE T Serv Comput 5(1):86–98
9. Cavalli A, Higashino T, Nez M (2015) A survey on formal active and passive testing with applications to the cloud. Ann Telecommun 70(3-4):85–93

10. Fleurey F, Morin B, Solberg A (2011) A model-driven approach to develop adaptive firmwares. In: 6th intl. symposium on software engineering for adaptive and self-managing systems. ACM, New York, pp 168–177
11. Glombitza N, Pfisterer D, Fischer S (2010) 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. ACM, New York, pp 2–7
12. Grace P, Barbosa J, Pickering B, Surridge M (2014) Taming the interoperability challenges of complex iot systems. In: Proceedings of the 1st ACM workshop on middleware for context-aware applications in the IoT, M4IOT '14. ACM, New York, pp 1–6
13. Hunkeler U, Truong HL, Stanford-Clark A (2008) Mqtt-s; a publish/subscribe protocol for wireless sensor networks. In: COMSWARE 2008, pp 791–798
14. Hyper/Cat (2013) Iot ecosystem demonstrator interoperability action plan. Technical Report Version 1.1
15. Inverardi P, Spalazzese R, Tivoli M (2011) Application-layer connector synthesis. in: 11th international school on formal methods for the design of computer, communication and software systems, pp 148–190
16. Issarny V, Bennaceur A (2012) Composing distributed systems: overcoming the interoperability challenge. In: 11th international symposium, FMCO 2012, pp 168–196
17. Pardo-Castellote G (2003) Omg data-distribution service: architectural overview. In: Proceedings of the 2003 IEEE conference on military communications - volume I, MILCOM'03. IEEE Computer Society, Washington, pp 242–247DC
18. Reiners R, Zimmermann A, Jentsch M, Zhang Y (2009) 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, pp 9–12
19. Schieferdecker I (2010) 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. Springer-Verlag, Berlin, pp 1–14
20. Vinoski S (2003) It's just a mapping problem. IEEE Int Comput 7(3):88–90