Enabling Model-Driven Software Development Tools for the Internet of Things

Karim Jahed
Queen's University
Kingston, Ontario, Canada
jahed@cs.queensu.ca

Juergen Dingel
Queen's University
Kingston, Ontario, Canada
dingel@cs.queensu.ca

ABSTRACT

The heterogeneity and complexity of Internet of Things (IoT) applications present new challenges to the software development process. Model-Driven Software Development (MDSD) is increasingly being recognized as a key paradigm in tackling many of these challenges, as evident by the emergence of a significant number of MDSD frameworks targeting IoT in the past couple of years. At the heart of IoT applications are embedded and realtime systems, a domain where model-driven development is well-established and many existing tools have a proven track record. Unfortunately, only a handful of these tools support out-of-the-box integration with the IoT. In this work, we discuss the different design and implementation decisions for enabling existing actor-oriented MDSD tools for the IoT. Moreover, we propose an integration approach based on the use of *proxy actors* and *system interfaces*. The approach offers seamless and flexible integration of external IoT devices into the user's model. We implement and evaluate our approach using the MDSD tool Papyrus for Realtime as a testbed.

KEYWORDS

Model-Driven Development, Internet of Things, UML-RT, Papyrus-RT

1 INTRODUCTION

The Internet of Things (IoT) is arguably one of the biggest leaps in the evolution of the Internet since the introduction of the World Wide Web. From a technological perspective, IoT is the logical result of the proliferation of Internet-ready, non-conventional computing devices, motivated in part by advances in wireless networking. A growing number of everyday physical objects – ranging from simple home appliances, wearables, and automotive, to complex industrial pipelines – are being connected to the Internet at an unprecedented rate. In fact, the number of IoT devices is expected to exceed 30 billion by the year 2020 [27], doubling a 2015 estimate of 15 billion devices [23].

The IoT promises far-reaching impact that touches nearly every aspect of our lives. Home automation, health monitoring, traffic management, and smart cities [36] are just a few examples of what IoT entends to enable. The enormous amount of data generated by hundreds of sensors embedded in our everyday lives, coupled with rapid breakthroughs in data analytics and machine learning, has given the promise of a trillion-dollar economy [1]. Now more than ever, businesses and governments are relying on data-driven insights to optimize industrial processes, guide management decisions, and draft public policies.

The IoT is envisioned as an infrastructure for a wide range of heterogeneous "things" that exchange all sort of data using a battery of network protocols. *Things* range from simple physical objects embedded with sensors and actuators to traditional computing devices and user interfaces, all the way to complex cloud services. An IoT application typically combines several of these entities in a well-defined manner to achieve certain goals, and provide the user with a unified, coherent experience.

The scale and complexity of envisioned IoT systems set forth new challenges for their design and development. Researchers have recently begun to recognize Model-Driven Engineering (MDE) as a key-enabler in tackling many of these challenges. The most prominent challenge being heterogeneity in both software and hardware [6]. Not only can IoT devices differ in terms of hardware and software, but the latter itself is often expected to be deployable on a wide variety of hardware. The abstraction capabilities that Domain Specific Languages (DSL) offer, coupled with automation by means of code generation, allows the specification of software systems in a way that transcends any single platform. A remarkable number of Model-Driven Development (MDD) tools for the Internet of Things has been proposed in the past couple of years [26] [28] [15] [13] [9] [18].

While MDD tools for the holistic design, development, and analysis of IoT systems continue to emerge, an array of tools and frameworks for the modeling real-time and embedded systems already exists and have a proven track record. Unfortunately, many of these tools lack integration with the IoT, as in they do not provide an explicit way to interact with third-party IoT devices. Significant effort has already been invested in recent years to remedy this problem, as we demonstrate in Section 5, however, as far as we can tell, the literature lacks a systematic presentation and discussion of the 'design space', i.e., the different fundamental design decisions and their advantages and disadvantages, and how to integrate code created with MDD tools into IoT applications.

In this work, we provide such a discussion of the design space while also describing a specific approach that we have chosen, implemented, and evaluated. We propose a combination of proxy model-components that acts as representative for external IoT devices and are easily embeddable in the user's model, along with specialized system interfaces that offer the modeller full control over the communication channel. This approach has the dual benefit of offering the complete abstraction of external devices when the particular details of the communication process are unimportant, as well as utmost flexibility and control over the network protocol when the application requires it. We describe the requirements for this integration at the different levels of the model-driven development stack and present the different design and implementation

options when possible. We corroborate some of our recommendations by conducting an experimental analysis using Papyrus-RT, an MDSD tool for embedded and realtime systems, as a testbed.

The rest of this paper is organized as follows: in Section 2 we introduce our intended application domain and introduce our testbed platform. We present our main approach in Section 3 and conduct our experimental analysis in Section 4. We highlight other tools and attempts in the literature to integrate into the IoT in Section 5. Section 6 concludes this work.

2 BACKGROUND

2.1 IoT Applications

While an IoT application can be composed of a multitude of components, they can generally be grouped into five categories: physical objects, gateway devices, people, services, and middlewares.

Physical objects. Physical objects, or Things, are everyday devices augmented with sensors and/or actuators and serve a specific purpose. These devices are often equipped with a short-range communication chip such as RFID, Bluetooth, or ZigBee, and their communication is restricted to immediate gateway devices. Common examples in a smart home application include smart light bulbs, electronic locks, and smart thermometers.

Gateway Devices. Due to their constrained nature, physical objects are not typically capable of communicating with the outside world. Instead, they rely on gateway devices, also known as IoT hubs. A gateway is a small computing device equipped with multiple radio technologies to manage Physical Objects, as well as LAN interfaces, such as Ethernet or WiFi, to connect to Internet services. Gateway devices vary in capabilities, but are typically programmable and provide an API that allows external actors to monitor and control connected Things. Examples include Samsung's SmartThings Hub, Wink Hub, and Raspberry Pies running the open source openHAB software, to name a few. In this work, gateway devices are our main concern. Our intention is to enable the model-driven development of software applications that can be deployed to these devices to control connected physical objects, and interface with the outside world.

People. People interact with IoT systems through user interfaces. User interfaces serve as a frontend to the system and provide refined views and interaction to accomplish a variety of tasks. Interfaces can be mobile, web-based, or even natural such as voice-controlled or gesture-based. A user interface commonly pulls data from many sources, including gateway devices and cloud services.

Services. Services are software systems that provide a set of functionalities via well-defined APIs. Services often run in the cloud and empower resource-constrained devices with functionalities that would otherwise be impossible to implement. Cloud services range from simple data providers, such as weather or traffic data providers, to advanced data analytics services.

Middlewares. An IoT middleware is a software system that facilitates the integration of different parts of an IoT system. By offering seamless interoperability between gateways, services, and user interfaces, a middleware helps in managing the heterogeneity of complex IoT applications. Popular examples include Node-RED, Flogo, and Eclipse Kura.

2.2 Distributed Real-time Systems

Our application domain encompasses distributed soft-realtime systems built upon message passing. In particular, we consider distributed applications that adopt the *Actor model of concurrency* [2]. The actor concurrency model proclaims computational entities, denoted *actors*, as first-class citizens; much like objects in Object-Oriented Programming. An actor is a unit of execution (possibly only conceptual) that, in response to a message it receives, can concurrently perform any sequence of three possible actions: update its own internal state, send messages to other actors, and create/destroy other actors.

Actors have well-defined *interfaces* that abstract their internal states and govern the way they interact with other actors. The interface of an actor exposes special communication points known as *ports*. Ports are linked using *channels* that mediate communication and dictate which actors are allowed to communicate via a given interface.

The actor paradigm continues to prove itself as one of the premier choices for the design of scalable and highly-distributed systems, as evident by its adoption in concurrent languages such as Erlang and Scala. Its inherently concurrent, modular, decoupled, and fault-tolerant structure, lessen many of the problems associated with distributed systems and alleviate the developers from having to deal with concurrency control and thread management. In short, the actor model makes it easier to write correct concurrent and parallel programs, by design. It is therefore to no surprise that a significant number of model-driven development tools and languages for real-time systems also adopts this model. Examples includes: MathWork's Simulink, The Generic Modeling Environment (GME) [22], Real-Time Object-Oriented Modeling (ROOM) [32], and Ptolemy [12], just to name a few.

2.3 The Model-Driven Development Stack

In general, we can identify three components in a typical model-driven development framework: the *modeling language*, the *modeling environment*, and the *runtime system*. At the top of the stack is the modeling language which defines the language elements, the abstract syntax, and the semantics. The modeling environment implements the language's concrete syntax and provides the users with facilities to edit, debug, and analyze models. The modeling environment can also offer a number of model transformations, most commonly a code generator to generate executable code from the model. At the bottom of the stack is the runtime system. The runtime system is the collection of services that the generated code can lean on to accomplish various tasks. In an actor-oriented framework, the runtime system could manage the actors' life cycle, implement the infrastructure for message exchange, and provide interfaces that give access to low-level system services such as logging, timing, and networking. It is important to note that the runtime system is not necessarily a discrete layer. While some tools might implement the runtime system as a collection of libraries linked against the generated code, or even as a virtual machine that executes the generated code, some others might embed the runtime services right into the generated code. In that case, the runtime system is simply the "extra" common code added to enable the execution of the model's code.

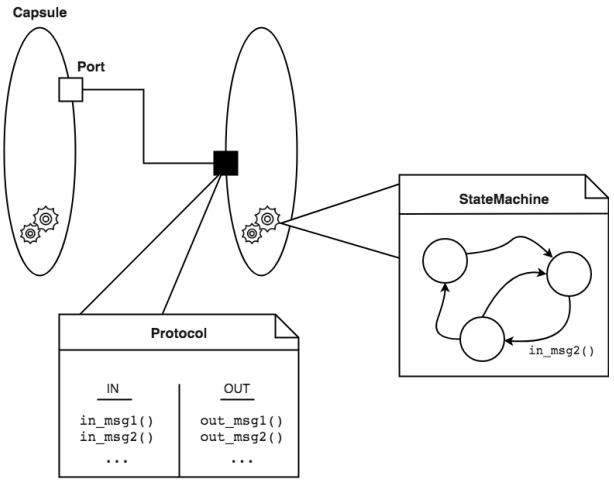

Figure 1: The main concepts of UML-RT. Capsules are active, executing classes whose behaviour is specified using a statechart. Capsules communicate solely via message passing using a well-defined protocol.

2.3.1 UML for Real Time. Although we do not assume any particular language or toolset in our approach, we use the UML for Real Time (UML-RT) profile to illustrate various concepts and provide concrete examples. UML-RT, an evolution from ROOM, is a small subset of UML specific to soft-realtime system modeling. UML-RT enjoys a range of successful tool support including IBM Rational RoseRT, IBM RSA-RTE, and more recently Papyrus-RT [1].

Figure 1 shows a rough summary of the main concepts in UML-RT (see [31] for a comprehensive discussion). UML-RT adopts a distributed message passing model based on the actor paradigm. *Capsules* are active, self-contained classes (i.e., actors) that encapsulate certain functionalities or represent a system component. Each Capsule maintains an internal state and can receive messages through its *Ports*. Each Port is typed with a specific *Protocol* that defines the set of messages that can be sent and/or received through that port (i.e. actor interface). Ports using the same Protocol can be linked via a connector (communication channel) which permits their corresponding Capsules to exchange messages as defined by the Protocol.

A Capsule's behaviour is specified using a statechart whose transitions are triggered by incoming messages. Snippets of action code can be hooked to transitions to perform arbitrary calculations, modify internal attributes, and send messages to other Capsules.

2.3.2 Papyrus-RT. Papyrus-RT is an open-source modeling environment developed by the PolarSys Eclipse Working Group. Based on the Papyrus platform [21], Papyrus-RT implements the UML-RT profile and allows complete, executable C++ code generation from UML-RT models. In this work, we use Papyrus-RT as a testbed to implement and analyze our proposed IoT integration approach.

We use an Intrusion Detection System (IDS), as a running example throughout this work. Figure 2(a) shows the structure of the system using a composition diagram as modeled in Papyrus-RT.

[1] Available at https://www.eclipse.org/papyrus-rt/

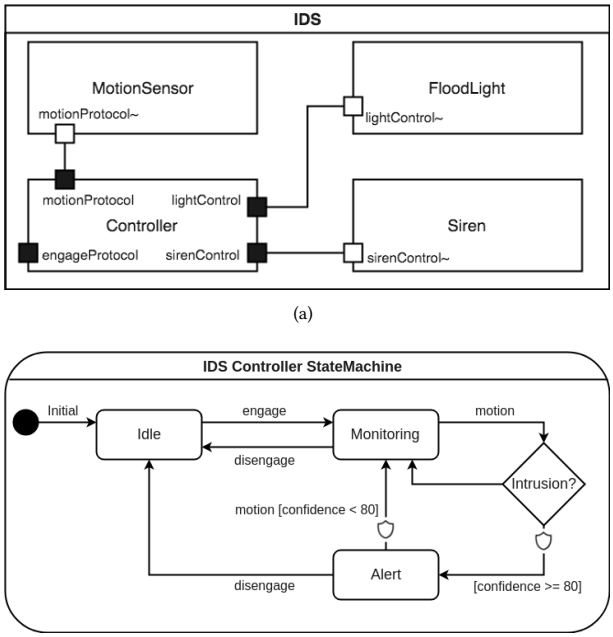

Figure 2: (a) Composition diagram for the Intrusion Detection System. (b) Statechart for the Controller capsule

The IDS system is composed of four capsules: a controller, a motion sensor, a flood light, and a siren. The controller communicates with various elements of the system via its ports. The controller receives periodic update messages from the motion sensor parametrized by a confidence level. When engaged, and depending on the confidence level, the controller decides whether to move to the Alarm state, activate the floodlight, and sound the siren. The controller then waits for a disengage message to move back to the IDLE state or move back to the Monitoring state whenever the confidence level of the motion sensor drops below a certain threshold (Figure 2(b)).

Note that the controller's engage/disengage signals are assumed to be sent by a component external to the IDS system, possibly an IoT device. This is currently not possible with Papyrus-RT and this was, in fact, part of the motivation for this work: how to best allow communication with external, arbitrary IoT components.

3 IOT INTEGRATION

In this section, we discuss the most important design decisions and lay out our proposed approach to enable collaboration with IoT devices in MDD environments. We discuss the support required at three different levels of the model-driven development stack, starting from the runtime system, through the modeling environment, up to the language level.

3.1 Runtime Support

In an actor-based distributed application stack, the runtime system (RTS) manages the actors' lifecycle, directs their execution, and provides them with an infrastructure for message passing. The RTS

also often implements an array of services that are extended to the application via *System Interfaces*. Examples of services offered include services for timing and scheduling, logging, and networking, among others. To support communication with external IoT devices, the runtime system must introduce support for IoT-related application protocols and data encoding formats, as well as new services to manage the communication channels with external devices.

3.1.1 ***Communication Protocols.*** There is literally a sea of IoT protocols, and choosing which ones to support can be a daunting task. Here, we are mainly concerned with data exchange protocols at the application layer rather than transport layer or link layer protocols, which we assume are implemented at the operating system level. In general, IoT protocols can be grouped into two categories depending on their messaging model: *publish-subscribe* or *request-reply*.

Publish-Subscribe. In publish-subscribe based protocols, a set of clients dubbed *subscribers* express interest in receiving certain kinds of messages by subscribing to *topics*. Messages are published to topics by *publishers*. A subscriber receives all messages published to a topic to which it subscribed. Some protocols that adopt publish-subscribe rely on *brokers* to relay messages between publishers and subscribers. Brokers are centralized servers that serve as a communication endpoint for all clients. The broker is primarily responsible for managing subscriptions, receiving messages, and delivering them to interested clients. One of the most prominent IoT protocols [16], the Message Queuing Telemetry Transport (MQTT), adopts this model. The MQTT Protocol is specifically designed for resource-constrained devices, adding very little overhead to messages and requiring only a few control packets. A major contributor to its lightweight-ness is its reliance on message brokers, which comes at the cost of scalability. Since all messages must travel through brokers, a broker acts as a single point of failure and can become a network bottleneck for high throughput applications. In essence, scaling an MQTT-based application requires scaling the brokers.

An alternative approach is through broker-less protocols, such as the Data Distribution Service (DDS) protocol. The DDS protocol is a decentralized publish-subscribe protocol that focuses primarily on high, near real-time performance. Its mainly intended for message-intensive machine-to-machine communication through direct messaging, although it still relies on centralized servers for initial peer discovery. Compared to MQTT, DDS can handle thousands of messages per second more while keeping latency and jitter to a minimum.

Request-Reply. Publish-subscribe is primarily a many-to-many paradigm where clients are more or less equals (though certain brokers might deny certain clients publishing rights). Request-reply, on the other hand, is a point-to-point paradigm. In the request-reply model, a client issues requests to (typically) a server and wait for replies, the most notable example being HTTP. HTTP is so ubiquitous that there is virtually no escape from supporting the protocol for IoT applications. A significant number of cloud services implement their API following the REST model over the HTTP protocol. The success of RESTful APIs over HTTP motivated the development of the Constrained Application Protocol (CoAP) [8]. Unlike HTTP, CoAP is designed specifically for constrained environments, allowing clients not only to issue traditional HTTP requests (via web proxies) but also expose their own RESTful services by acting essentially as a server. Like MQTT, CoAP is very wire-efficient with a very small header overhead. Unlike MQTT however, CoAP uses UDP as the underlying transport layer protocol. This allows CoAP to achieve a much higher message throughput at the expense of reliable message delivery.

	Intention	Transport	Model	Security	QoS	Availibility
MQTT	D2C	TCP	Pub/Sub	TLS	Reliable delivery	Broker is SPOF
DDS	D2D D2C	TCP UDP	Pub/Sub	TLS	Reliable delivery	Decentralized
CoAP	D2D	UDP	Req/Rep	DTLS	None	Decentralized

Table 1: Comparison of the MQTT, DDS, and CoAP IoT protocols. D2D: Device-to-Device, D2C: Device-to-Cloud, TCP: Transmission Control Protocol, UDP: User Datagram Protocol, TLS: Transport Layer Security, DTLS: Datagram Transport Layer Security.

Discussion. Table 1 shows a summary of the features of the three protocols discussed. Evidently, the choice of protocols is highly dependant on the application domain, and there are certainly many more than we can possibly discuss. In general, we can identify the need for at least one protocol suitable for device-to-cloud communication and another for device-to-device communication. MQTT serves well as a device-to-cloud protocol where message frequency is relatively low, while DDS and CoAP are better suited for high-frequency messaging where low-latency and jitter are required [34]. Between the two, DDS has the scalability advantage as a publish-subscribe protocol, while CoAP RESTful APIs shines for interoperability.

3.1.2 ***Message Serialization.*** In certain cases, the runtime system must serialize domain-level messages into payloads that can be delivered over the network protocol to its intended destination. Likewise, incoming payloads must be deserialized to messages that adhere to model-defined interfaces and that can be passed to local actors. Note that the necessity of these steps depends entirely on how communication with IoT devices is manifested at the model level (discussed in detail in Section 3.3. It may be, for example, that the raw IoT protocol is exposed to the modeller in which case preparing the payload is the responsibility of the modeller.

JSON. When serialization is necessary, however, the RTS must use a data format that is compatible with the other end of the communication channel. While there are infinitely many data formats, JSON remains overwhelmingly popular in the IoT domain and the Web in general, as evident by its adoption in major IoT platforms [24]. JSON is supported virtually by all programming languages. Its success is largely due to its simple, human-friendly notation and handful of data types.

To demonstrate how a language-level message specification can be serialized into a JSON message, consider the example UML-RT protocol for a Motion Detector shown in Figure 3. The protocol has one output signal `motion` that takes one integer parameter `confidence`. Whenever such message is directed to an external

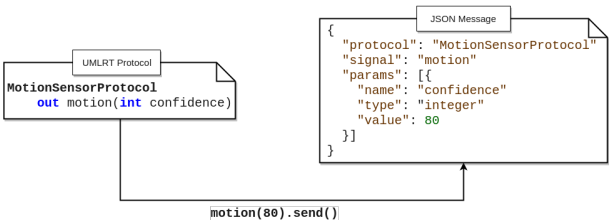

Figure 3: Serializing a UML-RT signal into a generic message encoded in JSON format

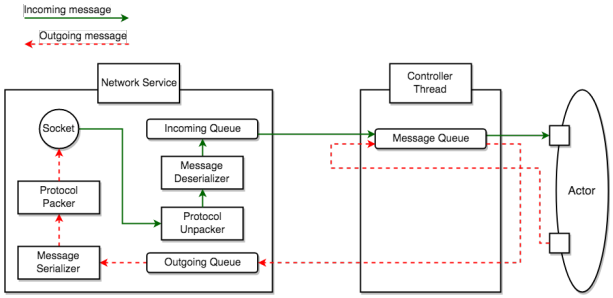

Figure 4: Handling of outgoing and incoming message by the network service

IoT device, the RTS can serialize the signal into the JSON string as shown in Figure 3, and transmit it over the wire.

Unfortunately, the user friendly and simple notation of JSON is a double-edged sword. The textual notation and extra syntactic decorators that makes the language human-readable can result in unnecessarily large payloads and significant processing time. This problem has not gone unnoticed, many binary compressed formats on top of JSON were introduced. Examples include BSON, Smile, and MessagePack, just to name a few.

Interface Definition Languages. A second problem entailed by the simplicity of JSON is its limited data type support. JSON supports four primitive data types, namely strings, integers, floats, and booleans, along with generic lists and objects. During deserialization, the RTS might be able to successfully deduce certain types, although not without loss of precision. However, the type of certain fields, such as character fields, might need to be stated explicitly in the message to avoid ambiguity.

For application domains that require absolute precision in data types, such as financial applications, a better approach is to adopt a data format that implements an Interface Definition Language (IDL). Example IDLs include Google's Protocol Buffer (Protobuf), Apache Thrift, and Apache Avro. IDLs allows the precise specification of messages in a language-and platform-neutral way through user-defined schemas. Out of these schemas, data serialization and deserialization classes for a variety of programming languages can be generated. The generated classes guarantee data interoperability among heterogeneous platforms.

Discussion. IDL-based serializers encode the data using a binary representation which often translates to superior performance and a smaller footprint than ASCII format such as JSON. However, in our context, the IDL schemas will need to be generated out of the models (for instance, out of protocols in UML-RT). This adds another layer of complexity to the code generator.

3.1.3 ***Network Services.*** For the chosen IoT protocols, the RTS needs to implement services to establish and maintain communication channels with IoT devices. These services typically run asynchronously from the main application thread, and continuously listen for incoming messages. Incoming messages are stored in a local FIFO queue, from which the message deserializer periodically polls in a producer-consumer fashion. After deserialization, a message is passed on to the main controller thread to be delivered to the appropriate actor. Similarly, the service also maintains a queue for outgoing messages. Messages pushed into the queue by the controller thread are serialized and transmitted over the wire. Figure 4 depicts this behaviour.

Failure recovery. An important factor in designing network services is failure recovery, or how to recover from network errors. Network errors vary from failure to deliver certain messages to complete connection drops. One option for handling errors is to follow a user-defined policy collected by the modeling tool as part of the configuration process. The policy could dictate, for example, whether to drop or re-transmit undelivered messages, or whether to automatically try and re-established dropped connections.

Another option for failure recovery is to expose an error-handling interface to the user's model. As mentioned earlier, the runtime system could uncover a number of system services to the user's model via specialized interfaces. In actor-oriented languages, ports can be typed with these interfaces to exchange message with system services. An error-handling interface could be used to notify the user's model of network-related events, such as undelivered messages, and the user model can choose to act accordingly by, for example re-sending the message or ignoring it all together.

3.2 Tooling support

The modeling environment can offer a number of facilities to ease the integration of external IoT components. The most essential of all is perhaps configuration options that supplement the model elements with the parameters required to establish and maintain communication channels with IoT devices. Take for instance a proxy actor (see Section 3.3.1) whose specific behaviour is implemented at the runtime level. In this case, the tool needs to collect parameters such as the IoT communication endpoint, network protocol, data format, authentication parameters, and other options that the runtime requires to successfully establish the communication channel.

In addition to basic configuration options, the modeling environment could also provide a number of tools to facilitate the integration of the model into IoT systems.

3.2.1 ***Interoperability.*** One of the biggest challenge in IoT that the modeling environment can help mitigate is interoperability. It is given that heterogenous IoT devices must be able to cooperate using different protocols, such CoAP or MQTT, and using different data formats such as JSON and XML. However, IoT devices must also be able to correctly interpret the meaning of exchanged

messages, i.e. their semantics. Consider for instance, the generic representation of a UML-RT signal in JSON format shown in Figure 3. It is highly unlikely that a large number of IoT devices, if any, are capable of interpreting the meaning of such message out of the box, even if they manage to successfully receive it and deserialize it. An IoT device at the receiving end would typically expect and act on device-specific messages.

Middlewares. One typical solution to the interoperability problem is by means of IoT middlewares [7]. A middleware can be a third-party, standalone service, that bridges IoT devices by providing support for a range of protocols and data formats, as well as user-defined functions to process messages as they flow from one device to another. One example middleware that fits this category is Node-RED [2]. Node-RED is a visual modeling environment that allows the user to construct data pipelines. Data flows from a source (MQTT client, for instance) and can be deserialized, reformatted, and translated, before it goes into a data sink (such as an HTTP client).

While third-party middleware can effectively remedy the interoperability problem, they can also contribute to extra communication overhead as all the messages must go through them, which also makes them a single point of failure for the entire system. Another class of interoperability middlewares are those that can be embedded directly into the IoT device software stack, offering a compatibility layer by implementing a common IoT standard. There has been a lot of effort in the IoT community to define standardized ways to accurately describe IoT components, as well as to standardize their interaction. Notable examples include the Open Mobile Alliance Light Weight Machine to Machine (LWM2M) specification [5], the oneM2M standard [33], and more recently, the Eclipse Vorto project.

IoT Object Models. Middlewares such as LWM2M and openM2M define a metamodel to accurately describe the resources and functionalities of IoT devices (commonly referred to as *Object Models*), along with a set of interfaces that specifies how this resources may be accessed and managed. Any device that implements these interfaces can effectively communicate with any other device by interpreting its corresponding object model, which is often kept in public repositories. By automatically generating these object models from the user's model, the modeling environment can ease the interoperability challenge between the user's model and external IoT devices. Of course, for other devices to be able to interface with the user's model, the runtime system must implement the compatibility layer as defined by the chosen specification.

Tool support for semantic interoperability. An alternative approach to handle semantic interoperability without relying on a middleware is to allow the modeller to define templates for message serialization and deserialization. This effectively integrates the message translation capabilities of a middleware right into the modeling environment. Instead of serializing model-level messages to generic messages (as shown in figure 3), the modeling tool could implement a templating language that allows the user to compose arbitrary messages. The templating language defines special placeholders for model-level variables, such as the signal name and the

[2] Available at https://nodered.org/

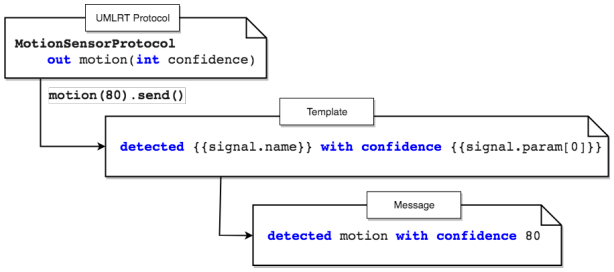

Figure 5: Serializing a UML-RT signal using a user-defined template

type and value of its formal parameters, that are resolved at runtime by a templating engine implemented as part of the runtime system. Figure 5 shows an example for serializing the UML-RT `motion` signal into a custom message using a user-defined template. Special placeholders such as `signal.name` and `signal.param[0]` are automatically resolved at runtime to the appropriate values depending on the input signal. This approach also works the other way around. Whenever a message is received, it is matched against the template to extract the value of the fields, such as the signal name, needed to construct the language-level message.

3.3 Modeling Language support

As a basic requirement, the modeling language must present the modeller with a notation to specify external IoT components, as well as their role within the model. While the specific notation is language-dependent, we can identify two generic approaches for message-passing based languages. The first is by means of *Proxy Actors* and allows the transparent inclusion of IoT components into existing models, while the second offers more flexibility to the modeller by exposing IoT protocols using *Internal Ports* and *System Interfaces*.

3.3.1 ***Proxy Actors.*** Generally, languages that provide a means to specify collaboration between components in a way that is independent of their internal behaviour can make use of existing notation, and thus, avoid convoluting the language. In actor-oriented languages such as UML-RT, an actor's interface serves as an abstraction layer that allows collaboration with other model elements irrespective of the actor's nature and realization. Therefore, an external IoT component can be represented as just another actor with an interface that defines the messages that may be exchanged with said component. The actor here serves as a mere proxy to some IoT device and does not necessarily implement any behaviour.

Like any other actor, proxy actors can receive and send messages to/from other actors in the system. However, proxy actors do not generally act upon received messages, instead, they simply relay them to the IoT component they substitute. In addition, a proxy actor listens for incoming messages from the external IoT component and delivers the message to the appropriate actor as specified by the collaboration model. Naturally, the proxy might need to reformat the messages between the IoT domain and the model domain. To illustrate, consider adding a mobile interface that allows the user to remotely engage and disengage our example Intrusion Detection

System. To implement this in UML-RT, we can represent the mobile user device as a regular capsule that is connected to the controller via its `engageProtocol` port, as shown in Figure 6(a).

The behaviour of a proxy actor can be implemented entirely at the model level or as part of the runtime library (see section 3.1). In the former case, there is virtually no need to modify any other part of the MDD toolchain. The actor encapsulates everything needed to communicate with the IoT component including the communication protocol and the message serialization/deserialization process. In the latter case, the behaviour of the proxy is deferred down the stack to the runtime system. In this case, it would suffice to label the actor as a "proxy" and annotate it with the appropriate parameters that allow the runtime system to establish the communication channel such as network protocol, data format, among others. As discussed in Section 3.2, this configuration can be provided by the IDE. However, other language features such as annotation, stereotypes, and deployment models could also be used.

Discussion. Obviously, there are benefits to both approaches. Implementing proxy actors as importable model-level components is a future-proof option and allow relatively easy support for emerging IoT technologies without the need to modify the runtime library. However, the behaviour of proxy actors is relatively complex and the modeling language might not be expressive enough to model the required behaviour. This is especially true when the action code is a restricted domain-specific language rather than a general-purpose language such as C++. Moreover, the introduction of many complex elements into the user's model might have a negative impact on the performance of the modeling environment and might introduce redundancy into the generated code, substantially increasing the size of the resulting binary. On the other hand, the runtime system can bundle resources for multiple proxy actors (for instance, using a single thread or socket connection for similar proxies), to optimize performance. Of course, the major drawback is the need to continuously update the runtime system and tooling to support emerging IoT protocols, data formats, etc.

3.3.2 ***Internal Ports.*** Internal ports are commonly used to expose low-level services provided by the runtime library to the model. Like border ports, internal ports are typed with an interface, typically referred to as a system interface, that defines the messages that may be exchanged with the underlying service.

Using a system interface, we can expose a generic IoT application protocol, implemented as a runtime service, to the modeller. While this approach does not provide the abstraction offered by proxy actors, it gives complete control and utmost flexibility to the user. Through the system interface, it is then up to the user to manage connections with the external IoT device, decode and process raw messages, and prepare and send outgoing data. Figure 6(b) shows how the external mobile user interface can be used to engage/disengage the IDS system via the MQTT IoT protocol implemented as a system interface.

The MQTT system interface declares the necessary functions to connect, subscribe, and publish messages, as well as inbound triggers to deliver received messages to the actor. In its initial transition, the IDS controller connects to the MQTT message broker and subscribes to a predefined topic "ids_control" using the `connect()` and

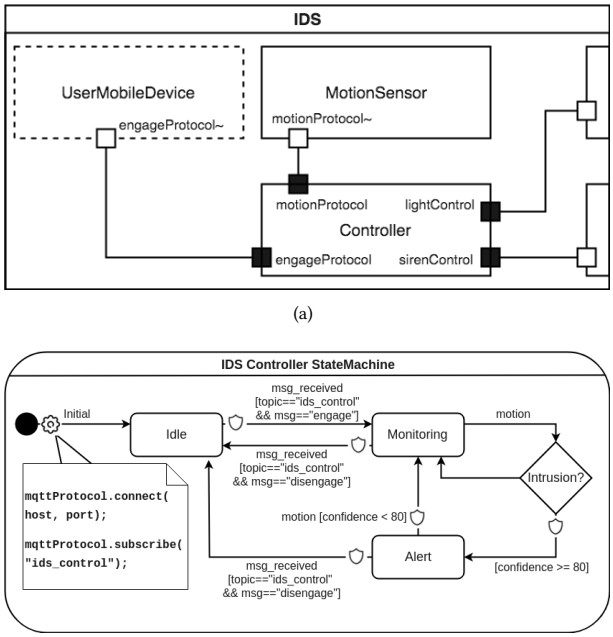

Figure 6: Connecting an external user mobile device to engage/disengage the controller using (a) a proxy actor and (b) using a system interface

`subscribe()` functions of the MQTT interface. Whenever a message with the topic "ids_control" is received by the MQTT service, the "msg_received()" signal is triggered with the received message as a parameter. The controller's statechart then parses the received message and decides whether to engage or disengage the system.

Discussion. It is worthy to mention that proxy actor and internal ports are not mutually exclusive. Rather a combination of both can prove to be a powerful addition to the language. Proxy actors provide a seamless integration of external components into the model with little effort from the modeller, at the expense of giving up control over the specific operation of the underlying protocol. In certain scenarios, however, such control is necessary. As an example, consider an application that needs to communicate with a cloud service using the MQTT protocol. However, in this example, the MQTT topic to which the service is listening to is not known before-hand, rather it is provided at runtime by some other device. By exposing the MQTT `subscribe()` and `publish()` functions to the modeller through a system interface, the model could dynamically subscribe to the topic once known and publish messages to the service. One thing to note here is that the flexibility of system interfaces is controlled by the tool developer. The developer can choose to expose all the functionalities of the underlying protocol or just a subset of them, depending on the intended use cases.

4 EXPERIMENTAL ANALYSIS

4.1 Implementation

Our implementation requires no modification to the UML-RT modeling language. A UML-RT capsule (with no behaviour) serves well as a proxy actor and integrates seamlessly with the rest of the model. The only requirement is to highlight a proxy capsule so that the runtime system can treat it appropriately. For that, we use a deployment map specified in a textual configuration file and supplied to the generated program as a command line argument. A deployment map typically maps components (capsules in our case) to processes or network hosts. The deployment map specifies, for each component, the address of the host where the component resides. In our implementation, we extend the deployment map to support IoT addresses. A capsule with an IoT protocol in its address, such as **`mqtt://`**`host:port` or **`coap://`**`host:port`, is treated by the runtime service as a proxy capsule. Support for another type of relevant configuration options, such as data format, quality-of-service (QoS) policies, and authentication parameters, were also added to the deployment map.

The deployment map is parsed by the Papyrus-RT runtime library, where most of our implementation resides. A network service at the runtime-level establishes a communication channel with each IoT device proxied by a capsule using the protocol specified in its address. UML-RT signals destined to proxy capsules are relayed to the network service by the capsule's execution controller. The network service serializes the signal and delivers it to the IoT device over the established communication channel. Similarly, messages received from the channel are deserialized by the network service and delivered to the destination capsule's controller.

Our implementation supports JSON and Google's Flatbuffers for message serialization. Flatbuffers is similar to Protobuf (discussed in Section 3.1.2), except that it supports in-buffer modification of the data prior to deserialization. Like Protobuf, Flatbuffers defines an Interface Definition Language (IDL) to describe structured data for serialization. From IDL models, Flatbuffers generates C++ helper classes that allow us to easily serialize and deserialize data in a platform-independent way. For JSON support, we use the RapidJSON library. While there are no official benchmarks, RapidJSON seems to be one of the fastest open-source JSON parsers available today. To encode UML-RT signals in JSON format, we use the same structure shown Figure 3.

In addition to the data serializers, our implementation supports the following IoT protocols:

TCP. While not an IoT application protocol, we added support for raw TCP for evaluation purposes. With the absence of application-layer-protocol overhead, TCP allows us to set benchmarks for our performance metrics.

MQTT. We use the Paho C++ client, part of the Eclipse IoT project, to support MQTT. An MQTT proxy capsule uses the broker's address as its deployment map address. The network service is optimized to bundle connections to the same broker, regardless of the number of capsules using the broker. Other protocol requirements, such as the topic name and quality-of-service (QoS) policy, are also specified in the deployment map.

CoAP. We added support for the CoAP protocol using the open-source libcoap library. Since CoAP is a request-reply protocol, the runtime network service always listens for incoming requests and responds appropriately. Messages are delivered and received using `POST` requests, for the most part.

DDS. We used OMG's OpenDDS library to support the DDS protocol. OpenDDS is a fairly mature library with many features, including support for both UDP and TCP as a transport-layer protocol. Unfortunately, however, DDS forces the use of its own specific IDL for message serialization. Since our implementation does not support DDS-IDL model generation from UML-RT signal, signals are always serialized into JSON or Flatbuffers messages first, then serialized again into a byte stream using DDS-IDL. This adds a small overhead to message serialization/deserialization whenever DDS is used.

4.2 Evaluation

4.2.1 Setup. To eliminate any chance for network variance and uncertainty, we conducted all of our experiments on the same host using UML-RT models to simulate external IoT devices. Moreover, whenever the protocol requires a message broker, a local broker on the same host is used. The host is equipped with a 2.2Ghz Intel Core i7 quad-core processor and 16GB of RAM. For performance evaluation, we use two simple Papyrus-RT models that represent a sender-receiver relationship. The two models, shown in Figure 7, are separate and the codes generated from these models run in separate processes. The `Sender` capsule in the `Sender Model` communicates with the `Receiver` capsule in the `Receiver Model` via a proxy (`ReceiverProxy`) capsule, and vice versa. From the perspective of the `Sender` (`Receiver`) capsule, `ReceiverProxy`) (`SenderProxy`) is an external IoT device, although it actually proxies a capsule in another UMLR-RT model.

On initialization, the `Sender` generates a random string message of a given length and sends it via its `sendRecvProtocol` port to the `ReceiverProxy` capsule then waits for an acknowledgement. Of course, the runtime system transmits this message to the `Receiver` capsule in the other model using the protocol defined in the deployment map. When the `Receiver` receives the message, it simply sends an acknowledgement signal to the `SenderProxy` capsule, which in turn is eventually delivered to the `Sender` capsule. Whenever the `Sender` receives an acknowledgement, it retransmits the same string again. This behaviour continues indefinitely.

4.2.2 Message serialization. We first evaluate the performance of data serialization using JSON versus Flatbuffers. Using the models of Figure 7, we vary the length of the random message transmitted by the `Sender` and measure the average message serialization and deserialization wall-time, as well as the message latency. The latency is computed at the `Sender` and is defined as the time spent between the moment a message is transmitted until an acknowledgement is received. In all of our experiments, we use raw TCP as a communication protocol and we let the models exchange messages for three minutes.

Table 2 shows the average serialization time, deserialization time, and round-trip latency for messages ranging from 1KB up to 256KB in size. As expected, Flatbuffers binary encoding outperformed JSON's ASCII representation for all cases. Message serialization and deserialization using Flatbuffers took respectively **94%** and **90%** less time than JSON, on average. Consequently, Flatbuffers message

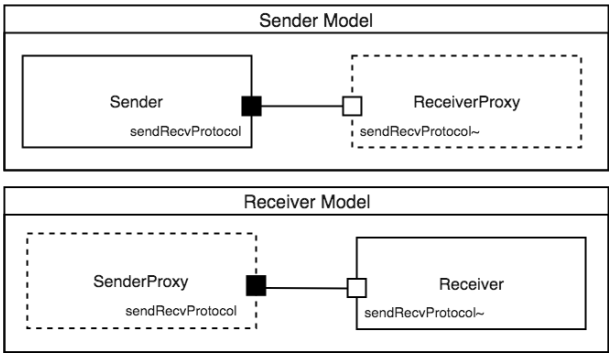

Figure 7: Models used for performance evaluation. The sender model communicates with the receiver model, and vice versa, via a proxy capsule.

Size (KB)	JSON			Flatbuffers		
	Avg. Ser. Time(μs)	Avg. Des. Time (μs)	Avg. Msg. Latency (ms)	Avg. Ser. Time (μs)	Avg. Des. Time (μs)	Avg. Msg. Latency (ms)
1	64.56	80.73	1.83	11.197	5.77	0.74
2	88.93	128.66	2.38	17.98	11.40	0.80
4	186.89	226.69	4.41	26.43	16.33	0.88
8	342.36	402.97	7.31	28.09	22.50	1.06
16	657.28	772.82	15.61	53.52	27.77	1.33
32	1337.87	1516.41	26.40	53.52	40.40	1.94
64	2518.11	2954.26	48.62	53.52	98.97	2.82
128	4593.13	5137.96	91.14	269.63	212.76	5.07
256	9280.38	10653.55	167.22	656.45	382.46	9.7

Table 2: Average message serialization and deserialization times in microseconds using JSON versus Flatbuffers

latency was on average **15%** that of JSON, with the percentage decreasing as the message size increases. It is worthy to mention that the evaluation scenario does not put JSON at a disadvantage. The transmitted messages generated by `Sender` capsule are ASCII strings which can be represented efficiently in JSON with little overhead from syntactic additions. In comparison, in other experiments we conducted where we transmitted binary payloads (as a list of integers in JSON), the size of the encoded JSON message quadrupled, tripling the latency (which includes network transmission time) when compared to ASCII payloads. In any case, JSON was never intended to carry binary data.

4.2.3 Protocols. We evaluate the IoT protocols we implemented by measuring the message throughput while varying the message size. The message throughput is defined as the number of acknowledged messages per second at the `Sender`. We use Flatbuffers for serialization in all the experiments, and we let the models run for three minutes. When applicable, we relax the QoS policy as much as possible, as long as message delivery is guaranteed.

Figure 8(a) shows the throughput for messages up to 64KB, while Figure 8(b) shows the throughput for larger messages up to 2048KB. Note that the CoAP and DDS/UDP protocols were unable to handle messages greater than 64KB due to their reliance on the unreliable UDP protocol. This, of course, can be mitigated in the future by implementing application-layer error-checking and retransmission in the network service.

Overall for small messages, CoAP performed very well and came close to the performance of pure TCP. On average, CoAP's throughput was **88%** that of TCP. On the other hand, all the other protocols showed similar performance for messages smaller than 64KB, reaching roughly **55%** the throughput of TCP. No significant difference in throughput was observed when using UDP versus TCP as a transport protocol for DDS. The outperformance of CoAP and TCP may be primarily due to their point-to-point architecture. In contrast, MQTT and DDS implement a publish-subscribe model, which will give them an edge in applications where many-to-many and one-to-many communication is prevalent.

For larger messages, the performance of MQTT began to quickly degrade, especially beyond the 512KB mark. MQTT's throughput averaged **39%** that of TCP, reaching as low as **8msg/s** for 2048KB messages. This degrade in performance is primarily due to MQTT's reliance on a third-party message broker. This requirement causes each message to be transmitted twice. In comparison, the peer-to-peer architecture of DDS/TCP managed to average **74%** of TCP's throughput.

5 RELATED WORK

Models are becoming more central not only to the design and development of IoT systems [26][15][13], but also for managing interoperability [17] [3], wiring IoT components [29], securing IoT applications [25][19], testing and simulation [4][11], deployment [14], and predicting their performance [20].

The separation of concerns, abstraction, and automation that MDD offers, and their role in taming the complexity of large heterogeneous systems, has motivated the development of MDD tools geared specifically towards IoT systems. The authors in [30] presented IoTLink, a framework for the rapid prototyping of IoT applications. Build on top of the Eclipse Modeling Framework, IoTLink allows the specification of IoT devices and their interactions through a visual modeling tool, out of which full Java code generation is possible. IoTLink implements a number of communication components that can be integrated into the model to consume data from external IoT devices, including an MQTT input and a REST input capable of parsing JSON and XML data. Calvin [28] is an actor-oriented framework for the complete development of IoT applications, from the initial specification all the way to deployment and provisioning. Actors are specified using CalvinScript and depend on the Calvin runtime for execution. One important aspect of Calvin, directly related to our work, is the ability to communicate with external, non-Calvin components by wrapping their REST API in a "dumb actor", essentially acting as a proxy for the service.

Beside emergent IoT-specific tools, there has been an effort to connect existing tools, that were primarily intended for the development of closed systems, to the Internet of Things. The authors in [35] recognized the importance of integrating legacy mechatronic systems into modern IoT manufacturing environment as a key requirement for Industry 4.0. For that, they introduced UML4IoT, a UML profile to generate IoT-compliant interfaces for mechatronic components. Given a UML model that expresses the object-oriented structure of the target component, the authors show how to generate an *IoTWrapper*, a RESTFul interface that exposes the component's properties to the outside world. Their IoTWrapper interface

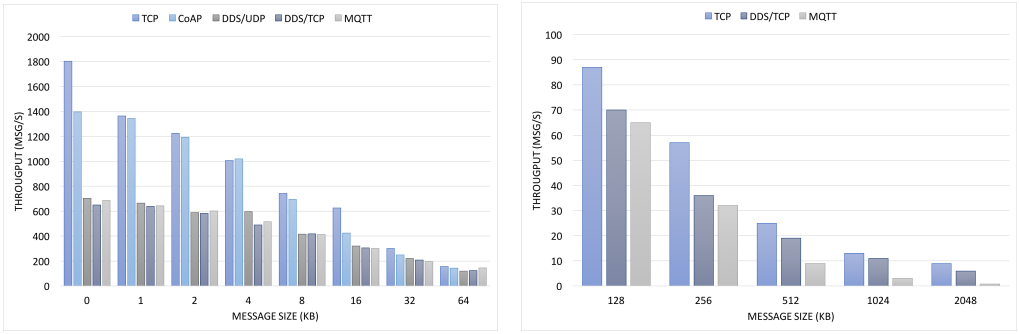

Figure 8: Message throughput for messages up to (a) 64KB and (b) 2048KB.

adopts the LWM2M standards that allow interoperability with other LWM2M-compliant IoT devices. The authors also show how this interface can be generated from annotated Java code of the component in case its UML model was not available.

The Ptolemy II modeling framework adopts *Accessors* [10], a concept similar to proxy actors introduced in Section 3.3.1. Also intended for actor-oriented systems, an Accessor is basically an actor that wraps a remote device or service, and can interface with the rest of the system. Unlike proxy actors, however, Accessors are not just a modeling concept whose specific behaviour is implemented by the tool's runtime system, according to the tool's specific needs. Instead, Accessors have well-defined execution semantics. An Accessor is defined in a Javascript file that specifies the Accessor's interface, such as inputs, outputs, and parameters, as well as its functionalities in reaction to incoming triggers. Internally, Accessors implement asynchronous atomic callbacks to invoke remote services and handle responses asynchronously. The execution of Accessors is managed by a *swarmlet host*, which includes a Javascript interpreter and a collection of libraries that offer the Accessor's script access to system services such as disk I/O and networking.

IBM Rhapsody, a UML-based system engineering modeling tool for embedded and realtime systems, offers MQTT support through an MQTT UML Profile extension. Rhapsody integrates nicely with IBM's IoT ecosystem. Part of that ecosystem is IBM's Internet of Things Workbench, a cloud service for specifying the architecture of IoT systems. The workbench allows the definition of abstract IoT devices as well as their interaction using Interaction Diagrams. For any IoT device defined in the workbench, the user can export a UML class that can be imported directly into a Rhapsody model. The class inherits the *MQTTProfile* stereotype and includes all the functions defined in the interaction diagram. When triggered, these functions are transmitted as MQTT messages to the corresponding device.

MathWorks Simulink is another model-based design tool for embedded systems that offers model simulation as well as C, C++, and HDL code generation. The Simulink modeling environment offers configurable MQTT blocks that can be linked to the rest of the model. Data delivered to the input ports of an MQTT sink block is published as MQTT messages to the broker/topic specified in the block's configuration. Similarly, source blocks can be used to receive MQTT messages.

6 CONCLUSION

In this work, we presented and discussed the different design and implementation options for enabling actor-based, model-driven software development tools for the Internet of Things. We proposed an approach based on the combination of proxy actors and system interfaces to enable seamless integration of external IoT devices in the user's model while offering control over the protocol operations when the application demands it. We implemented our approach in Papyrus-RT and evaluated the performance of several popular IoT protocols and data serializers. Our experiments showed that IDL-based message serialization not only offers cross-platform portability but also superior performance. In addition, we showed that the CoAP protocol, although relatively new, can easily outperform other protocols when the message size is reasonably small. For larger messages, the broker-based MQTT architecture throttles performance and is surpassed by DDS's peer-to-peer architecture. Our future work is geared towards supporting runtime adaptation, a growing requirement for IoT applications, at the model level.

REFERENCES

[1] 2018. Worldwide Semiannual Internet of Things Spending Guide. (2018). Retrieved April, 2018 from https://www.idc.com/getdoc.jsp?containerId=IDC_P29475

[2] Gul A Agha. 1985. *Actors: A model of concurrent computation in distributed systems.* Technical Report. MASSACHUSETTS INST OF TECH CAMBRIDGE ARTIFICIAL INTELLIGENCE LAB.

[3] Carlos Agostinho, Pedro Pinto, and Ricardo Jardim-Goncalves. 2014. Dynamic adaptors to support model-driven interoperability and enhance sensing enterprise networks. *IFAC Proceedings Volumes* 47, 3 (2014), 2400–2407.

[4] Abbas Ahmad, Fabrice Bouquet, Elizabeta Fourneret, Franck Le Gall, and Bruno Legeard. 2016. Model-Based Testing as a Service for IoT Platforms. In *International Symposium on Leveraging Applications of Formal Methods.* Springer, 727–742.

[5] Open Mobile Alliance. 2017. LWM2M Specification 1.0. *Open Mobile Alliance: San Diego, CA, USA* (2017).

[6] Kevin Ashton et al. 2009. That âĂŸinternet of thingsâĂŹ thing. *RFID journal* 22, 7 (2009), 97–114.

[7] Soma Bandyopadhyay, Munmun Sengupta, Souvik Maiti, and Subhajit Dutta. 2011. Role of middleware for Internet of Things: A study. *International Journal of Computer Science and Engineering Survey* 2, 3 (2011), 94–105.

[8] Carsten Bormann, Angelo P Castellani, and Zach Shelby. 2012. CoAP: An application protocol for billions of tiny internet nodes. *IEEE Internet Computing* 16, 2 (2012), 62–67.

[9] Marco Brambilla, Eric Umuhoza, and Roberto Acerbis. 2017. Model-driven development of user interfaces for IoT systems via domain-specific components and patterns. *Journal of Internet Services and Applications* 8, 1 (2017), 14.

[10] Christopher Brooks, Chadlia Jerad, Hokeun Kim, Edward A. Lee, Marten Lohstroh, Victor Nouvellet, Beth Osyk, and Matt Weber. 2018. A Component Architecture for the Internet of Things. *To Appear in Proceedings of the IEEE* (2018).

[11] Mihal Brumbulli and Emmanuel Gaudin. 2016. Towards model-driven simulation of the Internet of Things. In *Complex Systems Design & Management Asia*. Springer, 17–29.

[12] Joseph T Buck, Soonhoi Ha, Edward A Lee, and David G Messerschmitt. 1994. Ptolemy: A framework for simulating and prototyping heterogeneous systems. (1994).

[13] Federico Ciccozzi and Romina Spalazzese. 2016. MDE4IoT: supporting the internet of things with model-driven engineering. In *International Symposium on Intelligent and Distributed Computing*. Springer, 67–76.

[14] Iván Corredor, Ana M Bernardos, Josué Iglesias, and José R Casar. 2012. Model-driven methodology for rapid deployment of smart spaces based on resource-oriented architectures. *Sensors* 12, 7 (2012), 9286–9335.

[15] Bruno Costa, Paulo F Pires, Flávia C Delicato, Wei Li, and Albert Y Zomaya. 2016. Design and Analysis of IoT Applications: A Model-Driven Approach. In *Dependable, Autonomic and Secure Computing, 14th Intl Conf on Pervasive Intelligence and Computing, 2nd Intl Conf on Big Data Intelligence and Computing and Cyber Science and Technology Congress (DASC/PiCom/DataCom/CyberSciTech), 2016 IEEE 14th Intl C*. IEEE, 392–399.

[16] Konstantinos Fysarakis, Ioannis Askoxylakis, Othonas Soultatos, Ioannis Papaefstathiou, Charalampos Manifavas, and Vasilios Katos. 2016. Which IoT protocol? Comparing standardized approaches over a common M2M application. In *Global Communications Conference (GLOBECOM), 2016 IEEE*. IEEE, 1–7.

[17] Paul Grace, Brian Pickering, and Mike Surridge. 2016. Model-driven interoperability: engineering heterogeneous IoT systems. *Annals of Telecommunications* 71, 3-4 (2016), 141–150.

[18] Mahmoud Hussein, Shuai Li, and Ansgar Radermacher. 2017. Model-driven Development of Adaptive IoT Systems. In *4st International Workshop on Interplay of Model-Driven and Component-Based Software Engineering (ModComp) 2017 Workshop Pre-proceedings*. 20.

[19] Ricardo Jardim-Goncalves. 2017. A Model-Driven Adaptive Approach for IoT Security. In *Model-Driven Engineering and Software Development: 4th International Conference, MODELSWARD 2016, Rome, Italy, February 19-21, 2016, Revised Selected Papers*, Vol. 692. Springer, 194.

[20] Johannes Kroß, Sebastian Voss, and Helmut Krcmar. 2017. Towards a model-driven performance prediction approach for Internet of Things architectures. *Open Journal of Internet Of Things (OJIOT)* 3, 1 (2017), 136–141.

[21] Agnes Lanusse, Yann Tanguy, Huascar Espinoza, Chokri Mraidha, Sebastien Gerard, Patrick Tessier, Remi Schnekenburger, Hubert Dubois, and François Terrier. 2009. Papyrus UML: an open source toolset for MDA. In *Proc. of the Fifth European Conference on Model-Driven Architecture Foundations and Applications (ECMDA-FA 2009)*. 1–4.

[22] Akos Ledeczi, Miklos Maroti, Arpad Bakay, Gabor Karsai, Jason Garrett, Charles Thomason, Greg Nordstrom, Jonathan Sprinkle, and Peter Volgyesi. 2001. The generic modeling environment. In *Workshop on Intelligent Signal Processing, Budapest, Hungary*, Vol. 17. 1.

[23] Sam Lucero et al. 2016. IoT platforms: enabling the Internet of Things. *IHS Technology white paper* (2016).

[24] Julien Mineraud, Oleksiy Mazhelis, Xiang Su, and Sasu Tarkoma. 2016. A gap analysis of Internet-of-Things platforms. *Computer Communications* 89 (2016), 5–16.

[25] Ricardo Neisse, Gary Steri, Igor Nai Fovino, and Gianmarco Baldini. 2015. SecKit: a model-based security toolkit for the internet of things. *Computers & Security* 54 (2015), 60–76.

[26] Xuan Thang Nguyen, Huu Tam Tran, Harun Baraki, and Kurt Geihs. 2015. FRASAD: A framework for model-driven IoT Application Development. In *Internet of Things (WF-IoT), 2015 IEEE 2nd World Forum on*. IEEE, 387–392.

[27] Amy Nordrum. 2016. Popular internet of things forecast of 50 billion devices by 2020 is outdated. *IEEE Spectrum* 18 (2016).

[28] Per Persson and Ola Angelsmark. 2015. Calvin–merging cloud and iot. *Procedia Computer Science* 52 (2015), 210–217.

[29] Ferry Pramudianto, Markus Eisenhauer, Carlos Alberto Kamienski, Djamel Sadok, and Eduardo J Souto. 2016. Connecting the Internet of Things rapidly through a model driven approach. In *Internet of Things (WF-IoT), 2016 IEEE 3rd World Forum on*. IEEE, 135–140.

[30] Ferry Pramudianto, Carlos Alberto Kamienski, Eduardo Souto, Fabrizio Borelli, Lucas L Gomes, Djamel Sadok, and Matthias Jarke. 2014. IoT link: An internet of things prototyping toolkit. In *Ubiquitous Intelligence and Computing, 2014 IEEE 11th Intl Conf on and IEEE 11th Intl Conf on and Autonomic and Trusted Computing, and IEEE 14th Intl Conf on Scalable Computing and Communications and Its Associated Workshops (UTC-ATC-ScalCom)*. IEEE, 1–9.

[31] Bran Selic. 1998. Using UML for modeling complex real-time systems. In *Languages, compilers, and tools for embedded systems*. Springer, 250–260.

[32] Bran Selic, Garth Gullekson, and Paul Ward. 1994. Real-time object oriented modeling and design. (1994).

[33] Jorg Swetina, Guang Lu, Philip Jacobs, Francois Ennesser, and JaeSeung Song. 2014. Toward a standardized common M2M service layer platform: Introduction to oneM2M. *IEEE Wireless Communications* 21, 3 (2014), 20–26.

[34] Alejandro Talaminos-Barroso, Miguel A Estudillo-Valderrama, Laura M Roa, Javier Reina-Tosina, and Francisco Ortega-Ruiz. 2016. A Machine-to-Machine protocol benchmark for eHealth applications–Use case: Respiratory rehabilitation. *Computer methods and programs in biomedicine* 129 (2016), 1–11.

[35] Kleanthis Thramboulidis and Foivos Christoulakis. 2016. UML4IoT – A UML-based approach to exploit IoT in cyber-physical manufacturing systems. *Computers in Industry* 82 (2016), 259–272.

[36] Andrea Zanella, Nicola Bui, Angelo Castellani, Lorenzo Vangelista, and Michele Zorzi. 2014. Internet of things for smart cities. *IEEE Internet of Things journal* 1, 1 (2014), 22–32.