Software and Systems Modeling (2022) 21:1583–1611
https://doi.org/10.1007/s10270-021-00945-3

REGULAR PAPER

Model-driven development of asynchronous message-driven architectures with AsyncAPI

Abel Gómez[1] · Markel Iglesias-Urkia[2] · Lorea Belategi[2] · Xabier Mendialdua[2] · Jordi Cabot[3]

Received: 30 October 2020 / Revised: 15 October 2021 / Accepted: 20 October 2021 / Published online: 2 December 2021
© The Author(s) 2021

Abstract

In the Internet-of-Things (IoT) vision, everyday objects evolve into cyber-physical systems. The massive use and deployment of these systems has given place to the Industry 4.0 or Industrial IoT (IIoT). Due to its scalability requirements, IIoT architectures are typically distributed and asynchronous. In this scenario, one of the most widely used paradigms is *publish/subscribe*, where messages are sent and received based on a set of categories or *topics*. However, these architectures face interoperability challenges. Consistency in message categories and structure is the key to avoid potential losses of information. Ensuring this consistency requires complex data processing logic both on the publisher and the subscriber sides. In this paper, we present our proposal relying on *AsyncAPI* to automate the design and implementation of these asynchronous architectures using model-driven techniques for the generation of (part of) message-driven infrastructures. Our proposal offers two different ways of designing the architectures: either graphically, by modeling and annotating the messages that are sent among the different IoT devices, or textually, by implementing an editor compliant with the *AsyncAPI* specification. We have evaluated our proposal by conducting a set of experiments with 25 subjects with different expertise and background. The experiments show that one-third of the subjects were able to design and implement a working architecture in less than an hour without previous knowledge of our proposal, and an additional one-third estimated that they would only need less than two hours in total.

Keywords Publish/subscribe · Cyber-physical systems (CPS) · Message-driven architectures · Asynchronous communication · AsyncAPI · Industrial Internet of Things (IIoT)

Communicated by Federico Ciccozzi, Nicolas Ferry, Arnor Solberg, and Manuel Wimmer.

✉ Abel Gómez
agomezlla@uoc.edu

Markel Iglesias-Urkia
miglesias@ikerlan.es

Lorea Belategi
lbelategi@ikerlan.es

Xabier Mendialdua
xmendialdua@ikerlan.es

Jordi Cabot
jordi.cabot@icrea.cat

[1] Internet Interdisciplinary Institute (IN3), Universitat Oberta de Catalunya (UOC), Barcelona, Spain

[2] Ikerlan Technology Research Centre, Basque Research and Technology Alliance (BRTA), Arrasate-Mondragón, Spain

[3] ICREA – Internet Interdisciplinary Institute (IN3), Universitat Oberta de Catalunya (UOC), Barcelona, Spain

1 Introduction

The emergence of the Internet of Things (IoT) [18] has dramatically changed how physical objects are conceived in our society and industry. In the new era of the IoT, every object becomes a complex cyber-physical system (CPS) [27], where both the physical characteristics and the software that manages them are highly intertwined. Nowadays, many everyday objects are in fact CPSs, which increasingly use sensors and interfaces (APIs) to interact and exchange data with the cloud [45].

The ideas behind the IoT have been especially embraced by industry in the Industrial IoT (IIoT) or Industry 4.0 [28]. It is in this Industry 4.0 scenario where CPSs become especially relevant, mainly in control and monitoring tasks [31].

IIoT has largely contributed to the growing interest in message-driven architectures [33], best used for asynchronous communication. Indeed, in order to achieve higher degrees of scalability, CPSs are typically deployed on

message-driven asynchronous architectures that improve the overall behavior and reliability of systems. One of the most popular paradigms today is *publish/subscribe* [1]—followed by, for example, the *Message Queuing Telemetry Transport* (MQTT) protocol—where messages that are sent to and from a CPS are not directed to a certain recipient, but are proactively published and consumed by the agents involved according to certain criteria or categories. However, although these distributed architectures are especially scalable and tolerant to changes, they are not problem-free: since communication is done between equals, there must be an agreement between all parties on what are the expected message categories, as well as on their internal format and structure.

This is a key challenge we face at *Ikerlan*. As a technology center, *Ikerlan* currently coordinates different projects developing solutions to monitor, control and supervise systems of remote IoT devices in manufacturing plants, consumer goods, warehouses or smart buildings. Such solutions must support environments where a large number of devices send and consume data (e.g., sensor information, batch processed data, etc.). Following the current trend, the solutions developed are based on message-driven architectures following the *publish/subscribe* paradigm.

Based on our experience in working on a large number of IoT projects of different sizes and domains, we propose a model-driven solution to design and develop these architectures efficiently. Our proposal relies on the *AsyncAPI Specification* [5] to formalize and (semi)automate the design and implementation of these architectures. Previously [21], we presented both AsyncAPI-based concrete and abstract syntaxes and code generation templates to generate a Java-based internal DSL to support message-driven architectures. In this paper, we generalize and extend our proposal, and we provide further details. A key extension not presented before is the graphical concrete syntax based on data models together with a set of model transformation to kick start the modeling and development of message-driven architectures. The transformations also guarantee the interoperability between the different syntaxes to ensure that developers can, for instance, start with the graphical one and then refine the result with the textual representation. These extensions have been implemented and integrated in our toolkit.

Furthermore, this paper completes our previous work by reporting on the evaluation of our toolkit. The evaluation we present here has been carried out with a group of 25 subjects—both end-users and developers—with different levels of knowledge in the required concepts. Specifically, we asked them to implement a small example architecture of an automated warehouse. The goal of this evaluation was to answer the following research questions: (i)how do users think our toolkit helps them in the definition and development of an asynchronous message-driven architectures? (ii)is previous knowledge required to use our toolkit? (iii)which are the weaknesses and strengths of our toolkit? and, (iv)does the graphical syntax provide real benefits?

To answer these questions, the experiment was conducted in two phases: first, we evaluated the usefulness of the initial tool and got the feedback of the participants using a questionnaire, and second, after evaluating the initial results, we selected the 11 subjects with previous knowledge on modeling and asked them to use the graphical syntax to implement the same exercise to evaluate the extensions done. The complete prototype is available as an open-source toolkit freely available online[1].

The rest of this paper is structured as follows: Section 2 motivates this work by introducing a small use case that will be used as a running example throughout the rest of the paper, while Sect. 3 presents *AsyncAPI*, which serves as the background for this work. Section 4 presents our proposed workflow for designing and developing message-driven architectures as well as the overall picture of our proposed tool, and Sect. 5 presents in detail our tool, describing all the main artifacts and components involved. Section 6 presents how we validated our proposal, first discussing on our initial experiences and internal evaluation and second by presenting our experiments with external subjects. Section 7 presents the related work. We finish the paper with Sect. 8, where we present our conclusions and summarize our roadmap.

2 Motivation

Monitoring and control needs, as well as security and reliability requirements, make possible—and even desirable—to reuse a generic reference architecture in IIoT environments. Architectures in these environments are typically message-based, thus allowing a low coupling among the elements in the architecture. One common message-based paradigm is *publish/subscribe*, where messages are not directly sent to the recipients who will consume them, but are published under a certain category called *topic*. Only the devices subscribed to a certain *topic* will receive the messages published under it.

A common use case of these message-based architectures is IoT devices publishing monitoring and status data. Such data, whose volume can be very high, may be consumed by a cloud application that filters and processes it. On the other hand, control messages may be sent to reconfigure the IoT devices through a *Frontend* when needed. The central element of this architecture would be the *message broker*: the

[1] https://hdl.handle.net/20.500.12004/1/A/ASYNCAPI/001.

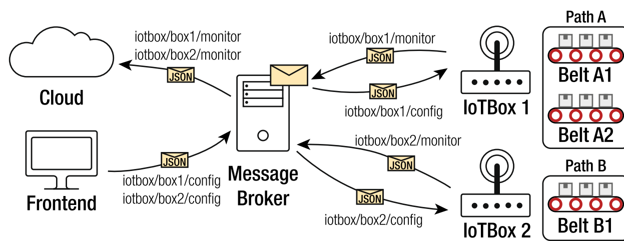

Fig. 1 Example of a message-based architecture in IoT

element in charge of managing publications, subscriptions, and the flow of messages between the elements of the network.

Example In order to illustrate the kind of architectures explained above, we will use a simplified use case of an automated warehouse from an industrial partner as shown in Fig. 1. The *message broker* is shown in the center of the figure, and as aforementioned, it is in charge of managing the different publications and subscriptions. The warehouse has different *paths*, each one containing a varying number of *conveyor belts*, as depicted on the right-hand side of the figure. IoTBoxes are IoT devices which are distributed throughout the factory and are capable of monitoring and controlling different paths. In the example, IoTBox 1 controls and monitors *Path A* (and thus belts *A1* and *A2*), while IoTBox 2 controls and monitors *Path B* (and as a consequence, belt *B1*). The IoTBoxes periodically collect data from the belts in the path – e.g., speed values. This datum is sent to be further processed in the *cloud*. To do this, IoTBox 1 publishes its monitoring information in topic `iotbox/box1/monitor`, while IoTBox 2 does the same on topic `iotbox/box2/monitor`. To receive the monitoring data, the *cloud* is subscribed to topic `iotbox/+/monitor`—the + symbol acts as a wildcard—and as a consequence, the *cloud* receives the monitoring data of both IoTBoxes under topics `iotbox/box1/monitor` and `iotbox/box2/monitor`.

IoTBoxes may also receive configuration commands—for example, to change the monitoring frequency. These configuration commands can be issued remotely via the *frontend*. For example, the frontend may publish the desired monitoring frequency for *Path A*—which is controlled by IoTBox 1—by publishing a message with the correct format under topic `iotbox/box1/config`. Since IoTBox 1 is subscribed to this topic, it will receive the configuration command and will reconfigure itself as requested. The same applies to IoTBox 2 and *Path B*, but under topic `iotbox/box2/config` in this case.

As it can be guessed by examining the example above, one of the major challenges that these architectures pose is **consistency** [22]: the format of the messages exchanged and the *topics* under which they are published and subscribed

Listing 1 Example message as to be published under `iotbox/box1/monitor`

```
1  {
2      "id":"Path_A",
3      "belts": [
4          {
5              "id": "Belt_A1",
6              "ts": "2020-09-09T05:44:54.482Z",
7              "speed": 10
8          },{
9              "id": "Belt_A2",
10             "ts": "2020-09-09T05:44:54.599Z",
11             "speed": 15
12         }
13     ]
14 }
```

must be kept consistent throughout the life cycle of the system. Failure to comply with this could result in a system malfunction: if any of the agents introduces (even minimal) changes in the definition of communications, and these changes are not propagated to all the agents involved, interoperability problems will inevitably occur. Communication can cease to be effective for two reasons: (i) because there is a **divergence in the *topics*** under which messages are published, thus resulting in agents not receiving messages they are interested in; or (ii) because there is a **divergence in the format** of the messages of a certain *topic*, and therefore, these cannot be understood by the subscribers receiving them.

Example Listing 1 shows a well-formed message as it is published by IoTBox 1 under topic `iotbox/box1/monitor`. The message contains a JSON object with two fields: `id` and `belts`. `id` identifies the path, and `belts` is an array of objects, each one containing three fields: a belt `id`, the timestamp (`ts`) when the speed was observed on the belt, and the `speed` of the belt in m/s. An example of divergence in the topics is the case when the cloud subscribes to the wrong topic by mistake (e.g., to `iotbox/box1/monitoring` instead of `iotbox/box1/monitor`). In such a case, the monitoring data of IoTBox 1 will not be delivered to the cloud ever. An example of divergence in the format can happen if the developers coding the cloud application do not pay attention to the message format. For example, if the `speed` is treated as a float, precision errors may occur or even the parsing may fail; or if the timestamp is treated as a long with the epoch time (instead of as a formatted string) even a runtime error may happen.

3 AsyncAPI: Towards a standard language for describing message-based architectures

Aforementioned consistency issues are not unique to message-based architectures where communication occurs asyn-

chronously. In fact, other architectures also manifest them, as it is the case of resource-oriented architectures where communication occurs synchronously. However, in these cases, the industry has already proposed standardized solutions to support the development of such architectures. An example is *OpenAPI* and its complete ecosystem. The *OpenAPI Specification* [41] is a description format for APIs based on the REST [19] paradigm that allows, among other things, to specify the operations offered by the API, the type and parameters of each operation, the authentication methods, etc.

For message-based architectures, and taking inspiration from *OpenAPI*, the *AsyncAPI Specification* [5] proposal has recently emerged as a promising alternative. *AsyncAPI* descriptions are expected to be both human and machine readable. To achieve this goal, files defining a message-driven API are represented as JSON objects and conform to the JSON standards[2]. Such files allow describing, among other things, the message brokers of an architecture, the *topics* of interest, or the different formats of the messages associated with each of the *topics*. Next, we introduce the most preeminent concepts—JSON objects in the *AsyncAPI Specification*—of the *AsyncAPI* proposal for future reference[3]:

The AsyncAPI object is the root document object of an API definition. It combines resource listing and API declaration together into one document. Its main fields are: *asyncapi*, to specify the *AsyncAPI Specification version* being used; *info*, an *Info* object; *servers*, a *Servers* object; *channels*, a *Channels* object; and *components*, a *Components* object.

The Info object provides the API metadata, such as its *title*, *version*, *description*, *termsOfService*, *contact*, and *license*.

The Servers object is a map of *Server* objects.

A Server object typically represents a message broker (or a similar computer program). This object is used to capture details such as URLs, protocols and security configuration of such brokers. Variable substitution is also supported. The object contains, among other fields, a *url* to the target host, its *protocol* (e.g., *http*, *mqtt*, *stomp*, *kafka*, etc.), the *protocolVersion*, a *description*, or a map of *variables*.

The Channels object is a map holding relative path names and individual *Channel Item* objects. *Channel* path names are relative to servers. *Channels* are also known as *topics*, *routing keys*, *event types* or *paths* depending on the protocol or technology used.

A Channel Item object describes the operations available on a single channel (i.e., *topic*). Typical fields are: *description*, to describe the channel; *subscribe*, an *Operation* object; *publish*, an Operation object too; or *parameters*, a map of the parameters included in the channel name.

An Operation object describes a publish or a subscribe operation. This provides a place to document how and why messages are sent and received. Most common fields are: *operationId*, a unique string used to identify the operation; *summary*, a short *summary* of what the operation is about; *description*, a verbose explanation of the operation; and *message*, a *Message* object with the definition of the message that will be published or received on this channel.

A Message object describes a message received on a given channel and operation. For a message, the following fields can be specified among other: *name*, a machine-friendly name for the message; *title*, a human-friendly title for the message; *summary*; *description*; or *payload*, which can be of any type but defaults to *Schema* object.

A Schema object allows the definition of input and output data types. These types can be objects, but also primitives and arrays. This object is a superset of the JSON Schema Specification Draft 7[4]. Typical fields of an *Schema* object are: *title*; *type* (any of *boolean*, *integer*, *number*, *string*, *object*, *array* or *null*); *enum*, to limit possible values from a list of literals; *properties*, to specify the fields of objects; *maxItems* and *minItems*, to specify the cardinality of arrays; or items, to specify the schema of the array elements.

A Reference object is a simple object which allows referencing other components in the specification, internally and externally. It only contains the *$ref* field, which is a URI.

The Components object holds a set of reusable objects for different aspects of the *AsyncAPI* definition. Elements defined within the *Components* object can be referenced by using a *Reference* object. Reusable objects are mapped by name in their corresponding field. Some examples are: *schemas*, for *Schema* definitions; *messages*, for *Message* definitions; *parameters*, for *Parameters*; or *operationTraits* and *messageTraits*, which are traits that can be applied to operations and messages, respectively, and are defined similarly to *Operations* and *Messages*.

[2] YAML, being a superset of JSON, can be used as well to represent an *AsyncAPI Specification* file too.

[3] An interested reader may find Fig. 4 useful, where we propose a metamodel for the *AsyncAPI Specification* based on the concepts presented here.

[4] https://json-schema.org/specification-links.html#draft-7

Listing 2: AsyncAPI specification for an IoTbox

```
{
 "asyncapi" : "2.0.0",
 "info" : {
  "title" : "IoTBox API",
  "version" : "1.0.0"
 },
 "servers" : {
  "production" : {
   "protocol" : "mqtt",
   "url" : "example.com:1883"
  }
 },
 "channels" : {
  "iotbox/{id}/monitor" : {
   "parameters" : {
    "id" : {
     "description" : "The ID of the IoTBox",
     "schema": { "type" : "string" }
    }
   },
   "publish" : {
    "operationId" : "publishStatus",
    "message" : {
     "$ref" : "#/components/messages/statusMessage"
    }
   },
   "subscribe" : {
    "operationId" : "subscribeStatus",
    "message" : {
     "$ref" : "#/components/messages/statusMessage"
    }
   }
  }
 },
 "components" : {
  "messages" : {
   "statusMessage" : {
    "description" : "Status of a given path",
    "payload" : {
     "$ref" : "#/components/schemas/pathInfo"
    }
   }
  },
  "schemas" : {
   "pathInfo" : {
    "type" : "object",
    "properties" : {
     "id" : {
      "type" : "string",
      "description" : "Identifier of the path"
     },
     "belts" : {
      "type" : "array",
      "description" : "Info of belts in this path",
      "items" : {
       "$ref" : "#/components/schemas/beltInfo"
      }
     }
    }
   },
   "beltInfo" : {
    "type" : "object",
    "properties" : {
     "id" : {
      "type" : "string",
      "description" : "Identifier of the belt"
     },
     "ts" : {
      "type" : "string",
      "title" : "Timestamp"
     },
     "speed" : {
      "type" : "integer",
      "description" : "Speed of the belts in m/s"
     }
    }
   }
  }
 }
}
```

Example Listing 2 shows, in a simplified way, how part of our running example is specified using AsyncAPI. To keep the example manageable, we have specified only the monitoring part (thus excluding the configuration topics—e.g., `iotbox/box1/config`—and associated messages). As it can be seen, in line 2, we specify that the definition adheres to the *AsyncAPI Specification* version `2.0.0`, while in lines 3–6, we specify the information of our API. As lines 7–12 specify, our infrastructure has a single server, called `production` – whose host name is `example.com`—with an MQTT broker listening on port `1883`. The rest of the *AsyncAPI* object specifies the topics exposed by our API, and the format of the messages that can be interchanged. This is done via the `channels` property (lines 13–34), which in turn, references some reusable artifacts that have been defined within the `components` property (lines 35–79). Line 14 specifies the name of the only channel—i.e., topic—of our infrastructure: `iotbox/{id}/monitor`. As it can be guessed, `iotbox/{id}/monitor` is a parameterized name, in which the `{id}` substring is substituted by the actual IoTBox name when publishing a message (thus publishing either under the `iotbox/box1/monitor` or `iotbox/box2/monitor` topics). Lines 16-19 specify the actual information of the parameter: in line 16, we specify its name; in line 17, we provide a description; and in line 18, we specify its type. The publish and subscribe operations are specified in lines 21–26 and 27–32, respectively. As both operations publish and receive the same kind of messages, they reference the reusable definition named `statusMessage`, which is defined in the `messages` property of the `components` object (lines 36–43). The payload of a `statusMessage` is a `pathInfo` object. The schema of `pathInfo` objects is specified in lines 45–60. A `pathInfo` is a JSON object with two properties: `id`, a string value; and `belts`, an array of `beltInfo` objects. As specified in lines 61–77, a `beltInfo` is an object with three properties: an `id`; a timestamp (whose name is `ts`); and a numeric value with the speed of the belt. As it can be observed, the example message shown in Sect. 2—i.e., Listing 1—contains a `pathInfo` object conforming to the specification in lines 45–77.

As it can be observed in the example, the *AsyncAPI Specification* allows defining all the relevant information needed to design and execute a message-driven API. However, as of writing this manuscript, *AsyncAPI* is only an emerging proposal and its ecosystem is still flourishing. Some documentation and code generators exist—with different maturity levels[5]—but still no integrated development workflow to design and implement message-driven architectures has been proposed since transitions between different development tasks must be manually done.

[5] See https://shapeup.asyncapi.io to follow the development progress of the different tools under the umbrella of the *AsyncAPI Initiative*.

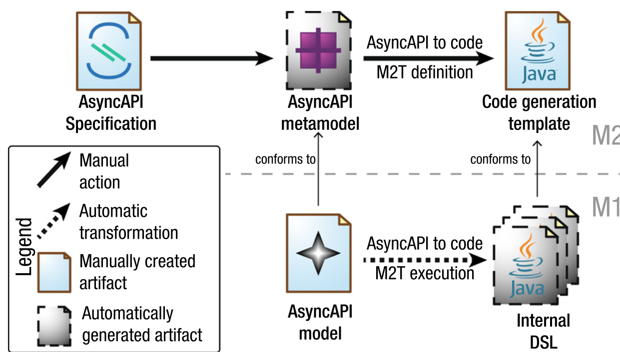

Fig. 2 Generic model-driven process where AsyncAPI serves as the *single source of truth*

4 Model-driven development of asynchronous message-driven architectures

One of the major flaws of message-driven architectures, as Sect. 2 illustrates, is how easily the knowledge about the infrastructure dilutes among all the elements involved in it. As a consequence, it is very easy to introduce divergences on how messages are sent and consumed by the different actors involved. This major flaw can, however, be solved if we use models as the *single source of truth* used throughout the design, development and execution of the infrastructure.

As we have seen in Sect. 3, despite its current limitations, *AsyncAPI* provides the grounds to design a complete conceptual framework. But, in order to make an effective use of *AsyncAPI*, a proper process and infrastructure supporting not only the design, but also the whole life cycle of a message-driven architecture, is still needed; models are the perfect asset to overcome these current limitations. Based on our experience, using model-driven techniques allow us (i) to keep our approach modular and extensible, alleviating a possible *vendor lock-in*; (ii) to integrate other IoT standards, different programming languages and frameworks, or future projects with not yet known requirements at *Ikerlan*; and (iii) to boost our productivity by taking advantage of the plethora of model-driven technologies and solutions available in the market—metamodeling frameworks, code generation engines, model transformation engines, etc.

Figure 2 shows what a typical model-driven workflow—with the main involved artifacts represented as a paper sheet—looks like. On the one hand, artifacts depicted with a solid line represent manually created artifacts, while artifacts depicted with dashed lines are automatically generated using different transformations. On the other hand, artifacts at the M1 layer (the model layer) are defined—or automatically created—each time a new development process is enacted, while artifacts at the M2 layer (the metamodel layer) are defined only once during the development of the toolkit itself.

The figure represents that, based on the *AsyncAPI Specification*, a corresponding *AsyncAPI metamodel* can be created for its abstract syntax. Using the concepts of this abstract syntax, a model-to-text (M2T) transformation generating executable code can be defined—*AsyncAPI to code*. Thus, by executing this M2T transformation, a library managing all basic functionality of a message-driven infrastructure—that will be shared by all the elements participating in it—can be generated from an *AsyncAPI model*. This library can be implemented in such a way that it exposes an internal *domain specific language* (DSL) that can be used by the *client code*[6] to perform tasks such as message creation, message parsing and processing, publications, subscriptions, etc.

4.1 The *AsyncAPI Toolkit*: Implementing a model-based development workflow from the beginning

While this generic workflow provides the grounds for a minimal working solution, it is impractical since it does not consider the concrete syntax for *AsyncAPI definitions* (which are JSON-based according to the specification) and assumes designers would take care of manually creating the AsyncAPI models from the JSON-based descriptions.

Instead, Fig. 3 shows how we envision a completely practical solution for an improved workflow, extending the previous building blocks and covering all the way from the concrete syntax to the code generation phase. We have implemented this workflow as an open-source solution called *AsyncAPI Toolkit* also following model-driven development principles: instead of manually developing an *AsyncAPI metamodel*, we have taken advantage of the *Xtext framework* [53] to provide both a concrete and an abstract syntax for *AsyncAPI*.

Thus, based on the *AsyncAPI Specification* introduced in Sect. 3, we have developed an *AsyncAPI JSON grammar* in Xtext—see Sect. 5.1—that validates message-driven API definitions conforming to the *AsyncAPI Specification*. Likewise, from this grammar, Xtext automatically generates the corresponding *AsyncAPI metamodel* and all the tooling—editor with content assist, parser, etc.—to easily create and transform *AsyncAPI JSON definitions* into *AsyncAPI models* conforming to the *AsyncAPI metamodel*—see Sect. 5.2. Given the availability of both a metamodel for *AsyncAPI* and the specification of an API as a conformant model, the workflow can continue as explained above, i.e., by executing a M2T transformation that generates an internal DSL.

[6] We understand as *client code* the actual applications making use of the messages sent and received, and whose logic cannot be captured in the *AsyncAPI definition*. An example of client code would be the application running in the *cloud*, which is in charge of processing the monitoring data sent by the IoTBoxes.

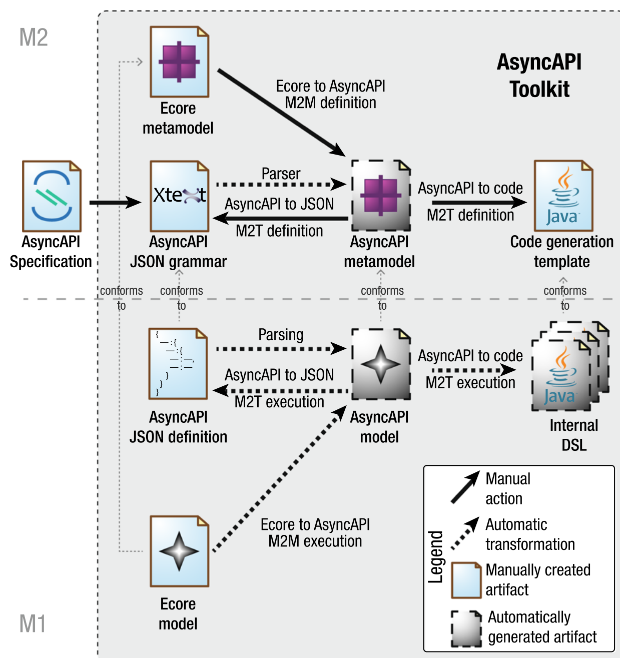

Fig. 3 Actual model-driven development process implemented by *AsyncAPI Toolkit* with its main involved artifacts

At this moment, the *AsyncAPI Toolkit* supports the Java language and generates a library that assists developers in the creation, publication and reception of well-formed messages by providing a *fluent API*—see Sect. 5.5—as proposed by [20].

4.2 Bootstrapping a message-driven architecture from data models

It is noteworthy that, since these architectures are message-based, data modeling plays a crucial role. For this reason, we have completed the above workflow with an alternative (graphical) concrete syntax focused on the modeling of the messages to be exchanged. Thus, instead of using *JSON Schema*, messages can be directly specified using *Ecore models* which conform to the *Ecore metamodel* provided by the *Eclipse Modeling Framework* (EMF) [51]. Such *Ecore models* can be further annotated to capture extra information—channels, servers, etc.— that may be needed to specify a fully working *AsyncAPI definition* (see Sect. 5.3 for an example). Nevertheless, it must be pointed out that *JSON Schema* is more expressive than plain *Ecore*—e.g., think about the *anyOf*, *oneOf*, *allOf* or *not* keywords, to name a few—and the same applies to the annotations we have made available in Ecore, which have been limited to the bare minimum. In these cases, **data modeling is conceived as a means to bootstrap an *AsyncAPI JSON definition* that can be later manually refined** thus reducing the initial development time[7]. Besides its limitations, obtaining a JSON-based representation of an *Ecore model* poses several advantages: (i) allows developers and architects to create a working *AsyncAPI* definition without requiring deep knowledge of the specification while (ii) keeps the modeling environment simple and manageable; and (iii) allows staying compliant with the *AsyncAPI Specification* for those unfamiliar with modeling (iv) also enabling experienced developers and architects to refine and complete details of the architecture that cannot be captured using Ecore in an easy way[8].

In order to integrate data models in the proposed development workflow, we have defined both the *Ecore to AsyncAPI* model-to-model (M2M) and the *AsyncAPI to JSON* M2T transformations shown in Fig. 3: the former transforms an annotated data model to its equivalent *AsyncAPI model*, while the latter serializes back the abstract definition of the *AsyncAPI model* to its JSON-based representation. Once we have both an AsyncAPI model and its JSON-based representation, the workflow can proceed as explained in Sect. 4.1.

Example An architect willing to use AsyncAPI as the *single source of truth* in our warehouse use case would proceed as follows. The architect would create a data model of the messages exchanged among all the devices in the message-driven architecture. After adding the desired annotations for the servers, channels, and messages, an equivalent *AsyncAPI model*—which is automatically transformed to the corresponding *AsyncAPI JSON definition*—is generated[9]. If we consider only the monitoring part of our use case, in practice, this definition is exactly the one we show in Listing 2 since our proposal fully complies with the *AsyncAPI Specification*. While the user is editing, our *AsyncAPI Toolkit* creates the corresponding *AsyncAPI model* and executes the *AsyncAPI to code* M2T transformation generating the internal DSL on-

7 We use the term *bootstrapping* because we transform a less expressive definition—an annotated Ecore data model—to a more expressive definition—the *AsyncAPI JSON definition*—which will act as the new starting point of the development process. In this sense, and for the sake of simplicity, once an *AsyncAPI JSON definition* has been created, modifications on it cannot be ported back to an Ecore data model because of its different levels of expressiveness.

8 Alternatively, data modeling in the *AsyncAPI Toolkit* could be achieved by providing a UML profile to annotate UML class diagrams. While this possibility has not been yet discarded and can be considered complementary, for the current purposes plain annotations have been considered enough.

9 Of course, the architect can start the development process by directly creating the *AsyncAPI JSON definition*, but as aforementioned, starting from data models allows bootstrapping the development process with less effort as we justify in Sect. 6.3.3.

the-fly. The generated DSL—which is an executable library exposing a fluent API in Java—can be directly distributed in source code form or as packaged binaries to the developers of the different components of the architecture (e.g., the IoTBoxes, the cloud, or the frontend). Thus, a developer wanting to publish a message or consuming a message does not need to care about other elements in the architecture or external documentation: all he or she has to do is to import the libraries of the DSL. The DSL will provide all the functionality needed to connect to a specific *broker*, create a specific message in the right format, and publish it, or vice versa: connect to the specified *broker*, subscribe to a specific topic, and receive the messages in the right and native format of the platform being used.

5 The *AsyncAPI Toolkit* under the microscope

We have implemented the *AsyncAPI Toolkit* workflow as an open-source solution,[10] and as highlithed above, we have followed model-driven development principles to create it. Instead of manually developing a set of editors supporting the textual *AsyncAPI JSON Grammar* or the *AsyncAPI metamodel*, we have chosen Xtext to provide both a concrete and an abstract syntax for *AsyncAPI*. Furthermore, we take advantage of all the EMF ecosystem to develop the rest of the toolkit. This is the case for our proposal for message modeling—which uses existing Ecore editors—and all the M2M and M2T transformations shown in Fig. 3, namely *Ecore to AsyncAPI*, *AsyncAPI to JSON* and *AsyncAPI to code*.

Next, we describe in detail how our *AsyncAPI Toolkit* has been built[11] and how architects and developers can take advantage of it.

We will start by explaining both the *AsyncAPI JSON grammar* in Xtext and the automatically generated *AsyncAPI metamodel*, since these are central assets of our proposal. Next, we will explain how to bootstrap an *AsyncAPI JSON definition* from an Ecore-based data model. And we will end the Section by explaining how the *fluent API* providing the internal DSL is generated and used.

[10] https://hdl.handle.net/20.500.12004/1/A/ASYNCAPI/001.

[11] Only short illustrative excerpts of the different elements of the *AsyncAPI Toolkit* will be shown for brevity purposes. For a full reference, please check our repository.

5.1 A JSON-based concrete syntax for AsyncAPI

The main manual step that Xtext requires to provide a concrete syntax for a textual language is the development of an Xtext grammar. Listing 3 shows an excerpt of the grammar[12] we have defined to support the definition of AsyncAPIs in JSON following the concrete syntax proposed in [5].

In short, we have defined an Xtext rule for each one of the concepts defined in *Schema* section of the aforementioned document. Listing 3 shows—in a simplified way—the rules to define in JSON an AsyncAPI specification version 2, with its *Info*, a set of *Servers*, *Channels*, and the *Components* Section. Taking as an example the *AsyncAPI* rule (line 1), in line 2 we specify that the application of the rule will produce an *AsyncAPI* object when parsing an input text, while lines 3 and 12 specify that a *AsyncAPI* is a textual element enclosed between the characters `{` and `}`. The parentheses in lines 4 and 11 denote an unordered group, i.e., the patterns between them, which are separated by an `&` symbol, may match only once and in any possible order. Line 5, for example, specifies that the version of an *AsyncAPI* is a sequence of characters starting with the `"asyncapi"` keyword, followed by a `:` symbol and followed by a text matching the *VersionInfo* rule (which in turn, is an *enumerated*, specifying that only version `"2.0.0"` is supported). The value of the parsed version number will be stored in an attribute of the *AsyncAPI* object named *version* of type *VersionNumber*. It is necessary to clarify some details of the grammar: first, in order to get advantage of the features provided by the Xtext unordered groups, we have defined the commas between groups as optional (`','?` expression near the end of each group), and second, we have relaxed the requirements of some elements marking them as optional (`?` symbol at the end of each group) to minimize the number of errors reported while parsing the input files for not overwhelming the users of the tool[13]. These optionalities can be, however, later enforced programmatically so that the tool only accepts valid JSON instances.

[12] See https://www.eclipse.org/Xtext/documentation/301_grammarlanguage.html for a full reference of Xtext grammars.

[13] Since Xtext performs a *live validation* of the document being edited, it is necessary to allow inconsistent intermediate states of the underlying model while creating it: if hard requirements are set at the metamodel level, most of the errors reported will refer to parts of the document not added yet, rather than to real errors of the already written parts.

Listing 3: Excerpt of the Xtext grammar

```
1  AsyncAPI:      // Rule name
2    {AsyncAPI}  // Output type name
3    '{'         // The curly brace character
4    (
5        ( '"asyncapi"' ':' version=VersionNumber ','? )
6      & ( '"info"' ':' info=Info ','? )
7      & ( '"servers"' ':' '{' servers+=Server (','
         servers+=Server)* '}' ','? )?
8      & ( '"channels"' ':' '{' channels+=Channel (','
         channels+=Channel)* '}' ','? )?
9      & ( '"components"' ':' components=Components ','?
          )?
10     // More properties omitted for brevity
11   )
12   '}';
13
14 // Info rule omitted for brevity
15
16 Server:
17   {Server} name=STRING ':' '{' (
18       ( '"url"' ':' url=STRING ','? )
19     & ( '"protocol"' ':' protocol=Protocol ','? )
20     & ( '"description"' ':' description=STRING ',' )?
21     // More properties omitted for brevity purposes
22   ) '}';
23
24 // Channel and Components omitted for brevity
25
26 enum VersionNumber:
27   _200 = '"2.0.0"';
28
29 enum Protocol:
30   amqp = '"amqp"' | amqps = '"amqps"' | mqtt = '"mqtt
       "'
31   | mqtts = '"mqtts"' | ws = '"ws"' | wss = '"wss"'
32   | stomp = '"stomp"' | stomps = '"stomps"';
33
34 terminal STRING:
35     '"' ( '\\' . | !('\\'|'"') )* '"'
36   | "'" ( '\\' . | !('\\'|"'") )* "'";
37
38 // Other rules omitted for brevity
```

Example As it can be seen, the *AsyncAPI* definition included in Listing 2 can be parsed by applying the rules of the grammar showed in Listing 3: the curly brace in line 1 of Listing 2 matches the curly brace in line 3 of Listing 3; the tokens in line 2 of Listing 2 (`"asyncapi": "2.0.0"`) match the tokens specified in line 5 of Listing 3; etc.

5.2 An abstract syntax for AsyncAPI

Based on the grammar above, Xtext is able to generate an equivalent EMF-based metamodel. Figure 4 shows the *Ecore* metamodel generated as a result of the *AsyncAPI JSON grammar*. As aforementioned, instances conforming to this metamodel are automatically created out of textual descriptions thanks to the tooling generated by Xtext. As it can be observed, it basically contains a class for each one of the rules defined in the Xtext grammar, and each one of the classes contains the attributes specified in its corresponding rule. For example, among other elements, Fig. 4 shows the *AsyncAPI* and *Server* classes which correspond to the rules included in Listing 3, as well as the enumerations for *VersionInfo* and *Protocol*. The existence of this automatically generated metamodel enables the use of any other EMF-based tool, such as model transformation engines.

It is noteworthy to mention that we have been extremely careful when defining the *AsyncAPI JSON grammar* so that the generated EMF-based metamodel closely represents the domain—i.e., the *AsyncAPI Specification*—and not only its textual representation. Thus, **this metamodel is not only an utility artifact that enables us to parse *AsyncAPI JSON specifications*, but it is a meaningful metamodel close to the concepts of the *AsyncAPI Specification***: as it can be seen, there is a direct match between it and the main concepts explained in Sect. 3, and as a consequence, we will not describe in detail the concepts shown in the figure to avoid being redundant.

Example The result of parsing the *AsyncAPI* definition in Listing 2 is partly shown in Fig. 5. As it can be seen, the automatically generated model contains a single root *AsyncAPI* object, containing *Info*, *Server*, *Channel* and *Component* objects. The properties of the `production` *Server* are shown in the *Properties* tab. As the *AsyncAPI* definition in Listing 2 specifies, the name of the *Channel* object is `iotbox/{id}/monitor`, and it contains the operations `publishStatus` and `subscribeStatus`, and the *Parameter* `id`. The *Components* object contains the *Schemas* `pathInfo` and `beltInfo`, and the *Message* `statusMessage`.

5.3 An Ecore-based concrete syntax for data modeling to bootstrap a message-driven architecture

As it can be seen in Listing 2, the specification of the messages format is the longest and most verbose part of a message-driven architecture definition, even for the simplest use case. This is even more evident in real-world scenarios, where we can find tens or hundreds of different messages in the same architecture. For this reason, we propose to allow bootstrapping an *AsyncAPI*-based development process by modeling the different messages that may be sent and received. Since *Ecore* is the core (meta)modeling language provided by EMF —also having a stable and adequate tool support—we have decided to adopt it as the most convenient way to provide data modeling capabilities in our *AsyncAPI Toolkit*. In short, *Ecore* can be considered as an implementation of the *Essential MOF (EMOF) Model* proposed by the OMG [40].

Figure 6 shows an excerpt of the Ecore (meta)modeling language[14]. In the figure, classes with a gray background are abstract, while classes with a lighter yellow background are concrete. As it can be seen, *Ecore* is a constrained subset of the UML2 [38] class models—as EMOF also is.

[14] Since the *AsyncAPI Toolkit* focuses on data modeling, Fig. 6 only focuses on structural modeling primitives. Other primitives—such as *EOperations*—have been omitted for brevity purposes.

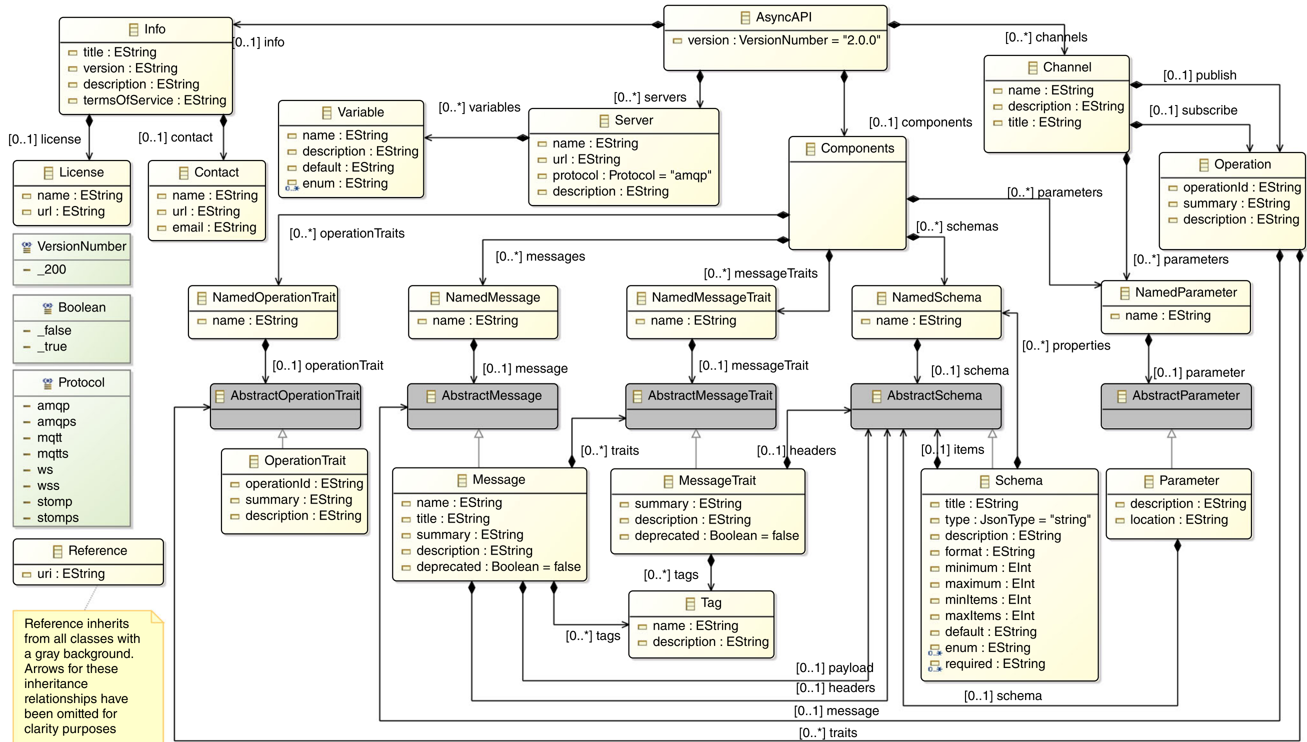

Fig. 4 AsyncAPI metamodel

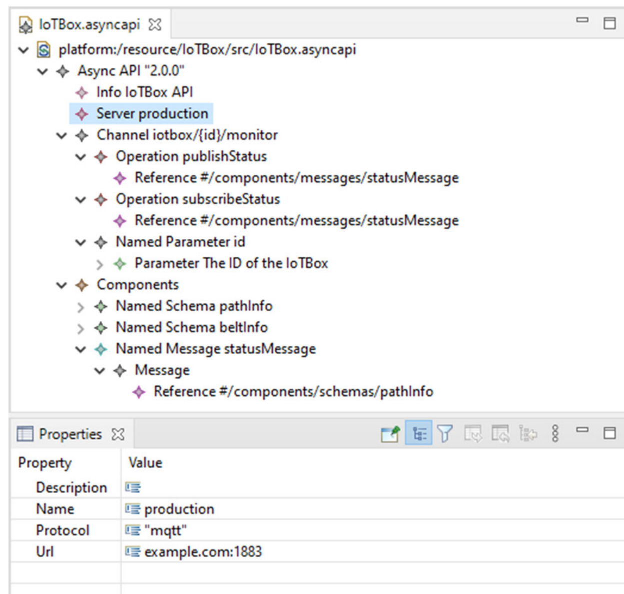

Fig. 5 *AsyncAPI* definition of the factory use case represented as an instance of the *AsyncAPI metamodel*

In short[15], an *Ecore* model is composed of *EModelElements*, which can be either elements with a name (*ENamedElements*) or annotations (*EAnnotations*). Among the *ENamedElements* we find: *EPackages*, *EClassifiers*, *ETypedElements* and *EEnumLiterals*. *EPackages* are used to group *EClassifiers*. *EClassifiers*, which can be either *EClasses* and *EDataTypes*, are used to represent and classify domain entities. *ETypedElements*—and *EStructuralFeatures*—are used to represent properties of domain entitites via *EAttributes*, or relationships among domain entities via *EReferences*. Regarding *EAnnotations*, any *EModelElement* can have any number of them. *EAnnotations* are identified by its *source* attribute and contain a set of *details* in the form of a *key-value* map. In *Ecore*, *EAnnotations* have been used since the beginning as the standard way to customize automated processes which take as an input a model[16]. Thus, in *AsyncAPI Toolkit* we have adopted this same approach to complement data models with additional information that will allow obtaining a fully working *AsyncAPI* definition.

Table 1 shows the annotations currently supported by the *AsyncAPI Toolkit*. Information about *Servers* is directly added to the *EPackage* containing the data model. There can be as many *Server* annotations as servers are in the architecture. For each server, we can specify its *name*, its *url*—including the listening *port*—and the supported *protocol*. Details about *Channels*, *Messages*, and *Schemas* can be all added in the corresponding *EClass*. This is because: (i)

[15] For an extensive explanation of *Ecore*, see [49].

[16] For example, *EAnnotations* are used to include model constraints that can be checked at runtime. *EAnnotations* are also used to tweak the behaviour of the standard code generation templates.

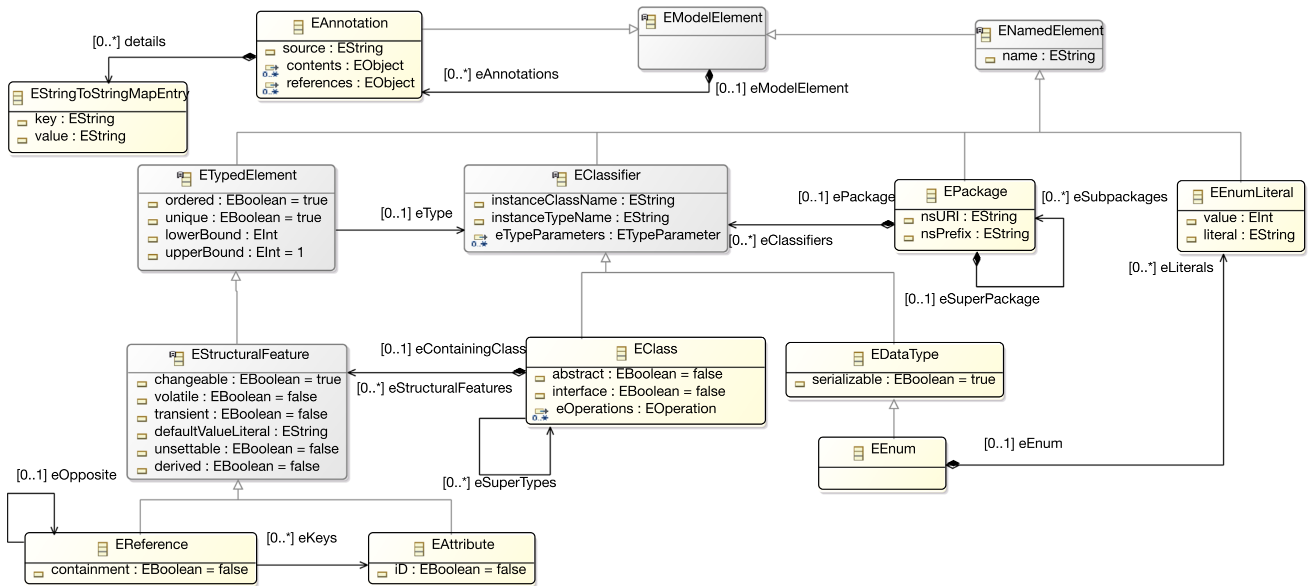

Fig. 6 Subset of the Ecore metamodel

Table 1 ecore annotations

ECore element	EAnnotation source[a]	description	keys
Epackage	`Server`	list of servers	`name`: server name
			`url`: server url, including port
			`protocol`: Asyncapi supported protocol
Eclass	`Channel`	the eclass represents the payload of a given channel	`name`: channel name
			`description`: channel description
			`publish`: publish `operationid`
			`subscribe`: subscribe `operationid`
			`parameters`: comma-separated list of parameters used in the channel `name`, if any
Eclass	`Message`	the eclass represents the payload of a given message	`name`: message name
Eclass	`Schema`	the eclass represents a payload	`name`: schema name
			`title`: friendly name of the schema
EStructuralfeature	`Schema`	the estructuralfeature is a property part of a payload	`title`: friendly name for the property

[a] to avoid redundancies, we have omitted the common part of the *eannotation source*—an identifier in the form of a uri, which in this case is http://io.github.abelgomez/asyncapi/eannotations/—for all elements in the table (e.g., `server` should be read as http://io.github.abelgomez/asyncapi/eannotations/server)

EClasses represent domain entities, which in our case are *Messages* and their corresponding payload (i.e., *Schema*), and (ii) a single *Channel* refers to a single *Message*—for both its publish and subscribe operations—and a single *Message* refers to a single *Schema*.

As a consequence, all this information can be added in the *EClass* itself. For a *Channel*, we can specify its *name*, a *description*, the publish and subscribe *operationIds*, and the *parameters*, if any; for a *Message*, we can specify its name; and for a *Schema* we can specify its *name* and its *title*. It is noteworthy to mention that most of these annotations are not strictly required[17]. This may happen for two reasons:

(i) because a default value can be directly derived from the plain *Ecore* model; or (ii) because the property we are specifying is optional in the *AsyncAPI Specification*.

The former is what happens when an *EClass* is annotated as a *Channel*, and no *Message* nor *Schema* annotations are

[17] See our online documentation for details about optional and required annotations.

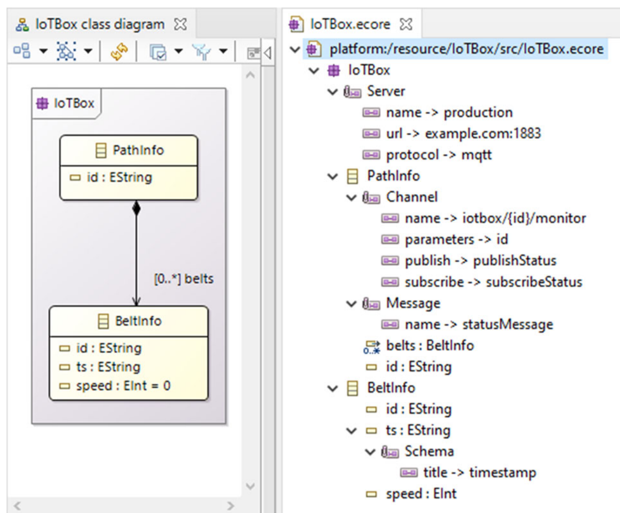

Fig. 7 Ecore representation of the IoTBox example

added: in that case, *Message* and *Schema names* are derived from the *EClass* name). An example of the latter case is the *Schema EAnnotation* when applied to an *EStructuralFeature*: *title* is optional in the *AsyncAPI Specification* and thus is not required.

Example Figure 7 shows what the messages of our running example look like whenmodeled using *Ecore*. The left-hand side of the figure shows how the data model is represented in a graphical way, while the right-hand side of the figure shows the same model represented in a tree viewer with the corresponding *EAnnotations*. As it can be seen, we have a single *EPackage* called `IoTBox` which contains two *EClasses*: `PathInfo` and `BeltInfo`. `PathInfo` has an `id` *EAttribute* and contains a set of `BeltInfos` via the `belts` *EReference*. A `BeltInfo` has three *EAttributes*: `id`, `ts`, and `speed`. This data model perfectly matches the JSON Schema expressed in lines 45–77 of Listing 2.

In the right-hand side of the figure, we have included the *Servers* information directly in the `IoTBox` *EPackage*. Since `statusMessages` are published and subscribed under topic—i.e., *Channel* – `iotbox/{id}/monitor`, and `pathInfo` is the *Schema* of `statusMessage`, we have added the corresponding *Channel* and *Message EAnnotations* to the `PathInfo` *EClass*. Since the name of the `PathInfo` *EClass* matches[18] the `pathInfo` *Schema* name, there is no need to add the *Schema EAnnotation*. Finally, the *Schema* annotation in the `ts` property of `BeltInfo` indicates that we want to refer to this property as `timestamp` rather than `ts`.

Listing 4: Excerpt of the *Ecore to AsyncAPI* M2M transformation

```
1 public static final String BASE_EANNOTATION_URI = "http://io.github.abelgomez/asyncapi/eAnnotations/"
2 public static final String EANNOTATION_SERVER = BASE_EANNOTATION_URI + "Server"
3
4 static def AsyncAPI asyncApi(EPackage ePackage) {
5   return AsyncApiFactory.eINSTANCE.createAsyncAPI => [
6     version = VersionNumber._200
7     info = AsyncApiFactory.eINSTANCE.createInfo => [
8       title = ePackage.nsPrefix
9       version = "1.0.0"
10    ]
11    servers += ePackage.allServers
12    channels += ePackage.allChannels
13    components = AsyncApiFactory.eINSTANCE.createComponents => [
14      messages += ePackage.allMessages
15      schemas += ePackage.allSchemas
16    ]
17  ]
18 }
19
20 static def Iterable<Server> allServers(EPackage ePackage) {
21   return #[
22     ePackage.servers,
23     EcoreUtil2.getAllContentsOfType(ePackage, EPackage).map[servers].flatten
24   ].flatten
25 }
26
27 static def Iterable<Server> servers(EPackage ePackage) {
28   return ePackage.EAnnotations.filter[source == EANNOTATION_SERVER].map [
29     val details = details
30     AsyncApiFactory.eINSTANCE.createServer => [
31       name = details.get("name")
32       url = details.get("url")
33       protocol = Protocol.getByName(details.get("protocol"))
34     ]
35   ]
36 }
```

[18] Letter case conventions differ between *Ecore EClasses*—where names must start by an uppercase letter—and JSON Schemas—where names usually start by a lowercase letter. As a consequence, the name `PathInfo` in Ecore is considered equivalent to `pathInfo` in JSON.

5.4 From an annotated data model to an AsyncAPI definition

Once an annotated data model has been created, it can be automatically transformed to a valid *AsyncAPI definition*. As depicted in Fig. 3, this transformation is done in two steps: first, the *Ecore to AsyncAPI* M2M transformation converts the annotated model into an *AsyncAPI model*, and second, the *AsyncAPI to JSON* M2T transformation serializes the *AsyncAPI model* into a JSON document.

To implement both transformations, we have used *Xtend* [52]. *Xtend* is a flexible and expressive dialect of Java, which compiles into readable Java compatible source code. *Xtend* is the language which provides code generation support to the *Xtext* framework, and as a consequence, transformations in this language can be very easily integrated in our *AsyncAPI Toolkit*.

Listing 4 shows an small excerpt of the *Ecore to AsyncAPI* M2M transformation. Lines 4–18 show the rule to create an *AsyncAPI* object from an *EPackage* object. Specifically, in line 5 we specify that we will return a new *AsyncAPI* object created by calling the factory method `createAsyncAPI`. Then, we make use of the *with* operator (=>) to initialize this new object by executing the code between square brackets. Code between square brackets is a lambda expression that is executed over the left-hand-side argument of the *with* operator returning it. Thus, `version`, `info`, `servers`, `channels` and `components` are not local variables, but properties of the new *AsyncAPI* object. This pattern to create and initialize objects is repeated throughout the whole transformation for its simplicity, readability and conciseness (e.g., see lines 7–10, 13–16 and 30–34). Line 6 shows a simple initialization of the AsyncAPI *version* property from a static value (`VersionNumber` is an enumeration, and `_200` a literal); lines 7–10 and 13–16 show the initialization of properties with nested objects (an *Info* object and a *Components* object, respectively); line 8 shows the initialization of a property from a primitive value extracted from the source *Ecore* model; and lines 11, 12, 14 and 15 show the initialization of properties by delegating the creation of the target object to other transformation rules. In this case, two important features of *Xtend* are exploited: *extension methods* and *infix operators and operator overloading*. *Extension methods* allow adding new methods to existing types without modifying them: when invoking a method that can receive arguments, instead of passing the first argument inside the parentheses of the method invocation, the method can be called with the first argument as its receiver. *Infix operators* together with *operator overloading* allow *Xtend* to express the *add* operation in a collection as +=. Considering these two features, we can see that `servers += ePackage.allServers` is indeed equivalent to `servers.addAll(allServers(ePackage))`.

Method `allServers` applies the `servers` method to the *EPackage* passed as an argument and all its nested *EPackages*. The former calculation is done in line 22, while the latter is done in line 23 (the `.map[...]` function allows applying the operation passed as argument to all the elements of a given collection). Then, an immutable list is created with the result of both transformations via the `#[ ... ]` operator, and this list is finally returned after being flattened (lines 21–24).

Finally, the `servers` method (lines 27–36) creates a list of *Servers* from a given *EPackage*. To do it, first we extract all *EAnnotations* in the *EPackage* whose *source* is `Server`, and afterwards, for each one of these *EAnnotations*, we create a new *Server* from its *details*[19].

The rest of the model is created by executing the transformation rules in lines 12, 14 and 15—which, in turn, call other transformation rules.

Listing 5: Excerpt of the *AsyncAPI to JSON* M2T transformation

```
1  static def CharSequence generate(AsyncAPI asyncAPI)
2  '''
3  {
4    "asyncapi": «asyncAPI.version»,
5    "info": «asyncAPI.info.generate»,
6    «IF !asyncAPI.servers.empty»
7    "servers": {
8      «asyncAPI.servers.map[generate].join(",\n")»
9    },
10   «ENDIF»
11   «IF !asyncAPI.channels.empty»
12   "channels": {
13     «asyncAPI.channels.map[generate].join(",\n")»
14   },
15   «ENDIF»
16   "components": «asyncAPI.components.generate»
17 }
18 '''
19
20 static def CharSequence generate(Info i)
21 '''
22 {
23   "title" : "«i.title»",
24   "version" : "«i.version»"
25 }'''
26
27 static def CharSequence generate(Server s)
28 '''
29 "«s.name»" : {
30   "url" : "«s.url»",
31   "protocol" : "«s.protocol.getName»"
32 }'''
```

Once an *AsyncAPI model* has been obtained, serializing it to its JSON-based representation is straightforward. In order to define the *AsyncAPI to JSON* M2T trasnformation, we have used *Xtend* again, but now taking advantage of its *template expressions*. Templates are surrounded by triple single quotes (`'''`) and allow for readable string concatenation. A template expression can span multiple lines, and expressions can be nested using guillemets—i.e., «`expression`».

[19] We must assign the *details* property of the *EAnnotation* to a local variable (i.e., `val details = details` in line 29) because the *details* property is not available in the scope of lines 31–33 (which is *Server*, and not *EAnnotation*).

Listing 5 shows the transformation rules that serialize the *AsyncAPI* (lines 1–18), *Info* (lines 20–25) and *Server* (lines 27–32) objects. In this case, notice that we also make use of *extension methods* to improve the readability of the transformation. As aforementioned, text enclosed between three single quotes—with the exception of the bits enclosed between guillemets «...»—is copied as is in the final textual file. Text between guillemets is evaluated, and its `toString()` representation is automatically inserted at the expression position. It is noteworthy that *templates* allow conditions as shown in lines *6–10* and *11–15*. In those cases, the text enclosed between the `IF` and `ENDIF` markers will only be serialized if the condition is met.

Example By applying the *Ecore to AsyncAPI* M2M transformation, we are able to transform the data model depicted in Fig. 7 into the *AsyncAPI model* depicted in Fig. 5. As it can be seen, the rules shown in Listing 4 would allow to create the *AsyncAPI*, *Info* and *Server* objects shown at the upper part of Fig. 5 out of the *EPackage* declaration in Fig. 7. Afterward, after applying the *AsyncAPI to JSON* M2T transformation, the *AsyncAPI model* in Fig. 5 can be serialized as it is shown in Listing 2. For example, the transformation rules shown in Listing 5 are the ones that generate lines 1–12 and 80 of Listing 2: lines 3 and 17 in Listing 5 produce lines 1 and 80 in Listing 2; line 4 in Listing 5 produces line 2 in Listing 2; lines 5 and 20–25 in Listing 5 produce lines 3–6 in Listing 2; and lines 7–9 in Listing 5 produce lines 7–12 in Listing 2.

5.5 An internal Java DSL for effective message-driven communication

The last step of our proposed workflow is the generation of a library implementing an internal DSL, which can be used and shared among all the elements in the infrastructure, and that will ensure that all participants know the same topics and are able to create and consume messages conforming the same schemas and formats. Currently, our *AsyncAPI Toolkit* only supports the generation of Java code, but other languages and technologies can be easily plugged in by providing the proper code generation templates.

The M2T transformation applies the following rules:

- A Server is chosen as the default *Server* of the architecture.
- Channels are transformed to packages whose name corresponds to the *Channel*—or topic—name. Hierarchies of topics—i.e., using slashes (/)—are respected.
- Operations in *Channels* are transformed to classes inside the packages of their corresponding *Channels*. Classes for channel *Operations*, among other things, provide static operations for publishing and subscribing in their respective topics using the default *Server*. If *Channel* permits the use of *Parameters*, a nested class is generated to manage the parameter substitution and recovery.
- Messages do not generate a Java artifact: since there is a direct correspondence between *Messages* and *Schemas* (via the *payload* association), information about *Messages* is integrated within the corresponding *Schema* when needed.
- The Components object is transformed to a `components` package.
- Schemas are transformed to immutable classes: they are cloneable and serializable (via Gson[20]), have getters but no setters, and have a private constructor. Instances are created via a dedicated nested *builder* class.

  Schema classes can be generated directly in the `components` package if they are reusable, or nested. For example, an *Operation* can have nested declarations for its corresponding *Message* and *Schema*. In that case, the *Schema* class is declared as a nested class within the *Operation* class.

Listing 6: Excerpt of the code generation template

```
1 def String schemaClassBuilder(Schema s, String
    schemaClassName)
2 '''
3 public static class «schemaClassName»Builder {
4
5   private «schemaClassName» instance = new «
      schemaClassName»();
6
7   public static «schemaClassName»Builder newBuilder()
      {
8     return new «schemaClassName»Builder();
9   }
10  «FOR p : s.properties»
11
12  «p.namedSchemaInitializerMethods(schemaClassName)»
13  «ENDFOR»
14
15  public «schemaClassName» build() {
16    try {
17      return («schemaClassName») instance.clone();
18    } catch (CloneNotSupportedException e) {
19      throw new RuntimeException("Unable to build: "
          + this, e);
20    }
21  }
22 }
23 '''
```

Listing 6 shows a very small excerpt of the code generation template. Specifically, it shows the template method generating the builder in charge of creating instances of a *Schema* class (which, when serialized, will be the payloads of the messages sent and received). As shown in line 1, this method receives as an input the *Schema* instance as declared in Fig. 4, and the *Schema* class name[21]. As it can be seen in line 3, the template generates a class, whose name is the one of the

[20] https://github.com/google/gson.

[21] This is because JSON Schemas do not have a name, as a consequence, the class name has to be calculated from other elements, such as the *map* containing it, the *Message* for which it is a *payload*, the *Operation* sending or receiving it, or any other element of the architecture.

Schema class plus the suffix `Builder`. The builder contains a private instance of the *Schema* class (line 5), which will be the one being modified and returned to the client code once the build process finishes. The builder class contains a static factory method (`newBuilder()`, lines 7–9), and a builder method (`build()`, lines 15–21) that finishes the build phase and returns a immutable copy of the instance of the *Schema* class. As it can be seen in lines 10–13, the template generates an *initializer method* for each one of the properties of the schema. These are the methods that the client code must call between the factory method and the build method: since the builder class is nested within the *Schema* class, and the *Schema* class does not have setter, these *initializer methods* will be the only ones allowed to initialize and modify the instances of the *Schema* class. After that, they will remain immutable.

Listing 7: Main class of the IoTbox

```
1 public static class PathInfoBuilder {
2
3   private PathInfo instance = new PathInfo();
4
5   public static PathInfoBuilder newBuilder() {
6     return new PathInfoBuilder();
7   }
8
9   public PathInfoBuilder withId(String id) {
10    this.instance.id = id;
11    return this;
12  }
13
14  public PathInfoBuilder addToBelts(BeltInfo beltsElt
       ) {
15    this.instance.belts.add(beltsElt);
16    return this;
17  }
18
19  public PathInfo build() {
20    try {
21      return (PathInfo) instance.clone();
22    } catch (CloneNotSupportedException e) {
23      throw new RuntimeException("Unable to build: "
           + this, e);
24    }
25  }
26 }
```

Example Listing 7 shows what the generated code for the builder class looks like for the `pathInfo` *Schema*. As explained in Sect. 3 and Listing 2, a `pathInfo` is an object with two properties: `id`, a string; and `belts`, an array of `beltInfo` objects. Thus, from the `pathInfo` *Schema* a `PathInfo` class is generated (not shown) with a `PathInfoBuilder` nested class. It can be seen that the generated code contains the private instance of the `PathInfo` (line 3), the factory method (lines 5–7), the builder method (lines 19–25), and two initializer methods. The first initializer method (lines 9–12) is in charge of setting the `id` of the `PathInfo`, which is a Java String. Since `id` is single-valued, the initializer method is prefixed with the word `with`. The second initializer method (lines 14–17) is in charge of initializing the list of `BeltInfo` objects. Since `belts` is many-valued, the initializer method is prefixed with the words `addTo`.

As it can be seen, the library provides specific artifacts for the most important concepts of the *AsyncAPI definition*: *operations* on each *channel—topic*—are unequivocally grouped in their specific packages; *payloads* of messages can only be created by using specific *builders*; and *messages* can only be sent and received using dedicated methods of specific classes.

Forcing developers to use builder classes and *getters* of the generated *Schema* classes—among other things—guarantees that the structure and the type of the messages are preserved and shared among all the elements of the architecture. The same applies, for example, with respect to the classes for publishing and subscribing since the relationship between *Channels*, *Operations*, *Messages* and *Schemas* is explicitly encoded in the internal DSL: static typing in Java ensures that developers do not mix incompatible types, or publish (or receive) messages in the wrong topics[22].

Example Listings 8 and 9 show two examples of client code using the internal DSL in Java generated for our example *AsyncAPI definition* (Listing 2), and more specifically, Listing 8 shows the code needed to create and publish the example message shown in Listing 1. As it can be seen in the imports (lines 5–6), there exists a `iotbox._id_.monitor` package (generated from the `iotbox/{id}/monitor` *Channel*), with a `MonitorPublish` class (generated from the `monitorPublish` operation). Since the `iotbox/{id}/monitor` *Channel* has an `id` parameter, also a `PublishStatusMessageParams` is created in order to perform the substitution and recovery of parameter values. It can also be seen that reusable *Schemas* (`pathInfo` and `beltInfo`) have produced the corresponding classes in the `schemas` package (see lines 8–9).

Thus, in order to create the message, developers only need to use the provided classes. For example, to create an instance of `PathInfo`, a new `PathInfoBuilder` can be obtained by invoking `PathInfo.newBuilder()` (line 13), and then, it can be initialized by using the provided *fluent interface* [20] (e.g., methods `withId` and `addToBelts`). If any of the methods needs another object as an argument—such as for `addToBelts`—it can be created by using the corresponding builder as shown in lines 15–21. When a friendly name is available for a given *property*—for example, because a *title* was specified—the method provided by the *fluent interface* will use it instead of the actual *Schema* property name. Lines 18 and 25 are an example of this: timestamps are a property called `ts`, but the provided method is `withTimestamp` rather than `withTs`, thus making the code more understandable. Once all the properties have been

[22] In any case, these checks can be proactively done in the generated code following a *fail fast* approach in the case these features are not natively provided by the language of the generated code (e.g., dynamically typed languages).

set, the `build()` method is invoked. It is noteworthy to mention that validation logic—such as checking of required properties—could also be added in the `build()` method following a *fail fast* approach.

Listing 8: Main class of the IoTBox Publisher

```
1 package main;
2
3 import java.text.MessageFormat;
4 import java.time.Instant;
5
6 import iotbox._id_.monitor.PublishStatus;
7 import iotbox._id_.monitor.PublishStatus.
      PublishStatusParams;
8 import schemas.BeltInfo;
9 import schemas.PathInfo;
10
11 public class MonitorPublish {
12   public static void main(String[] args) throws
         Exception {
13     PathInfo payload = PathInfo.newBuilder()
14         .withId("Path_A")
15         .addToBelts(
16           BeltInfo.newBuilder()
17             .withId("Belt_A1")
18             .withTimestamp(Instant.now().toString())
19             .withSpeed(10)
20           .build()
21         )
22         .addToBelts(
23           BeltInfo.newBuilder()
24             .withId("Belt_A2")
25             .withTimestamp(Instant.now().toString())
26             .withSpeed(15)
27           .build()
28         )
29       .build();
30     PublishStatusParams params =
31         PublishStatusParams.create().withId("box1");
32     System.out.println(MessageFormat.format(
33       "Publishing on topic ''{0}'':\n{1}",
34       PublishStatus.expand(params),
35       payload.toJson()
36     ));
37     PublishStatus.publish(payload, params);
38   }
39 }
```

Listing 9: Main class of the IoTBox Subscriber

```
1 package main;
2
3 import java.text.MessageFormat;
4 import iotbox._id_.monitor.SubscribeStatus;
5
6 public class MonitorSubscribe {
7   public static void main(String[] args) throws
         Exception {
8     SubscribeStatus.subscribe((message, params) -> {
9       System.out.println(MessageFormat.format(
10        "Message received from IoTBox ''{0}''!",
11        params.getId()
12      ));
13      System.out.println(MessageFormat.format(
14        "Info about production line ''{0}'':",
15        message.getId()
16      ));
17      message.getBelts().stream().forEach(
18        belt -> System.out.println(
19          MessageFormat.format(
20            "Belt {0} was running at {1} m/s at {2}",
21            belt.getId(),
22            belt.getSpeed(),
23            belt.getTimestamp()
24          )
25        )
26      );
27    });
28  }
29 }
```

Finally, once the payload of the message and the parameters have been created—parameters are created using a similar *fluent interface* as shown in lines 30–31—the `publish` operation can be invoked (see line 37). The `publish` method of the `MonitorPublish` class only accepts instances of `PathInfo` as the first argument and instances of `PublishStatusMessageParams` as the second argument. The publish operation will be in charge of doing the parameter substitution, and publishing the payload passed as an argument in the `iotbox/box1/monitor` topic. This will ensure that both the payload and the topic names will be syntactically correct and will match.

Listing 9 shows how an example application—such as one running in the *cloud*—will subscribe to the `iotbox/+/monitor` topic and will receive messages sent to it. As it can be seen, it only needs to invoke the `SubscribeStatusMessage.subscribe` method passing a callback function (expressed as a lambda expression in lines 8–27): the callback function will receive the value of the parameters in the `params` argument—which is of type `SubscribeStatusMessageParams`—and the message payload in the `message` argument—which is of type `PathInfo`. From this point on, client code can make use of the getters provided by the generated code to retrieve all the information from them. As it can be seen, the example code only prints all the received information by the standard output.

6 *AsyncAPI Toolkit* Evaluation

The aim of this section is to evaluate the *AsyncAPI Toolkit*. As we discuss in Sect. 7, there are no other tools focusing on the modeling of AsyncAPI definitions based on data models that allow generating fully functional executable code out-of-the-box. Thus, we focus on evaluating the perceived usability and usefulness of our tool, and its strengths and weaknesses, based on short interviews with developers using *AsyncAPI Toolkit* in internal projects and with the results of an empirical experiment involving additional participants.

More specifically, the evaluation of the tool focuses on answering the following research questions:

- RQ1 Which are the perceived benefits of using *AsyncAPI Toolkit* to define and develop an asynchronous message-driven architecture?
- RQ2 Is it possible to use *AsyncAPI Toolkit* without previous knowledge of the tool? How much time is necessary to be autonomous and to obtain significant results, i.e., a running asynchronous architecture?
- RQ3 Which are the weaknesses and strengths of *AsyncAPI Toolkit*?

RQ4 Does using a graphical syntax for defining the *AsyncAPI definition* provide any benefits?

In order to answer these questions, we have conducted two different evaluations of *AsyncAPI Toolkit*. On the one hand, we have used *AsyncAPI Toolkit* internally in a number of projects and, as a result, a first set of conclusions and lessons learned have been drawn, as explained in Sect. 6.2, answering RQ1. On the other hand, an empirical external validation has also been carried out once the internal developers considered *AsyncAPI Toolkit* was mature enough. The aim of this external validation was twofold: first, quantifying the perceived benefits of using *AsyncAPI Toolkit* when developing a message-driven architecture, specially regarding learning and development time taking into account the previous experience of the subjects building similar projects by hand, and thus answering RQ2; and second, gathering feedback and educated opinions in order to answer RQ3 and RQ4. Details of this empirical evaluation are described in Sect. 6.3.

6.1 Threats to Validity

Before describing the evaluation, we first discuss the main threats to validity that can affect the result of the validations presented next. In this regard, the *construct validity*, *internal validity*, and *external validity* must be taken into account [56].

Regarding the **construct validity**, as described in the introduction, the aim of the *AsyncAPI Toolkit* is to help with the development of IoT projects. Therefore, to be able to evaluate the usability of the toolkit the main questions to be answered are related to the ease of use for developers of such projects. The four RQs are focused on evaluating exactly that, and in the following subsections the questions in the questionnaire are linked to the different RQs.

To be able to avoid bias in the **internal validity**, the weight of the evaluation is on the external validation. This is why although we developed a first internal validation focusing on our own experience using the *AsyncAPI Toolkit* in internal projects, an external validation was also done with people with different profiles and no previous involvement in the development of the toolkit.

The cornerstone of the performed evaluation is gathering relevant information from the subjects that conducted the experiment for **external validity**. The information was related to weaknesses and strengths, users impressions about usability and the perceived usefulness of the presented toolkit in order to answer the previously mentioned research questions. The external evaluation was mainly focused on interviews and questionnaires, as it allowed to collect relevant data from the subjects. As subjects are the basis of the evaluation, they also become the main threat.

Our threats are mainly associated with the participants, the number of subjects—which could have been increased to increase the soundness of the results—and the scales used for the measurements. In the *First Phase*, the subjects of our experiment could have a different previous knowledge before participating in the experiment. We tried to alleviate this issue in two ways: (i)we explicitly asked for this information in the survey to consider this variable in our experiments—although the claimed level of expertise is anyway subjective since each participant's scale may be different—and (ii) we tried to cover a diverse set of profiles in the selection by choosing engineers, developers, staff in training, researchers, etc., to gather different points of view.

On the other hand, since modeling tools used in the *Second Phase* were not developed as part of this experiment, we tried to focus the experiment on the contribution of the *AsyncAPI Toolkit* by selecting those subjects with previous expertise in MDD. As aforementioned, our idea is to avoid a bias toward the *AsyncAPI Toolkit* functionality caused by existing modeling tools, since non-expert users may be unable to distinguish preexisting tools from the *AsyncAPI Toolkit* itself.

Related to the scalability of the exercise to real-world problems, even though the example being developed in the evaluation phase was limited by the timing, it represented complex communication data that can be found in real world and that can be easily adapted to real scenarios.

6.2 Internal Validation

AsyncAPI Toolkit has been developed in the context of the MegaM@Rt2[23] project and it has already been applied to some of its use cases. It has also been tested in some internal projects at *Ikerlan* and it is being used for the development of a use case in the COMP4DRONES project[24]. This section aims to answer RQ1 (perceived benefits of the *AsyncAPI Toolkit*) and discusses some of the reflections and benefits observed during the development of these projects.

Lower development and deployment time.

Adopting the *AsyncAPI Toolkit* in a project significantly decreases the time to develop and deploy the software system. On the one hand, reusing the generic metamodel simplifies the definition of the schemas of the project and, with the automated code generation, there is no need to implement the boilerplate code in the client side, reducing manual time-consuming tasks. In fact, our observations show that the time to develop the infrastructure code has been reduced to nearly one-third of the initial time. Obviously, there is an initial cost in developing the *AsyncAPI Toolkit* itself, training the people in using it, and adapting the continuous integration infrastructure so that all components share the same version of the library implementing the internal DSL. But this cost

[23] https://megamart2-ecsel.eu/.

[24] https://www.comp4drones.eu/.

is quickly compensated when using it over several projects, as it happens with any new MDE infrastructure [14].

Increased code quality.

In addition to lowering the development time, the generated code is better structured and allows to reuse common code blocks, which increases the overall code quality. The reused parts are already tested; hence, the verification process of the system is simpler. Moreover, the generated code is also easier to maintain, as bugs or improvements in common parts need to be addressed only once. This leads to a reduction on engineering and maintenance costs.

In this sense, it has been detected that using the *AsyncAPI Toolkit* in different projects, the time to detect bugs has been decreased as more bugs are detected in the initial stages of the implementation and execution of the developed systems.

DSL benefits for Industry 4.0.

Another upside is that, as diverse existing application domains share similarities—especially regarding communication requirements—the same solution can be applicable to all of them. In this context, we can more easily port our *AsyncAPI*-based solutions to a variety of related domains. This is especially interesting in the context of software product lines (SPLs).

Easy documentation.

AsyncAPI includes a tool to automatically generate documentation out of an API. Although it is not integral part of our toolkit, it can be directly used side by side the *AsyncAPI Toolkit* since we stick to the *AsyncAPI Specification*. Thus, from a model or definition created by the *AsyncAPI Toolkit*, the *AsyncAPI* tool is able to generate the corresponding documentation in HTML, Markdown or React.

At *Ikerlan*, this feature has been regarded as an important aspect in the decision to move forward with the adoption of *AsyncAPI Toolkit*. This documentation capability, together with the use of *AsyncAPI* as *single source of truth*, enables all project participants (stakeholders, designers, architects, etc.) to share a common definition of the API, favoring interoperability and reducing the number of errors in the software development life cycle.

Requirements definition, validation and maintainability.

Being a message-driven architecture, capturing the requirements can be directly done with the *AsyncAPI definition*. Therefore, there is no need to maintain a separate text document that needs to be interpreted by developers, as the requirements are specified in the *AsyncAPI definition* itself.

This has been another key reason to select *AsyncAPI* for ongoing and future projects at *Ikerlan*. Combined with the previous point, we see *AsyncAPI Toolkit* as a toolkit able to support most phases of the development cycle, from requirements to code-generation to—in the future—testing.

6.3 External Validation

The following paragraphs will describe the process we have followed to conduct the external validation, describing in detail: (i) the experiment carried out; (ii) the set of samples; (iii) the threats; and (iv) the results or conclusions gathered from it.

6.3.1 Experimental design

The evaluation of the *AsyncAPI Toolkit* was carried out in two phases.

First Phase—For the first phase, 25 subjects were invited to implement with *AsyncAPI Toolkit* a small example of an industrial system that communicates asynchronously. The example used is the exact same one presented in Sect. 2: an automated warehouse with *paths*, *belts* and *speed* measurements, where monitoring data and configuration messages can be sent and received. To solve the exercise, the participants had to specify the complete architecture and the format of the messages in JSON, using the *AsyncAPI Toolkit* and following the *AsyncAPI* specification. After specifying the architecture, they had to implement two simple applications using the automatically generated classes based on the DSL—i.e., *fluent API*: one for publishing new messages and other for consuming messages. The participants were given a one-hour deadline to develop this exercise, knowing that the set time was tight. After trying to implement the example application, the subjects answered a questionnaire about their knowledge of the field, previous background, and their impressions on the use of the *AsyncAPI Toolkit* to solve the exercise. The goal of this first phase was to answer `RQ2` and `RQ3`.

The questionnaire consisted on a set of questions listed in Table 2 that evaluate the performance and usability of *AsyncAPI Toolkit*. The questions aimed to assess the following aspects: (i) rate the previous knowledge; (ii) score the ease of use of the tool; (iii) evaluate the ease of installation; and (iv) measure development time. Questions #1 to #8 mostly aimed at answering `RQ2`. Questions #9 to #16 mostly aimed at rating the level of satisfaction with the different components and features of the tool, also including some open questions to gather additional suggestions or comments, and allowing us to dig deeper into the possible improvements thus answering `RQ3`.

Second Phase—After the first phase, results were analyzed and a second phase of the experiment was conducted based on them. This second phase consisted on evaluating the data modeling extensions of the *AsyncAPI Toolkit*—which are described in Sects. 4.2 and explained in detail in Sects. 5.3 and 5.4—with the goal of answering `RQ4`. Not all subjects participated in this second phase of the experiment: since modeling tools used in the *Second Phase* are part of the

Table 2 Phase 1 questionnaire

Question #	Question	Possible answers
1	What is your previous knowledge on:	–
	a) Java	1 to 5
	b) Asynchronous communication	1 to 5
	c) MQTT	1 to 5
	d) Eclipse	1 to 5
	e) AsyncAPI	1 to 5
	f) MDD (metamodels, models, ...)	1 to 5
2	How difficult has been to carry out the assigned task?	1 to 5
3	How much time did you spend installing the *AsyncAPI Toolkit*?	Number of minutes
4	How much time did you spend learning or acquiring the main concepts (knowledge of the subject)?	Open answer
5	Did you finish the exercise?	Yes / No
6	How much time did you spend implementing the example? (in case you finished)	Number of minutes
7	What did you manage to finish? (in case you didn't finish)	–
	a) Implement config payload	Yes / No
	b) Implement two config messages (publish and subscribe)	Yes / No
	c) Implement monitor payload	Yes / No
	d) Implement the two monitor messages (publish and subscribe)	Yes / No
	e) Implement one of the applications (publish or subscribe)	Yes / No
	f) Implement both applications (publish and subscribe)	Yes / No
8	How much time do you estimate you would require to finish the whole application? (in case you didn't finish)	Number of hours
9	What do you think about the tool?	1 to 5
10	How intuitive and easy to learn would you say the tool is?	1 to 5
11	Do you think the workflow of the tool is the correct one?	Open answer
12	Would you add any feature to the tool?	Open answer
13	Please rate your level of satisfaction with the following aspects of The *AsyncAPI Toolkit*?	–
	a) Ease of installation	1 to 5
	b) Overall reliability	1 to 5
	c) Overall performance	1 to 5
	d) Completeness and organization of documentation	1 to 5
	e) Clarity of documentation	1 to 5
	f) Ease of updating the application	1 to 5
14	Considering your complete experience with the software, how likely would you be to recommend its use to a friend or colleague?	1 to 5
15	Do you think using the tool reduces development and deployment time? How?	Open answer
16	Please add any additional comments/suggestions if you wish	Open answer

Eclipse environment and were not developed as part of this experiment, we focused on the contribution of the *AsyncAPI Toolkit* by selecting those subjects with previous knowledge in MDD—these were 11 subjects, almost half of the initial ones. We did this selection to avoid a bias toward the *AsyncAPI Toolkit* functionality caused by existing modeling tools. The exercise consisted in obtaining the *AsyncAPI* definition for the automated warehouse architecture by creating an annotated model of the messages of the system. For this, the subjects used Eclipse and the editor of their choice to create and annotate an Ecore model, generating the corresponding *AsyncAPI* definition with the *AsyncAPI Toolkit*.

Since the set of subjects was smaller and easier to manage than in the first phase, short interviews with the subjects were carried out during this phase. In order to guide these interviews, a list of predefined questions was specified. Interviews gave us the opportunity to better focus the questions and gather more valuable feedback. The questions guiding these interviews aimed at answering `RQ4` are listed in Table 3. The answers given by the subjects were analyzed one by one, and grouping the same main ideas afterwards. Different details

Table 3 Phase 2 interview question guide

Question #	Interview Questions
1	What do you think about this feature [the graphical syntax] of the *AsyncAPI Toolkit*?
2	If you compare phase 1 way of implementing the AsyncAPI specification and this one, what advantages and disadvantages do you find?
3	Do you think that using this feature to define the system facilitates the understanding of the whole system? Or on the contrary, do you think the way used in the first phase is easier and more understandable?
4	Do you think that this feature adds value to the tool? How?
5	Do you miss something to be included in the tool?
6	How would you rate this feature? And the whole tool?

that could suggest improvements or concerns were also considered for future work.

6.3.2 Sample

Participants were selected from the personnel—engineers, developers, staff in training—of *Ikerlan* and *Universitat Oberta de Catalunya* who responded positively to a personal invitation. In any case, no subject was directly or indirectly involved in the development of the *AsyncAPI Toolkit* or this proposal. The sample consisted of 25 subjects with very diverse knowledge in fields and tools like Java, asynchronous communications, MQTT, Eclipse, *AsyncAPI* or model-driven development (MDD). As listed in Table 2, their expertise level was captured in the questionnaire. A 1 to 5 rating scale was used to quantify their knowledge, where 1 means no knowledge and 5 means being an expert.

The chart in Fig. 8 depicts the participants' knowledge on the mentioned fields. As shown in the chart, subjects came from different backgrounds. Very few people had previous knowledge of the *AsyncAPI* specification, while for the rest of the fields, the previous knowledge was more evenly distributed.

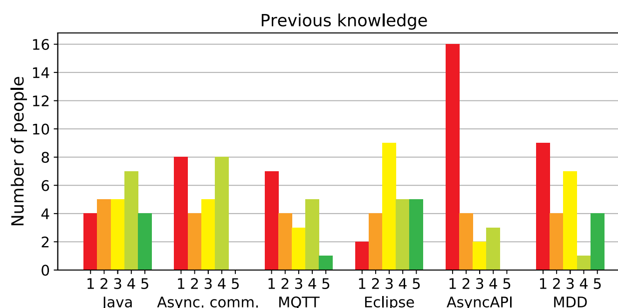

Fig. 8 Previous knowledge of the participants

6.3.3 Results

This section summarizes the results of the data collected with the questionnaire in the first phase and the guided interviews of the second phase.

First Phase—After analyzing the data, our experiment shows that **the 36% of the subjects were able to complete the experiment** by implementing the full use case, and for the remaining 64%, **another 36% estimated that would require only one additional hour**. These time estimations are based on the subjects perceptions but also on real data. The exercise was divided in six small sections with similar difficulty and required development time. The 36% that estimated one additional hour to finish the exercise, did complete at least 3 sections, and more than half of them, completed 4 or 5 sections (as they stated in question number 7). The 20% of the subjects estimated between 2 and 4 hours while only the 8% thought would require more than 4 hours to complete the whole exercise. This reveals that **only the 28.0% of the subjects estimated they would require more than 2 hours to finish the experiment**. Taking into account the complexity of the exercise, as described above, we believe this is a very good result.

Looking at these percentages in detail, the data did not reveal a direct or close relationship between the subject's previous knowledge in a specific field and the time to complete the full use case. Subfigures in Fig. 9 present this data, where it can be observed that there is not a direct correlation between knowledge on the different specific topics with the time to finish, but more generally experienced participants were more successful. This is the case for Java (Fig. 9a), Asynchronous Communication (Fig. 9b), MQTT (Fig. 9c), Eclipse (Fig. 9d) and *AsyncAPI* (Fig. 9e). However, we detected an exception with the subjects that had previous knowledge in MDE, as they were more comfortable, and their first example implementation was more successful than people with little or no knowledge. Figure 9f confirms this trend. Those with little or no knowledge of MDE required more time to develop the use case (or estimated so), while those with more expertise finished or were close to finish in the fixed time.

What our experiment also confirms—thus answering `RQ2` (previous knowledge and learning time)—is that the learning curve is smooth: as Fig. 11 shows, the training to learn the *AsyncAPI Toolkit* basics is not specially long or hard.

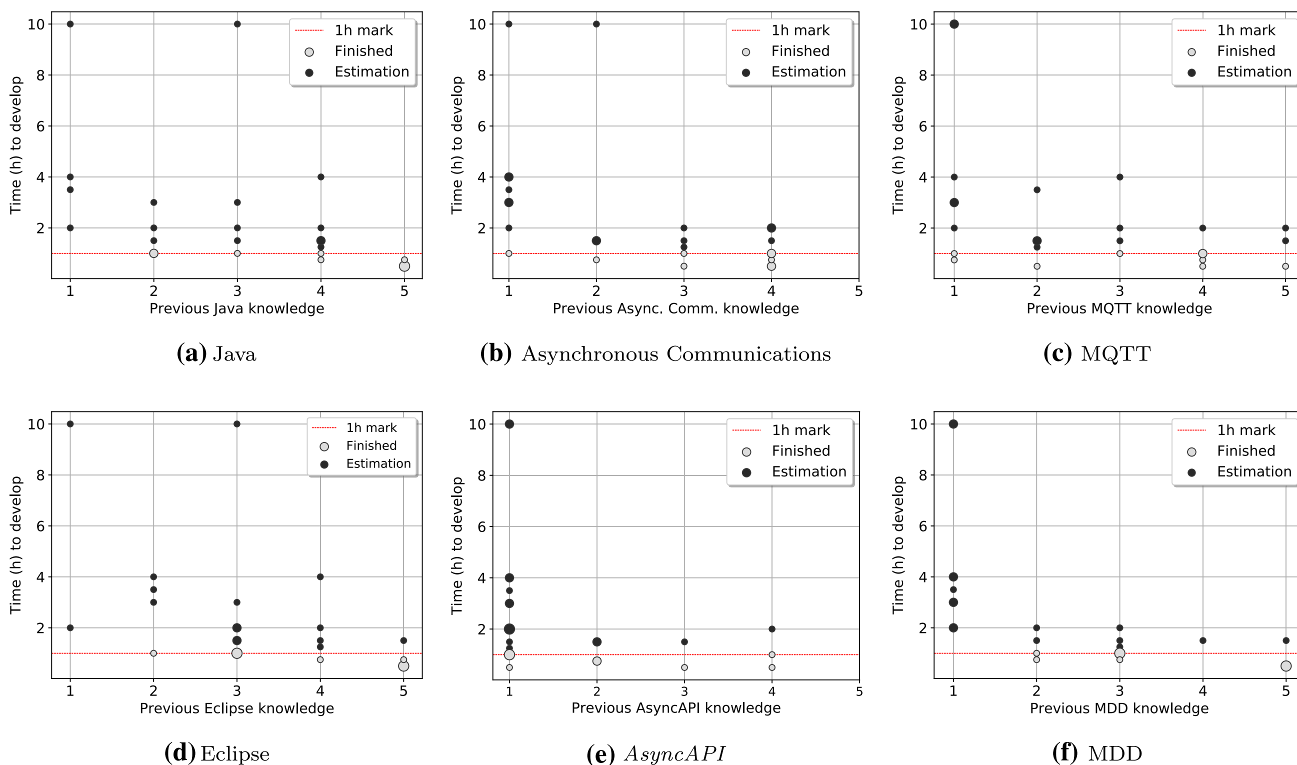

Fig. 9 Time to finish the exercise shows no direct correlation with previous knowledge on **a** Java, **b** asynchronous communication, **c** MQTT, **d** Eclipse, **e** *AsyncAPI*, and **f** MDD

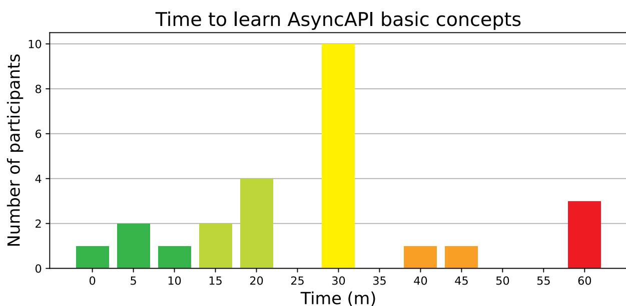

Fig. 10 Qualitative results

The chart depicted in the figure shows the time spent by the participants to read and get the basic knowledge about the *AsyncAPI* specification and *AsyncAPI Toolkit* to be able to start with the exercise. **In average, the subjects only required ≈28 minutes to get familiar with the specification and the tool** to be ready to start with the proposed exercise.

Figure 10 shows other evaluated facets related to *AsyncAPI Toolkit*, as clarity of the documentation and the overall reliability among others. Those facets were rated in a 1 to 5 scale—from unsatisfactory to very satisfactory—respectively. The obtained results are very positive.

Most of the subjects found the tool fairly intuitive (as we have just seen, the time to get familiar with the tool is quite small). The documentation completeness and organization, along with its clarity, have been positively rated. The subjects were also asked about the ease of installing and updating the application, with overall positive responses, standing out the easiness of installation.

Regarding the overall reliability and performance, few participants rated the *AsyncAPI Toolkit* low. Only around 4% of the participants found that it needs to improve its reliability and performance. Additionally, they were also asked about the development workflow around the tool. Almost all of them (92%) answered that the workflow of the *AsyncAPI Toolkit* is the correct one, with the missing 8% equally divided between negative and "don't know" answers, 4% each.

In general, we observed that *AsyncAPI Toolkit* **was highly rated on a global scoring and that most participants would recommend using it for implementing solutions that leverage asynchronous communications**.

The open questions allowed us to answer RQ3 (weaknesses and strengths of the toolkit), and dig deeper in the overall perception of the *AsyncAPI Toolkit* and helped us to find aspects for improvement. In this regard, one of such questions was whether the subjects think the tool reduces the development and deployment time and how. **The subjects unanimously answered that** *AsyncAPI Toolkit* **helps reducing development and deployment time**. Also, given

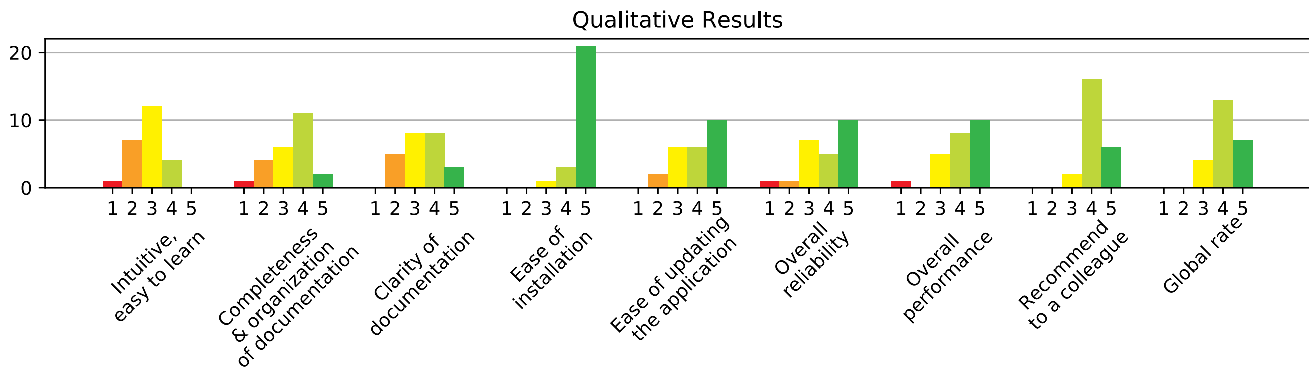

Fig. 11 Time spent learning the *AsyncAPI* basics

the overlapping between RQ1 (perceived benefits) and RQ3 (weaknesses and strengths), these open questions allowed us to confirm the results of the internal validation with the answers given by the external subjects. The following points were highlighted in the answers[25]:

- ***Automatic code generation*** makes faster and less error prone the development process while saving time. *"Everything concentrates in a single file and once it is filled and validated, the code is automatically generated"*.
- ***Model-based development*** simplifies the definition of complex tasks reducing time. *"Not implementing the classes* [of the DSL] *saves time"*.
- ***Focus on design***. Users can focus on designing the system and its API. *"Developers waste less time on configuring and managing the API workflow, and they focus on designing the API"*.
- ***Abstraction***. Users do not need to have big knowledge of the communication protocol, they only need to focus on defining the message content. *"The user only needs to define the content of the messages to be transmitted since the tool abstracts the user from knowing the details of the communication protocol"*.
- ***Modifications***. The tool facilitates modifications by the use of a *single source of truth*, as the classes are generated automatically, and only the *AsyncAPI* specification and the main application need to be changed. *"Automatic code generation reduces development time and using models facilitates modification reducing time too"*.
- ***Reuse***. What is generated can be reused in the application. The API definition can be easily reused and modified in similar projects too. *"You can reuse the generated API directly in your application"*.

Another open question was what features would the participants add to *AsyncAPI Toolkit*. The responses included pointers to possible improvements, e.g., adding a way to edit the model graphically or using a tree view, improving the *AsyncAPI* textual editor itself, improving error detection and providing feedback while editing, and adding support for more programming languages. Some of these points are further expanded in Sect. 8.

Second Phase—This section discusses our findings from the interviews carried out after the second exercise, where we asked the subjects to solve the proposed exercise by using an annotated *Ecore* data model, and answers RQ4 (benefits of the graphical syntax).

All subjects agreed that using a graphical editor to bootstrap a working specification was a more visual and helpful way to develop the *AsyncAPI* definition. However, some subjects highlighted that this way of modeling, i.e., using Ecore annotations, could appeal more to people with metamodeling experience.

One of the participants pointed that the learning curve was lower using the graphical editor and the specification could be defined quicker in addition to the benefit of using the same environment that they already use in other modeling tasks.

The model-driven approach provided by the tool was rated as useful and practical. Some subjects noted that the tool integrated seamlessly in Eclipse and that it helped any modeler to create *AsyncAPI* definitions and the corresponding scaffolding easily.

The participants mentioned several advantages related to the graphical editor. They are summarized in the following points:

- ***Less error prone***. The graphical view helps avoiding syntax and typing errors.
- ***Higher abstraction***. Users have a higher abstraction model that may facilitate understanding and implementing complex systems.
- ***Better user experience***. The majority agreed in that it was a better user experience. However, one participant with lesser modeling expertise preferred the textual editor.

[25] Quoted phrases have been directly extracted from subjects' declarations.

Meanwhile, the disadvantages found by those participants were:

- ***Burdensome editing procedure***. "Point-and-click" repetitive procedure when modeling and editing the *AsyncAPI* specification may become cumbersome.
- ***Requires knowledge of metamodel annotations***. While for people with expertise on metamodeling the feature was an advantage, participants without knowledge of metamodel annotations did not find it so easy.

Overall, **when subjects were asked about comparing both ways of specifying the system, they all unanimously agreed that using a graphical editor to create an annotated data model made it easier to understand the problem domain at a glance**. However, since the *AsyncAPI* specific information is embedded as annotations in the model, the user has to make an additional effort to understand how to map such information into the model elements. Also, it is not always clear how these annotations impact in the generated code, and this feature may require more knowledge and control on the modeled system.

As additional comments, the participants missed a kind of guided editor that facilitates the specification. For example, each time a channel is defined by the user, associated messages and schemas are automatically created even if no annotations for them are defined, and this is not obvious at first sight. Few participants brought up that documentation should be extended with other examples to facilitate the acquisition of knowledge. And a participant also pointed out the possibility to define an Unified Modeling Language (UML) profile for *AsyncAPI* metadata rather than annotations. This would ease the annotation phase using predefined stereotypes.

Regardless of this, everybody agreed the graphical editor clearly added value to the tool. As a participant noted, *"it is easier and faster to model using the tool than starting from scratch"*.

7 Related Work

In this section, we compare our proposal with other works around API specifications, IIoT languages and code generators, and model-based approaches both for data and communication.

As we will see, most of previous work that target modeling data and communication aspects focus on synchronous architectures. The support for message-driven ones, like the one we propose, is much more limited.

We first start by covering core modeling proposals for IIoT (Sect. 7.1). Then, we move to works focusing on model-based IIoT proposals relying on synchronous communication—mostly REST-based ones—in Sect. 7.2. Finally, model-based IIoT proposals on top of asynchronous architectures are covered in Sect. 7.3.

7.1 Domain-Specific Languages for IIoT

As we propose, MDE has already been used to accelerate the development process of industrial systems in the Industry 4.0 context. Among other works, the benefits of MDE for IIoT have been previously analyzed by Capilla et al. [8] and Young et al. [57], including some guidelines when modeling such systems to maximize their effectiveness [12].

Several works have explored the use of MDE in IoT at a global level. This is the case of Ciccozzi and Spalazzese [10], where the authors present the *MDE4IoT* framework supporting the modelling and self-adaptation of Emergent Configurations of connected systems. Clark and Barn [11] is a UML extension to define components that could be used in both service-oriented and event-driven architectures. AutoIoT [36] provides a generic metamodel and tool that designers can use to model their IoT scenarios either graphically (Web GUI) or textually (manually writing a JSON file). Models created with AutoIoT could then be the input of a model-to-text transformation to generate different implementations of the IoT application. ThingML [34] combines well-proven software-modeling constructs aligned with UML (statecharts and components) and an imperative platform-independent action language to specify IoT applications. Due to its popularity, other languages include a transformation to ThingML [46]. However, modeling of communication in ThingML is rather minimal. Similarly, Artikov et al. [4] defend the need to combine several modeling languages, each targeting a specific IoT concern, for a more consistent overall definition of the IoT system.

Indeed, these frameworks can be regarded as complementary to our own as they are more targeted to describing and implementing the IoT components themselves while we focus on describing the communication and interoperability among them. Next subsections cover proposals comprising as well the communication aspects, grouping them based on whether they mainly assume a synchronous or asynchronous communication.

7.2 Model-based approaches for synchronous communications

Several DSLs to model specific parts of IoT synchronous communication systems have been explored. One of such approaches is the one presented by Sneps-Sneppe and Namiot [48], where the authors present an extension of *Java Server Pages* to generate a web-based DSL to use in IoT applications. The proposed DSL enables IoT communications

between the devices that support the process and the sensors. Negash et al. [35] also propose a DSL that is specifically designed for IoT, namely *DoS-IL*. However, they go further and also created an interpreter for the DOM, allowing the browser to be manipulated through scripts that interact with the DOM. *CREST*, presented by Klikovits et al. [29], is another DSL that aims to model CPSs of small scale that has synchronous evolution and reactive behavior. Riedel et al. [44] present a tool that generates C, C# and Java code relying on SOAP Web Services (WS-SOAP) as the communication protocol. They propose the use of the *Essential Meta-Object Facility* (EMOF) [40] for data metamodels and EMF to generate the messages between IoT subsystems. Another approach is *TRILATERAL* [22,25], a tool that uses MDE with IoT communication protocols to generate artifacts for industrial CPSs. This tool allows using a visual editor to input a model based on the IEC 61850 standard for electrical substations and the tool automatically generates the C++ code that enables the devices to communicate, so far, using HTTP-REST, CoAP, or WS-SOAP (all of them synchronous).

A more recent initiative is the Web of Things (WoT) [55], which focuses on facilitating the interoperability among devices thanks to a shared definition of a common data model. A couple of modeling proposals around WoT are Delicato et al. [13] and Iglesias-Urkia et al. [23,24]. While WoT could support some level of asynchronous communication around its *event* concept, it mainly targets synchronous architectures.

Being RESTful architectures the most significant example of synchronous architectures, several approaches are implemented following this architectural style. For instance, Thramboulidis and Christoulakis [54] integrate CPSs and IoT with a framework named *UML4IoT* that allows automating the process of generating CPSs. To do that, the CPSs are modeled using SysML and implemented using an object-oriented API that is later transformed to a RESTful API, using LWM2M for the communication. Ivanchikj and Pautasso [26] present *RESTalk*, a DSL for modeling and visualizing RESTful conversations, i.e., a model of all possible sequences of HTTP message exchanges between client and servers. As before, RESTalk is based on the model of the *OpenAPI Specification* and provides a visual and textual DSL.

OpenAPI (formerly *Swagger*) [47] is a predominant effort to allow describing RESTful APIs [50]. The *OpenAPI* initiative offers several tools to assist developers—e.g., editors, document generators, code generators, etc.—proposed either by the consortium itself or by a growing ecosystem of third-party providers—e.g., APIs.guru [3]. We also start to see model-based tools for *OpenAPI* [16,17]. There are also some works that aim to infer API definitions from examples of calls or data points [7,15] and may offer an intermediate model to be annotated before the generation of the (OpenAPI) API description file.

OPC-UA [32] is a SOA-based popular alternative to RESTful architectures in the industrial automation domain. A core element in any OPC-UA definition is the information model aimed at representing the static and dynamic behavior of manufacturing systems. Several approaches, mostly based on UML extensions, have been proposed to facilitate the specification of such information models [30,42,43].

Many aspects of all these proposals have inspired key elements of the asynchronous approaches described next. The best example is *AsyncAPI* itself, which is heavily inspired by *OpenAPI*. However, due to the fundamental differences in the communication paradigm, previous approaches and tools cannot be directly reused in an *AsyncAPI* context as they are missing many of the key constructs to describe *AsyncAPI*—or more generally, asynchronous—definitions like channel, subscription, server or message.

7.3 Model-based approaches for message-driven architectures

Beyond *AsyncAPI*, asynchronous communication can also be defined with Data Distribution Service (DDS) [37], which is a data-centric publish/subscribe (DCPS) model for distributed application communication. While both DDS and *AsyncAPI* provide a specific language for data definition—the so-called Interface Definition Language (IDL) [39] in the case of the former, and JSON Schema in the case of the latter—they differ in different aspects: (i) DDS is oriented to the data itself and not the messages [2]; (ii) DDS is technology agnostic, and as a consequence, does not consider architectural elements such as the different protocols, servers and listening ports—to name a few—involved; and (iii) DDS has a more mature ecosystem.

However, although DDS is technology agnostic, its implementations are more niche-focused[26]: most of the solutions focus on supporting C and C++ and support to other languages are provided via bridges to execute native code in compiled libraries. For instance, OpenDDS[27] is an implementation of the DDS protocol that allows to define a data model with an IDL, and the code generator creates the data structure and a wrapper to access it. It creates C++ code that can be accessed using the Java Native Interface (JNI). Another commercial tool is IBM's Rhapsody[28], which allows to create DDS services in C++ or Ada. The final available tool worth mentioning is RTI Code Generator[29], which is a

[26] https://www.dds-foundation.org/dds-resources/.

[27] https://opendds.org/.

[28] https://www.ibm.com/docs/en/rhapsody/8.2?topic=tutorials-create-data-distribution-service-real-time-systems-application.

[29] https://community.rti.com/static/documentation/connext-dds/5.2.3/doc/manuals/connext_dds/code_generator/RTI_CodeGenerator_UsersManual.pdf

command line tool that uses an IDL or XML file as input and generates the code (Ada, C, C++, CLI, C# or Java) for the data structure, and both publisher and subscriber examples. However, the main downsides of environments supporting DDS is that they are less integrated, with a lower level of abstraction, and no support for internal DSLs—as *AsyncAPI Toolkit* proposes with its *fluent API*— is provided.

CyprIoT [6] includes modeling constructs to describe the communication channels and hierarchy of topics and subtopics but does not address the structure of the internals of the exchanged messages.

Finally, and specifically focusing on *AsyncAPI*, while the ecosystem of tools around it keeps growing at a fast pace[30], most of the tools still lack of some key features or are hard to integrate with external tools. For example, the *Java AsyncAPI* project[31] provides an implementation of the *AsyncAPI* specification in Java, similarly to what our *AsyncAPI metamodel* allows—*Ecore* metamodels are projected to Java code in EMF. However, although the *Java AsyncAPI* project allows dealing with AsyncAPI definitions in Java, it does not provide any code generation capabilities as the *AsyncAPI Toolkit* does.

Regarding code generation, the *AsyncAPI Initiative* now provides a generic template-based generator[32] which can be used to generate code in any language. As of writing this manuscript, only the following languages are officially supported by the *AsyncAPI generator*: HTML and Markdown for documentation; and Node.js, Java (relying on Spring), Python, and TypeScript NATS for executable code. In all cases, only a subset of the protocols considered by *AsyncAPI* are supported (although MQTT is generally—but not always—supported), and in some cases, some optional properties of the *AsyncAPI Specification* are marked as required for the code generators to work[33] thus breaking the compliance with the specification. In concrete the case of the *Java* code generator, the main difference with the *AsyncAPI Toolkit* is the lack of an internal DSL which helps developers to easily discover the data schema of the messages interchanged. Also, some importante features, such as parameters in topic names or server variables, are lacking or not fully supported. Finally, as opposed to our *AsyncAPI Toolkit*, which is implemented in Java for convenience purposes, the *AsyncAPI generator* is implemented in Node.js, which makes difficult to integrate it in a Java-based IDE. This is of special importance when the IDE also provides a plethora of MDE tools—as it is the case of Eclipse and EMF, which have become a *de facto* standard in the modeling community—which could help to build more complete and mature model-based tools around *AsyncAPI*. Thus, ours is the only approach targeting the modeling of *AsyncAPI* definitions providing an internal DSL in the form of a fluent API based on data models, and providing an IDE to develop the *AsyncAPI* definitions with full code generation capabilities in a single solution.

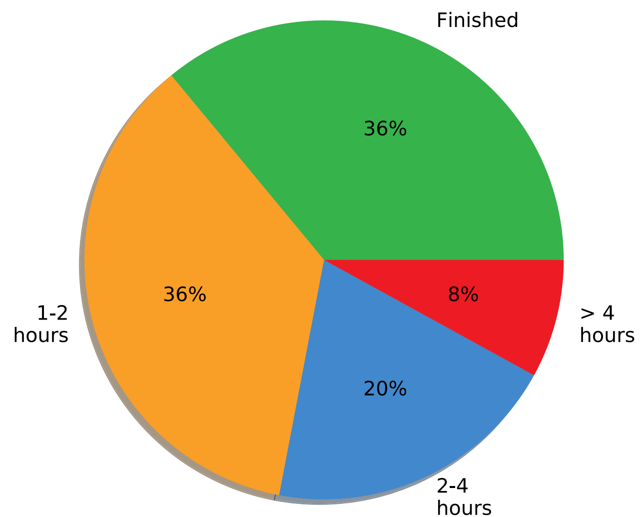

Fig. 12 Percentage of people finishing in 1h, 2h, 4h or more

8 Conclusions and further work

This article presents *AsyncAPI Toolkit*, a toolkit that allows specifying message-driven APIs using the *AsyncAPI* specification also providing automatic code generation. *AsyncAPI Toolkit* decreases the development time of architectures with asynchronous communication.

As described in the previous section, the investment in time to learn the basics of *AsyncAPI* and the *AsyncAPI Toolkit* is not too much. Even though few of the participants had previous knowledge of *AsyncAPI*, with a mean learning time of ≈28 minutes, 72% of the participants finished the exercise within the given hour or estimate that they would need less than a total of two hours to finish it as it is shown in Fig. 12. And the 36% of the participants were able to finish it within the fixed hour with only that mean learning time spent for initial knowledge acquisition.

Investing a bit more time in learning the basics of *AsyncAPI* should lead to better results in fewer time and effort. But for those developers who do not know the *AsyncAPI* specification in depth and do not have the need to deepen on it, the *AsyncAPI Toolkit* has proven to be a tool able to rapidly provide an interface to their systems.

Not only developers can get benefit from the *AsyncAPI Toolkit*, but also modelers can. The extensions discussed in

[30] https://www.asyncapi.com/docs/community/tooling.

[31] https://github.com/Pakisan/java-asyncapi.

[32] https://github.com/asyncapi/generator.

[33] E.g.., the `operationId` property must be always set to use the Node.js generators

Sects. 4.2, 5.3 and 5.4 are addressed to these users, who may expect this kind of functionality in any model-driven solution as they are used to it. In fact, after the first phase of the evaluation, most of the participants familiar with MDD requested or asked about a higher-level graphical editor. As a consequence, the *AsyncAPI Toolkit* allows starting a new API definition in two different ways: either creating a data model using an Ecore editor which generates an initial *AsyncAPI* definition, or directly creating the *AsyncAPI* definition textually.

Considering all participants suggestions and our own experience, these are the main lines of future work in our roadmap:

1. **Automatic test suite generation**: Having a new test generation functionality will facilitate the validation of the communication system's robustness. The goal would be to automatically generate different types of communication messages (both valid and invalid ones) considering the usual vulnerabilities that make not to comply the *single source of truth*. This mechanism will help to anticipate to possible problems and assure that the system will not be down due to those common issues.
2. **Cover other languages besides Java**: This has been highly requested. Our next milestone will include additional code generation templates to be able to automatically generate code according to the project's development language target: Python, C, C++, etc.
3. **User experience improvement**: Better error detection and identification during the specification editing, quick fix features to correct them, enhance *AsyncAPI* specification file editing with a tree view editor and include the requested editor assistant or wizards for the development of the model will all be part of our planned improvements in this line of work. Although they do not represent a new functionality for the *AsyncAPI Toolkit*, it makes a difference in user satisfaction and we think it will also improve the development time as nowadays error detection and fixing are not so intuitive for the user.

Finally, we will also evaluate *AsyncAPI Toolkit* further in to different aspects. On the one hand, we plan to perform additional empirical validations that help us better understand the specific trade-offs of introducing our model-driven *AsyncAPI* infrastructure in new industrial projects, when we get a larger dataset for this analysis and explore the option to employ our toolkit in reverse engineering scenarios, e.g., to generate visualizations of existing *AsyncAPI* specifications with our graphical syntax. On the other hand, we also plan to compare *AsyncAPI Toolkit* against other tools that undoubtedly will be developed based on *AsyncAPI*, following a systematic review approach such as the one described in [9].

Acknowledgements This project has received funding from the Electronic Component Systems for European Leadership Joint Undertaking under grant agreement N° 737494, this Joint Undertaking receives support from the European Union's Horizon 2020 research and innovation program and from Sweden, France, Spain, Italy, Finland & Czech Republic; from the European commission under the COMP4DRONES project under grant agreement N° 826610 with support from the ECSEL-JU 2018 program with national financing from France, Spain, Italy, Netherlands, Austria, Czech, Belgium and Latvia; and from the Spanish government under project *Open Data for All* (RETOS TIN2016-75944-R). We also want to acknowledge all participants for their collaboration in the experiment.

Funding Open Access funding provided thanks to the CRUE-CSIC agreement with Springer Nature.

Open Access This article is licensed under a Creative Commons Attribution 4.0 International License, which permits use, sharing, adaptation, distribution and reproduction in any medium or format, as long as you give appropriate credit to the original author(s) and the source, provide a link to the Creative Commons licence, and indicate if changes were made. The images or other third party material in this article are included in the article's Creative Commons licence, unless indicated otherwise in a credit line to the material. If material is not included in the article's Creative Commons licence and your intended use is not permitted by statutory regulation or exceeds the permitted use, you will need to obtain permission directly from the copyright holder. To view a copy of this licence, visit http://creativecommons.org/licenses/by/4.0/.

References

1. Al-Fuqaha, A., Guizani, M., Mohammadi, M., Aledhari, M., Ayyash, M.: Internet of things: a survey on enabling technologies, protocols, and applications. IEEE Commun. Surv. Tutor. **17**(4), 2347–2376 (2015)
2. Alaerjan, A., Kim, D.K., Ming, H., Malik, K.: (2018) Using DDS based on unified data model to improve interoperability of smart grids. In: 2018 6th IEEE International Conference on Smart Energy Grid Engineering, SEGE 2018, pp 110–114, https://doi.org/10.1109/SEGE.2018.8499513
3. APIs.guru API tooling for better developer experience. last accessed September (2020)https://apis.guru/ (2020)
4. Artikov, M., Meier, J., Winter, A. :Towards integrated iot-languages. In: 2019 International Conference on Information Science and Communications Technologies (ICISCT), pp 1–5, (2019) https://doi.org/10.1109/ICISCT47635.2019.9011964
5. AsyncAPI Initiative AsyncAPI specification 2.0.0. URL: https://www.asyncapi.com/docs/specifications/2.0.0/, last accessed May 2021 (2020)
6. Berrouyne, I., Adda, M., Mottu, J., Royer, J., Tisi, M.: Cypriot: framework for modelling and controlling network-based iot applications. In: Hung C, Papadopoulos GA (eds) Proceedings of the 34th ACM/SIGAPP Symposium on Applied Computing, SAC 2019, Limassol, Cyprus, April 8-12, 2019, ACM, pp 832–841, (2019)https://doi.org/10.1145/3297280.3297362
7. Cánovas-Izquierdo, J.L., Cabot, J.: Jsondiscoverer: visualizing the schema lurking behind JSON documents. Knowl. Based Syst. **103**, 52–55 (2016). https://doi.org/10.1016/j.knosys.2016.03.020
8. Capilla, R., Bosch, J., Trinidad, P., Ruiz-Cortés, A., Hinchey, M.: An overview of dynamic software product line architectures and techniques: observations from research and industry. J. Syst. Softw. **91**, 3–23 (2014)

9. Challenger, M., Kardas, G., Tekinerdogan, B.: A systematic approach to evaluating domain-specific modeling language environments for multi-agent systems. Softw. Qual. J. **24**(3), 755–795 (2016). https://doi.org/10.1007/s11219-015-9291-5
10. Ciccozzi, F., Spalazzese, R.: MDE4IoT: Supporting the internet of things with model-driven engineering. In: Badica, C., El Fallah, Seghrouchni A., Beynier, A., Camacho, D., Herpson, C., Hindriks, K., Novais, P. (eds.) Intelligent Distributed Computing X, pp. 67–76. Springer International Publishing, Cham (2017)
11. Clark, T., Barn, B.S.: A common basis for modelling service-oriented and event-driven architecture. In: Proceedings of the 5th India Software Engineering Conference, Association for Computing Machinery, New York, NY, USA, ISEC '12, p 23–32, (2012) https://doi.org/10.1145/2134254.2134258
12. Das, T.K., Dingel, J.: Model development guidelines for UML-RT: conventions, patterns and antipatterns. Softw. Syst. Model. **17**(3), 717–752 (2018). https://doi.org/10.1007/s10270-016-0549-6
13. Delicato, F.C., Pires, P.F., Batista, T.: Middleware Solutions for the Internet of Things. Springer Publishing Company Incorporated, London (2013)
14. Díaz, O., Villoria, F.M.: Generating blogs out of product catalogues: An MDE approach. J. Syst. Softw. **83**(10), 1970–1982 (2010). https://doi.org/10.1016/j.jss.2010.05.075
15. Ed-Douibi, H., Cánovas-Izquierdo, J.L., Cabot, J.: Example-driven web API specification discovery. In: Modelling Foundations and Applications–13th European Conference, ECMFA@STAF 2017, Marburg, Germany, July 19-20, 2017, Proceedings, pp 267–284, (2017)https://doi.org/10.1007/978-3-319-61482-3_16
16. Ed-Douibi, ,H., Cánovas-Izquierdo, J.L., Cabot, J.: Openapitouml: A tool to generate UML models from openapi definitions. In: Proceedings of Web Engineering–18th International Conference, ICWE 2018, Cáceres, Spain, June 5-8, 2018, pp 487–491, (2018). https://doi.org/10.1007/978-3-319-91662-0_41
17. Ed-Douibi, H., Izquierdo, J.L.C., Cabot, J.: Automatic generation of test cases for REST apis: A specification-based approach. In: 22nd IEEE International Enterprise Distributed Object Computing Conference, EDOC 2018, Stockholm, Sweden, October 16-19, 2018, pp 181–190,(2018) https://doi.org/10.1109/EDOC.2018.00031
18. Evans, D.: The internet of things: How the next evolution of the internet is changing everything. CISCO white paper **1**(2011), 1–11 (2011)
19. Fielding, R.T.: REST: architectural styles and the design of network-based software architectures. Doctoral dissertation, University of California, Irvine (2000)
20. Fowler, M.: Domain Specific Languages, 1st edn. Addison-Wesley Professional, Boston (2010)
21. Gómez, A., Iglesias-Urkia, M., Urbieta, A., Cabot, J.: A model-based approach for developing event-driven architectures with asyncapi. In: Proceedings of the 23rd ACM/IEEE International Conference on Model Driven Engineering Languages and Systems, Association for Computing Machinery, New York, NY, USA, MODELS '20, pp. 121–131, https://doi.org/10.1145/3365438.3410948 (2020)
22. Iglesias, A., Iglesias-Urkia, M., López-Davalillo, B., Charramendieta, S., Urbieta, A.: Trilateral: Software product line based multidomain iot artifact generation for industrial cps. In: Proceedings of the 7th International Conference on Model-Driven Engineering and Software Development - Volume 1: MODELSWARD, INSTICC, SciTePress, pp 64–73, https://doi.org/10.5220/0007343500640073 (2019)
23. Iglesias-Urkia, M., Gómez, A., Casado-Mansilla, D., Urbieta, A.: Enabling easy web of things compatible device generation using a model-driven engineering approach. In: ACM International Conference Proceeding Series, (2019). https://doi.org/10.1145/3365871.3365898
24. Iglesias-Urkia, M., Gómez, A., Casado-Mansilla, D., Urbieta, A.: Automatic generation of web of things servients using thing descriptions. Pers. Ubiquit. Comput. (2020). https://doi.org/10.1007/s00779-020-01413-3
25. Iglesias-Urkia, M., Iglesias, A., López-Davalillo, B., Charramendieta, S., Casado-Mansilla, D., Sagardui, G., Urbieta, A.: Trilateral: A model-based approach for industrial cps–monitoring and control. In: Hammoudi, S., Pires, L.F., Selić, B. (eds.) Model-Driven Engineering and Software Development, pp. 376–398. Springer International Publishing, Cham (2020)
26. Ivanchikj, A., Pautasso, C.: Modeling microservice conversations with restalk. In: Bucchiarone A, Dragoni N, Dustdar S, Lago P, Mazzara M, Rivera V, Sadovykh A (eds) Microservices: Science and Engineering, Springer International Publishing, Cham, pp 129–146, (2020). https://doi.org/10.1007/978-3-030-31646-4_6
27. Jazdi, N.: Cyber physical systems in the context of industry 4.0. In: 2014 IEEE International Conference on Automation, Quality and Testing, Robotics, pp 1–4, (2014). https://doi.org/10.1109/AQTR.2014.6857843
28. Kagermann, H., Helbig, J., Hellinger, A., Wahlster, W.: Recommendations for implementing the strategic initiative INDUSTRIE 4.0: Securing the future of German manufacturing industry; final report of the Industrie 4.0 Working Group. Forschungsunion (2013)
29. Klikovits, S., Linard, A., Buchs, D.: Crest - a dsl for reactive cyber-physical systems. In: Khendek F, Gotzhein R (eds) System Analysis and Modeling. Languages, Methods, and Tools for Systems Engineering, Springer International Publishing, Cham, pp 29–45 (2018)
30. Lee, B., Kim, D.K., Yang, H., Oh, S.: Model transformation between OPC UA and UML. Comput. Stand. Interface. **50**, 236–250 (2017). https://doi.org/10.1016/j.csi.2016.09.004
31. Leitão, P., Colombo, A.W., Karnouskos, S.: Industrial automation based on cyber-physical systems technologies: Prototype implementations and challenges. Comput. Ind. **81**, 11–25 (2016)
32. Leitner, S.H., Mahnke, W.: Opc ua-service-oriented architecture for industrial applications. ABB Corpor. Res. Center **48**, 61–66 (2006)
33. Michelson, B.M.: Event-driven architecture overview. Patricia Seybold Group **2**(12), 10–1571 (2006)
34. Morin, B., Harrand, N., Fleurey, F.: Model-based software engineering to tame the iot jungle. IEEE Softw. **34**(1), 30–36 (2017). https://doi.org/10.1109/MS.2017.11
35. Negash, B., Westerlund, T., Rahmani, A.M., Liljeberg, P., Tenhunen, H.: Dos-il: A domain specific internet of things language for resource constrained devices. In: Procedia Computer Science, Elsevier B.V., vol 109, pp. 416–423, (2017). https://doi.org/10.1016/j.procs.2017.05.411
36. Nepomuceno, T., Carneiro, T., Maia, P.H., Adnan, M., Nepomuceno, T., Martin, A.: (2020) Autoiot: A framework based on user-driven mde for generating iot applications. In: Proceedings of the 35th Annual ACM Symposium on Applied Computing, ACM, SAC '20, p 719–728, 10.1145/3341105.3373873
37. OMG (2015) Data Distribution Service (DDS), Ver. 1.4. https://www.omg.org/spec/DDS/1.4/
38. OMG (2017) OMG Unified Modeling Language (OMG UML), Ver. 2.5.1. https://www.omg.org/spec/UML/2.5.1/
39. OMG (2018) Interface Definition Language (IDL), Ver. 4.2. https://www.omg.org/spec/IDL/4.2/
40. OMG (2019) Meta Object Facility (MOF), Ver. 2.5.1. http://www.omg.org/spec/MOF/2.5.1/
41. OpenAPI Initiative (2020) OpenAPI Specification. URL: https://github.com/OAI/OpenAPI-Specification, last accessed May 2021
42. Pauker, F., Frühwirth, T., Kittl, B., Kastner, W.: A Systematic Approach to OPC UA Information Model Design. Procedia CIRP **57**, 321–326 (2016). https://doi.org/10.1016/j.procir.2016.11.056, factories of the Future in the digital environment - Proceedings of the 49th CIRP Conference on Manufacturing Systems

43. Pauker, F., Wolny, S., Fallah, S.M., Wimmer, M.: UML2OPC-UA - Transforming UML Class Diagrams to OPC UA Information Models. Procedia CIRP **67**, 128–133 (2018). https://doi.org/10.1016/j.procir.2017.12.188, 11th CIRP Conference on Intelligent Computation in Manufacturing Engineering, 19-21 July 2017, Gulf of Naples, Italy
44. Riedel, T., Fantana, N., Genaid, A., Yordanov, D., Schmidtke, H.R., Beigl, M.: Using web service gateways and code generation for sustainable iot system development. In: 2010 Internet of Things (IOT), Tokyo, Japan, pp 1–8, 10.1109/IOT.2010.5678449 (2010)
45. Schwab K (2017) The Fourth Industrial Revolution. Crown Publishing Group
46. Sharaf, M., Abusair, M., Eleiwi, R., Shana'a, Y., Saleh, I., Muccini, H.: Modeling and code generation framework for iot. In: i Casas PF, Sancho M, Sherratt E (eds) System Analysis and Modeling. Languages, Methods, and Tools for Industry 4.0 - 11th International Conference, SAM 2019, Munich, Germany, September 16-17, 2019, Proceedings, Springer, Lecture Notes in Computer Science, vol 11753, pp 99–115, https://doi.org/10.1007/978-3-030-30690-8_6 (2019)
47. SmartBear Software (2020) What Is OpenAPI? https://swagger.io/docs/specification/about/
48. Sneps-Sneppe, M., Namiot, D.:On web-based domain-specific language for internet of things. In: International Congress on Ultra Modern Telecommunications and Control Systems and Workshops, IEEE Computer Society, vol 2016-January, pp 287–292, https://doi.org/10.1109/ICUMT.2015.7382444 (2016)
49. Steinberg, D., Budinsky, F., Paternostro, M., Merks, E.: EMF: Eclipse Modeling Framework 2.0, 2nd edn. Addison-Wesley Professional, ISBN: 0321331885 (2009)
50. Tashtoush, Y., AlRashdan, M.N., Salameh, O., Alsmirat, M.: Swagger-based jquery ajax validation. In: 2019 IEEE 9th Annual Computing and Communication Workshop and Conference (CCWC), pp 0069–0072, https://doi.org/10.1109/CCWC.2019.8666542 (2019)
51. The Eclipse Foundation (2020) Eclipse Modeling Project - Eclipse Modeling Framework - Home. http://www.eclipse.org/emf/, Accessed May 2021
52. The Eclipse Foundation (2020) Xtend - Modernized Java. http://www.eclipse.org/xtend/. Accessed May 2021
53. The Eclipse Foundation (2020) Xtext - Language Engineering Made Easy! http://www.eclipse.org/Xtext/. Accessed May 2021
54. Thramboulidis, K., Christoulakis, F.: UML4IoT - A UML-based approach to exploit IoT in cyber-physical manufacturing systems. Comput. Ind. **82**, 259–272 (2016). https://doi.org/10.1016/j.compind.2016.05.010
55. W3C (2019) Web of Things at W3C. https://www.w3.org/WoT/
56. Wright, H.K., Kim, M., Perry, D.E.: Validity concerns in software engineering research. In: Proceedings of the FSE/SDP Workshop on Future of Software Engineering Research, Association for Computing Machinery, New York, FoSER '10, p 411–414, (2010). https://doi.org/10.1145/1882362.1882446
57. Young, B., Cheatwood, J., Peterson, T., Flores, R., Clements, P.C.: Product line engineering meets model based engineering in the defense and automotive industries. In: Proceedings of the 21st International Systems and Software Product Line Conference, SPLC 2017, Volume A, pp 175–179 (2017)

Publisher's Note Springer Nature remains neutral with regard to jurisdictional claims in published maps and institutional affiliations.

Abel Gómez is a senior researcher of the *Internet Interdisciplinary Institute*, a research center of the *Universitat Oberta de Catalunya*, Spain. Previously, he has hold different positions at the *Universidad de Zaragoza*, the *École des Mines de Nantes & Inria*, and the *Universitat Politécnica de València*; being this latter institution where he obtained his PhD degree in Computer Science. His research interests fall in the broad field of Model-Driven Engineering (MDE), and his research lines have evolved in two complementary directions: on the one hand, the development of core technologies to support MDE activities; and on the other hand, the application of MDE techniques to solve Software Engineering problems. More information is available at https://abel.gomez.llana.me.

Markel Iglesias-Urkia is a researcher at Ikerlan on the IoT and Digital Platforms team since 2019, after completing his PhD on Engineering for the Information Society and Sustainable Development from the University of Deusto in the same team. Previously, he received his BSc on Computer Systems Engineering, BSc Computer Science & Engineering and MSc on Embedded System Engineering from the University of the Basque Country. His research interests include Internet of Things (IoT), CoAP, lightweight protocols, Edge Computing among others. He is author or co-author of several peer-reviewed scientific publications in the field of the Internet of Things and Software Engineering.

Lorea Belategi joined IKERLAN as a researcher in Dependable Software Team. After completing her PhD at Mondragon Unibertsitatea, she worked in diverse sectors (petrochemical, energy, health, etc.). After several years she became responsible for software quality at Kiro Grifols where she coordinated and carried out the tasks of verification and validation of medical devices software to be certified according to FDA (Food and Drug Administration). Later, she moved back to research, at IKERLAN. Her research interests include software V&V, Edge Computing, Model-Driven Engineering (MDE), embedded systems. She is author or co-author of several peer-reviewed scientific publications in the field of model based software V&V.

Xabier Mendialdua is a researcher of the *Dependable Software Team* at *Ikerlan S. Coop.* Since 1994 he has been involved in multiple projects in several domains as automated warehouse management systems, monitoring systems, wind power and railway. His main activity is focused in the design and development of dependable software for embedded control systems and his research activities have been involved on applying model-driven and product-line approaches for the engineering of control systems.

Jordi Cabot received the B.Sc. and Ph.D. degrees in computer science from the Technical University of Catalonia. He was a Leader of an INRIA and LINA Research Group at École des Mines de Nantes, France, a Post-Doctoral Fellow with the University of Toronto, a Senior Lecturer with the Open University of Catalonia, and a Visiting Scholar with the Politecnico di Milano. He is currently an ICREA Research Professor at Internet Interdisciplinary Institute. His research interests include software and systems modeling, formal verification and the role AI can play in software development (and vice versa). He has published over 150 peer-reviewed conference and journal papers on these topics. Apart from his scientific publications, he writes and blogs about all these topics in several sites. He is a member of the IEEE and the ACM.