ThingML: A Language and Code Generation Framework for Heterogeneous Targets

Nicolas Harrand,
Franck Fleurey and
Brice Morin
SINTEF ICT
Oslo, Norway
first.last@sintef.no

Knut Eilif Husa
TellU AS
Asker, Norway
knuf.eilif.husa@tellu.no

ABSTRACT

One of the selling points of Model-Driven Software Engineering (MDSE) is the increase in productivity offered by automatically generating code from models. However, the practical adoption of code generation remains relatively slow and limited to niche applications. Tooling issues are often pointed out but more fundamentally, experience shows that: (i) models and modeling languages used for other purposes are not necessarily well suited for code generation and (ii) code generators are often seen as black-boxes which are not easy to trust and produce sub-optimal code. This paper presents and discusses our experiences applying the ThingML approach to different domains. ThingML includes a modeling language and tool designed for supporting code generation and a highly customizable multi-platform code generation framework. The approach is implemented in an open-source tool providing a family of code generators targeting heterogeneous platforms. It has been evaluated through several case studies and is being used for in the development of a commercial ambient assisted living system.

Keywords

MDE, DSML, Code generation, Heterogeneous, Customization

1. INTRODUCTION

Modern systems are complex assemblies of proprietary and open-source components, frameworks and services which are re-used, evolved, customized and composed to create and evolve applications. To cope with this evolution, the industry has adopted techniques such as agile methods, DevOps, continuous testing and virtualization. As pointed out in [19] and [3], to make a difference MDE has to propose solutions which fit in this reality, focus on specific problems and fit in the software development process as a whole.

In this paper, we present the ThingML approach developed as the result of a set of collaborations with industry partners and in an effort to bring the benefits of MDE in a practical setup usable by a wide range of developers. The ThingML approach focuses on the customizability of its code generators while providing the abstraction that developers need to improve productivity [15]. The approach does not aim at replacing programming or hiding source code but instead at helping developer produce better source code more efficiently. As noted by France in [8] programming and modeling are complementary activities.

The contribution of the paper is two-fold. First it details our experience applying MDE in practice and discusses the lessons learned over the three main iterations of the ThingML tool. Second it describes the resulting approach and more specifically discuses the code generation framework build around the ThingML language in order to allow developers to easily customize the code generators for the need of their specific target platforms and projects.

The ThingML code generation framework has been used to generate code in 3 different languages (C/C++, Java and Javascript), targeting around 10 different target platforms (ranging from tiny 8bit microcontrollers to servers) and 10 different communication protocols. In addition, the ThingML approach is currently being used by the Norwegian company Tellu for the development of a new range of eHealth and fall detection systems called Safe@Home, to be deployed in elderly homes.

The remainder of this paper is organized as follows. Section 2 details how the ThingML approach was iteratively refined and discusses lessons learned from each iteration. Section 3 presents the resulting ThingML approach, the code generation framework and discusses how it has been specialized to support a wide set of platforms. Section 4 discusses the communication extension points and plugins of the framework. Section 5 gives on overview of 3 systems developed with ThingML: the Tellu Safe@Home system, a micro-aerial vehicle platform and an Arduino based IoT framework. Finally, Section 6 discusses related work and Section 7 concludes by summarizing the key lessons learned.

2. CONTEXT AND BACKGROUND

The ThingML language and tool are the results of our experiences in applying MDE in practice to a range of different contexts during the last 5 years. The systems developed with ThingML include different types of projects ranging from case studies in research projects to product development in industry projects. In terms of the application domains, the

MODELS '16, October 02-07, 2016, Saint-Malo, France

DOI: http://dx.doi.org/10.1145/2976767.2976812

first use cases for ThingML were embedded systems (for example sensor networks in the oil and gas domain) but it has more generally been applied to Internet of Things (IoT) from which it got its name, Cyber-Physical Systems (CPS), home and building automation, medical devices, robotics, etc.

The ThingML approach is targeted at distributed reactive systems and is especially beneficial for applications which include heterogeneous platforms (which can range from tiny microcrontollers to cloud servers) and heterogeneous communication channels. To that extent, and if compared to the UML, ThingML can be considered as a domain specific modeling language (DSML). It is however not specific to any business domain which allows its use for a wide range of applications.

While models and MDE can be beneficial for many other purposes (*e.g.* requirements, testing) and actors in the software development process, the goal of ThingML is to bring MDE to the late design and the implementation phases of the software life cycle as well as to support maintenance and evolution tasks. The purpose of ThingML is to allow modeling software components and to automatically generate complete modules of code ready to be deployed. The target user group for ThingML is thus a broad range of software developers and architects.

2.1 Initial work with UML

The initial work leading to ThingML was aiming at applying MDE for the development of embedded software together with industrial partners. The embedded system development teams we worked with were composed of highly skilled developers. They follow processes which had been refined over the years and were able to deliver systems with tight reliability and performance constraints. In terms of modeling, these teams did not use UML or any UML tools but typically did use state machines in order to design, discuss and document the high level functionality of the different components of their systems. One team had a practice (and a textual notation) to systematically formalize those state machines in a wiki used by the development team. These models would be typically later transformed to code in a systematic but fully manual way, even though a number of open-source and commercial tools also existed to generate code from state machines.

Lessons learned

From the state of the art, UML together with profiles such as MARTE and SYSML appeared as natural choices to introduce MDE. Reusing standards has a number of benefits: UML is typically already used (to some extent) in design phases and/or for documentation purposes, tools are available, etc.

When it comes to code generation and implementation, many commercial UML tools include code generators.

However, while proposing a method based on UML was encouraged by decision makers and software designers, its use by the embedded systems development teams proved very impractical and counter-productive. Developers had no prior knowledge of UML and the UML modeler was difficult to use, only a very small sub-set of UML/SYSML/MARTE was relevant for the task, and the code generators were only able to generate incomplete code from statecharts (either without any action code or with action code written in text boxes). After a short trial, an approach based on a generic UML tool was rejected by practitioners.

2.2 Version 1

After the initial experiment using UML, a first version of the approach was proposed based on non-UML state machine tools dedicated to embedded system development. The approach was built as a three layer architecture: a domain layer, an application layer and a hardware and driver layer. The domain layer specifies a set of domain components (which could be both hardware or software) as well as their interfaces in terms of asynchronous messages. A simple textual DSL allowed for the definition of the domain model and a code generator was used to systematically generate APIs from the domain model and communication code [7]. The benefit of this explicit domain model was to ensure a decoupling between the application part which remained platform independent and the hardware/driver part which is by nature highly platform dependent.

The application layer was fully modeled using a commercial state machine tools (IAR VisualSTATE) and could only manipulate the components and messages defined in the domain model. The IAR VisualSTATE code generator was used to generate a complete module implementing the application which could be linked to the domain model API. In the state machine, the variables and actions were written directly in C. The driver and hardware layer consisted of manually implementing the APIs defined in the domain model. The approach ensured that each component was implemented in a separate module.

Lessons learned

The benefit of this first version was a good decoupling between the developers of the different drivers and the developers of an application. The use of a commercial modeling tool dedicated to state machines and embedded systems proved a lot more practical than the tailoring of a generic UML tool. The textual DSL used for the definition of the domain model also proved easy to use by the developers.

Some of the drawbacks included the difficulty of graphically specifying complex state machines and writing actions as C code in text boxes, the loose connection between the domain model and the state machine tool and the fully manual implementation of the driver/hardware layer.

2.3 Version 2

The second version of the ThingML tool was built as a single textual domain specific modeling language. It keeps the principles of the previous approach: the mandatory definition a component with explicit asynchronous message interfaces and the possibility to define both platform independent components and platform specific components independently. However, it does not rely anymore on any external tool or code generator. The component model previously defined is extended with a state machine model to specify the behavior of the components and an action language defines platform independent guards and actions in the state machines [5]. For platform specific code, a kick-down mechanism allows embedding platform code in the model. Finally, a configuration model also allows assembling components in order to generate code for complete assemblies of components.

This second version of ThingML was applied to a set of

use cases and a set of code generators has been developed and specialized to accommodate for different target platforms and programming languages. Those target platforms included C for microcontrollers, C for POSIX/Linux, and the JVM. Experience showed that the textual notation for the state machine was beneficial for editing and implementing the components behavior. The benefit of the textual syntax are its usability for developers, ability to handle large state machines, ability to support storage and versioning in any code repository, etc. The only drawback is that it does not allow to easily get an overview of the state machines. To provide this graphical overview, a transformation is provided in order to export graphical UML representation of the ThingML state machines.

Lessons learned

The experience from using the second version of ThingML showed a good usability of the proposed modeling languages (mostly based on UML) and the ability to generate code suitable for a wide range of target platforms (from microcontrollers to servers).

However, one limitation was that for each different target platform and even for each different project or application, the code generators had to be tailored and customized for the specific need in order to generate satisfactory code. While the code generator was implemented in Scala using a set of templates and helpers, customizing it often required a good understanding of its inner working, required a significant effort and often resulted in duplicating large parts of the code generator in order to make minor changes to the generated code.

3. THINGML APPROACH (VERSION 3)

The third and current version of the ThingML approach keeps the language unchanged but includes a complete rewrite of the code generator as a plain Java Object-Oriented framework designed to be easily customized by the developers for their individual organization and project needs. The following sections detail the design and usage of this version of ThingML.

The code generator is actually a family of compilers[1], each targeting a specific language (currently C, Java and JavaScript). Considerable effort has been made to identify and expose extension points (that will be detailed in Section 3.2) that enable deep customization providing adaptability to various platforms.

3.1 ThingML DSL

The ThingML language relies on two key structures: Things that represents software components, and Configurations that describe their interconnection. A Thing is an implementation unit, also referred to as component or process in other approaches. A Thing can define properties, functions, messages, ports and a set of state machines. The properties are variables which are local to a thing and can be accessed globally from within a thing (*e.g.* from a function or in a state machine). The functions are local to a thing. They can be used from anywhere in a thing but are not visible (private) from other things. The only public interface of a Thing is through the ports, which can send and receive a set of messages. The messages (and their signatures) are defined within a thing but can only be sent and received through ports.

The internal behavior of a thing, is implemented using a mix of:

1. Imperative programming to express simple procedures, either directly using the ThingML platform-independent action and expression language, or using the features of the target language (C, Java, JS, etc.) and wrapping existing libraries, or a combination of both options.
2. Event-Condition-Action (ECA) to express simple if-then rules to react on events in a stateless fashion
3. Composite State machine, conforming to UML statecharts to react on and orchestrate events in a stateful fashion

Ultimately, a ThingML compiler transforms a configuration into fully operational code in a targeted language, ready to be built and run. A configuration is a set of instances of previously defined things, and a set of connectors between two ports from those instances. By default, ThingML connectors are local connectors, which only allows for the communication between instances deployed on the same physical node. However, in a realistic setup, the logic needs to be distributed across several nodes, and inter-node communication is thus required. Therefore, we have added the possibility to connect a port with the outside of the configuration (see Section 4).

3.2 ThingML Code Generation Framework

The ThingML code generation framework defines a family of compilers able to transform a ThingML model into fully operational code in various languages. So far C, C++, Java and JavaScript are supported as target language. Each compiler is composed of a set of code generators, each of them responsible for the compilation of a specific sub-set of ThingML.

This modular structure allows for the customization of some extension points, while all the others can be reused as-is. The figure 1 presents the 10 different extension points we have identified.

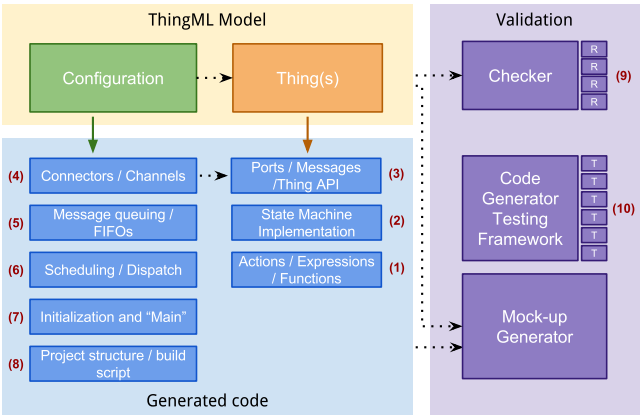

Figure 1: ThingML framework extension points

These extension points are separated into two groups: the ones corresponding to the generation of code for "Things" and the ones corresponding to the generation of code for

[1]or, more precisely, model-to-text transformations

Configurations (or applications). In the metamodel, the coupling between these two things is through the instances of Things which are contained in configurations. In the generated code, the idea is to also keep a separation between the reusable code which is generated for Things (types) and the code generated to combine instances of Things together into an application. Each extension point is basically an interface (or abstract class) in the code generation framework with a set of methods responsible for generating the code associated to a given model element. Note that extension points can overlap, in the sense that they can potentially read the same model elements. Model elements cannot be modified (read-only) during the code generation process, but different extension points can share information through a common context. This way, different extension points can agree on naming conventions, etc. For example, an action responsible for setting the value of a property needs to know how the property is actually named in the generated code (generated by another extension point).

1) Actions / Expressions / Functions: This part of the code generator corresponds to the code generated for actions, expressions and functions contained in a Thing. The generated code mostly depends on the language supported by the target platform (C, Java, etc.), and the code generators should be quite reusable across different platforms supporting the same language. The implementation of this extension point consists of a visitor on the Actions and Expressions part of the metamodel. New code generators can be created by inheriting from that abstract visitor and implementing all its methods. Alternatively, if only a minor modification of an existing code generator is needed, it is possible to inherit from the existing visitor and only override a subset of its methods.

2) Behavior implementation: This part of the code generator corresponds to the code generated from the state machine structures and ECA contained in Things. There are main strategies and frameworks available in the literature in order to implement state machines. Depending on the capabilities, languages and libraries available on the target platform, the platform expert should have the flexibility of specifying how the behavior is mapped to executable code. In some cases, the code generator can produce the entire code for the state machines, for example using a state machine design pattern in C++ or Java, and in other cases the code generator might rely on an existing framework available on the target platform, such as state.js for executing JavaScript state machines. To allow for this flexibility, the ThingML framework should provide a set of helpers to traverse the different metaclasses responsible for modeling the behavior and leave the freedom of creating new concrete generators and/or customizing existing code generator templates.

3) Ports / Messages / Thing APIs: This part of the code generator corresponds to the wrapping of "things" into reusable components on the target platform. Depending on the target platform, the language and the context in which the application is deployed, the code generated for a "thing" can be tailored to generate either custom modules or to fit particular coding constraints or middleware to be used on the target platform. At this level, a Thing is a black box which should offer an API to send and receive messages through its ports. In practice, this should be customized by the platform experts in order to fit the best practices and frameworks available on the target platform. As a best practice, the generated modules and APIs for things should be manually usable in case the rest of the system (or part of it) is written directly in the target language. For example, in object oriented languages, a facade and the observer pattern can be used to provide an easy to use API for the generated code. In C, a module with the proper header with structures and call-backs should be generated.

4) Connectors / Channels: This part of the code generator is in charge of generating the code corresponding to the connectors and transporting messages from one Thing to the next. This is the client side of the APIs generated for the Things. In practice, the connector can connect two things running in the same process on a single platform or things which are remotely connected through some sort of network. This specific point is described with more details in Section 4.

5) Message Queuing / FIFOs: This part of the generator is related to the connectors and channels but is specifically used to tailor how messages are handled when the connectors are linking two things running on the same platform. When the connectors are between things separated by a network or some sort of inter-process communication, the asynchronous nature of messages is ensured by construction. However, inside a single process specific additional code should be generated in order to store messages in FIFOs and dispatch them asynchronously. Depending on the target platform, the platform expert might reuse existing message queues provided by the operating system or a specific framework. If no message queuing service is available, like on the Arduino platform for example, the code for the queues can be fully generated.

6) Scheduling / Dispatch: This part of the code generator is in charge of generating the code which orchestrates the set of Things running on one platform. The generated code should activate successively the state machines of each component and handle the dispatch of messages between the components using the channels and message queues. Depending on the target platform, the scheduling can be based on the use of operating system services, threads, an active object design pattern or any other suitable strategy.

7) Initialization and "Main": This part of the code generator is in charge of generating the entry point and initialization code in order to set up and start the generated application on the target platform. The ThingML framework provides some helpers to list the instances to be created, the connections to be made and the set of variables to be initialized together with their initial values.

8) Project structure / build script: This extension point is not generating code as such, but the required file structure and builds scripts in order to make the generated code well packaged and easy to compile and deploy on the target platform. The ThingML framework provides access to all the buffers in which the code has been generated and allows creating the file structure which fits the particular target platform. For example, the Arduino compiler concatenates all the generated code into a single file which can be opened by the Arduino IDE. The Linux C code generator creates separate C modules with header files and generates a Makefile to compile the application. The Java code generators create Maven project and pom.xml files in order to allow compiling and deploying the generated code. The platform expert can customize the project structure and build scripts in order to fit the best practices of the target platform.

9) Checker: This part of the framework provides support for validating an input ThingML model before generating code from it. It not only includes syntactical checks, but also a customizable set of rules that apply to the application logic expressed in a model. While most of this rules are generic and apply to all compilers regardless of the targeted language, some are specific to a certain platform or language. Checking the deterministic nature of transition in a state machine can be useful disregarding the target language whereas verifying that no float type is defined on 64 bits on a platform that does not support it only makes sense on this specific platform. This system of rules can easily be extended (by adding rules) to enrich existing compilers, or to support validation for a new one. It is the role of the checker to collect and enforce rules added by every module of the compiler in use.

10) Code generator testing framework: In order to validate code generators, the code generator testing framework provides an extensible set of tests, with currently 140 tests. Those are ThingML models of programs from which expected outputs (oracle) are provided. As much as possible those model demonstrate one specific aspect of the ThingML language. Therefore, the framework is able to test a compiler, and check that these models are indeed transformed into code that produces the expected outputs. By comparing the outputs of different compilers, it is also possible to detect possible misalignement in the semantics implemented by these different compilers. The testing framework can be simply extended by adding new test models.

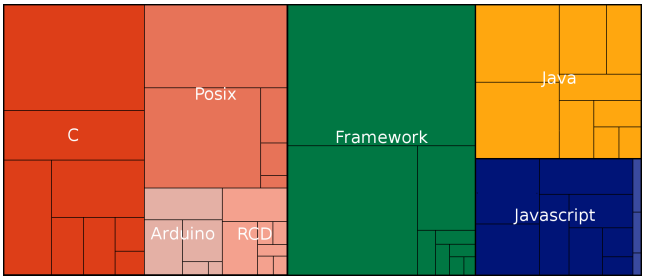

Figure 2: ThingML compilers SLoC distribution

The ThingML code generation framework is implemented as an Object-Oriented Java framework. Figure 2 shows the distribution of LoC for the different ThingML compilers. It reveals that the effort required to develop a compiler targeting a dialect from a supported language is significantly smaller than developing a new one from scratch.

For example, the Linux C and Arduino code generators actually share 95% of their code, meaning that the Linux and Arduino experts can really focus on the 5% differences between those two platforms. They mainly differ on the "Main" generator, and the "Build" generator, as the structure of the generated code expected differs on this points. (Note that one of the biggest source of line for the POSIX is network plugins.)

Similar gains have been obtained between plan Node.JS (JavaScript) and Espruino (The block corresponding to the slocs of Espruino is to the right of the one representing JavaScript). It is also noticeable than compiler targeting language of higher level (java, JavaScript) are implemented using fewer slocs.

4. COMMUNICATIONS

As mentioned earlier, an important extension point is the customization of the code generation for connectors. The goal of this extension point is to enable the seamless exchange of messages with software components running on a remote device, implemented or not in ThingML. In the end, communication is transparent for the developer, who can send a message just by calling `port!message(parameters)` and react to incoming messages using `event port?message do ... end`, as in a plain ThingML program. The following section aggregate the different conclusion drawn during our case studies (detailed in Section 4.2 and 4.3).

Typically, a ThingML component can be required to communicate with three type of components:

ThingML system: In some situations, a developer can model application with ThingML for both ends of the communication. In this case, communication stacks can be fully generated. The code generation framework must be extensible in order to flexibility on the choice of the transport protocol.

Open systems: In some other case, a ThingML component needs to communicate with another software component, whose sources are accessible to the developer. It is either possible to adapt the ThingML end, or to adapt the other one by generating code in its language. In addition to the support of a specific protocol, the code generation framework must also be adaptable in terms of message encoding. Indeed, the encoding must be understandable by the non-ThingML part of the application, or this part must be changed to understand ThingML messages.

Proprietary systems: In many cases a ThingML component will have to communicate with external closed-source components, whose implementation cannot be modified. In this case, the only option is to adapt the ThingML end both in terms of encoding and transport.

Regardless of the situation, code needs to be generated (at least for the ThingML end) in order to handle the following tasks:

Encoding/Decoding: In order to exchange data between different platforms, a serialization scheme must be chosen. While there exist numerous solutions and standards, many distributed applications need to leverage several of them. Indeed, the choice among them can be driven by various concerns (bandwidth, human readability). Furthermore, it is not always a choice, in the case where a component needs to communicate with a proprietary system, which already fixed the format for some exchanges.

Sending/Receiving: Code must be generated to support both message emission and reception adapted to the targeted protocol. For most cases these operations rely heavily on a pre-existing library in the targeted language. But it can require additional aspects depending on the paradigm. For example, for synchronous communications, a message can not necessarily be sent at any time, and some queuing might be required.

Configuration and Link management: Before any message exchange occurs, a network interface has typically to be configured, and depending on the protocol a connection might have to be established. Furthermore, some network paradigms require some additional logic such as keeping track of connected clients.

Platform	Type	Language	Memory
Avr 8bits[2]	Micro Controller	C/C++	2-8 KB
TI MSP430	Micro Controller	C/C++	8 KB
ARM Cortex PSoC 4	Embedded Processor	C/C++	32KB
Espruino (ARM)	Embedded Processor	javascript	48KB
MIPS (Atheros AR9331)	Embedded Processor	C/C++	64 MB
Raspberry Pi	Embedded Processor	C/C++, javascript, java	0.5-1 GB
Intel Edison	Embedded Processor	C/C++, javascript, java	1 GB
x86	Processor	C/C++, javascript, java	GBs
Linux/Windows	Cloud	C/C++, javascript, java	GBs

Table 1: Platforms currently supported by the ThingML code generation framework

Encoding/Decoding is a separated aspect. Two strategies can be used to enforce this separation and ensure the reusability of the code generators:

ThingML Component ThingML can always be used to wrap existing native library and model an ad-hoc component handling the previously described tasks. This can be especially interesting when the network is at the heart of the logic of the application. While this solution may be the less demanding in terms of initial effort, upgrading the component with new messages with different signatures can be tedious mainly because of serialization/parsing. Therefore, as much as the code relative to encoding/decoding should be generated.

Adapt the code generator In the long run, effort in maintenance will be greatly reduced by adapting the code generation framework in order to fully generate communication-related code. Even if the initial effort may seem more important, the reuse of already available modules for similar tasks can significantly lessen it.

4.1 Plugin Mechanism

As communication can be broken up into two separated aspects, Serialization and Transport/Link management, the ThingML code generation framework propose a plugin mechanism as an extension point on this issue. It defines two main abstract classes describing two types of inter-operable plugins: Network plugins and Serialization plugins. Network plugins generate code supporting the transport of raw messages and manage the logic relative to the network paradigm. Serialization plugins handle the format of these messages. This approach allows a programmer to de-correlate transport and format of messages, offering a better re-usability. As each protocol, format or implementation comes with its own limitations, plugins are authorized to add rules to the ThingML checker. It offers the possibility to validate the input model with customized (and situation specific) constraints. For example, a plugin handling a low level protocol with payloads of limited size could check that the model does not contain messages that are too long. But this mechanism can also be used in order to check that each information required to configure the interface (*e.g.*, the network address of the remote peer, the baudrate of the link, path of the interface) is indeed provided. Or even checking that a same hardware interface is not used twice with different settings at the same time.

4.2 Network Plugins / Transport

In this section, we list the different protocols experimented and sum up the requirement that they raise.

UART: We experimented two different situations with Universal Asynchronous Receiver Transmitter. The first scenario was to use it in a point to point context. The first issue encountered was that it provide for a stream of byte, so a message system needed to be built on top of that in order to support separation of encoding and transport. The first method implemented was to allow for message preceded by a header containing the size of the message, the second was to use a system including a start byte, a stop byte, and an escape byte. The second scenario targeted a ring shape network, with both static and dynamic discovery of the size of the ring. After what messages had to include a TTL in order to determine if they were to be forwarded to the next neighbor or not.

I2C: Inter-Integrated Circuit provide synchronous master/slave communications. Therefore its use requires asymmetrical code for the master and the slaves. We experimented with a system of asynchronous exchanges built on top of it, based on slave messages queuing until master solicitation. But a simple master request / slave response can also be used. Static address configuration was used.

MQTT: MQTT propose a publish / subscribe message exchange through the mean of a centralized broker. It brings up the aspect of topic subscription and topic selection (in case of multiple topics publication). It also demonstrates the need for error management.

Websocket: Websocket relies on Client/Server connection which, once established, allows for exchanges at the initiative of both sides. On the server side, the generated code must enable the management of multiple connections and therefore offer the possibility of sending messages to one specific client, in addition to broadcast. The application can also require being notified when a new client connects or disconnects. On the client side, applications must be able to detect if the connection is broken or fails to be established. While all this interaction might not be relevant to the application and then can be hidden, in some case they are required.

REST: We conduct experiments with HTTP used to communicate in a REST style. As it was based on HTTP, it is a form of Request / Response exchange. It showed the need for failure management (and retry), which can be hidden to the application and fully generated or not.

ROS: ROS is a publish / subscribe broker. Different from MQTT, the setup of a ROS connection is centralized, while the rest of the exchanges can be peer to peer. Also, ROS provides a way to generate an API from message signature description similar to ThingML, enabling a straighforward integration to our framework. It enables high rate, low la-

tency exchanges, which call for proper buffer configuration when used with resource constrained devices.

Bluetooth: Bluetooth can in fact be used with a UART plugin, through the Serial Port Profile. Once the connection is established, a Bluetooth device using SPP can be seen as a classic Serial channel. In order to do so, it is necessary to declare custom ThingML messages corresponding to setup instructions for the Bluetooth device and to translate them into instruction that the device can understand.

BLE, Zwave: Some experiments with BLE and Zwave devices showed that a separate serialization scheme is not necessarily needed as one can be provided in the library enabling communication with some devices.

Through these experiments, it quickly appears that a part of the complexity coming from these various communication protocols can be hidden from the model, because they are not relevant to the application logic (Automatic re-sending of messages, connection establishment, a part of error management, encryption). Network plugin can generate code handling these "technicalities".

Meanwhile, a part of this complexity needs to be exposed to the ThingML code to enable a communication management flexible enough. In order to do so, the ThingML code generation framework differentiates two aspects of communication management (See Figure 3):

Configuration: Through the mean of annotation on the ThingML keyword protocol, configuration at design time can be provided to the code generator in order to generate a tailor-made network interface. **Control:** A part of the communication handling (such as error management, client management) is inherently dynamic and hence the need for information exchange between the application logic and the generated network library at run time.

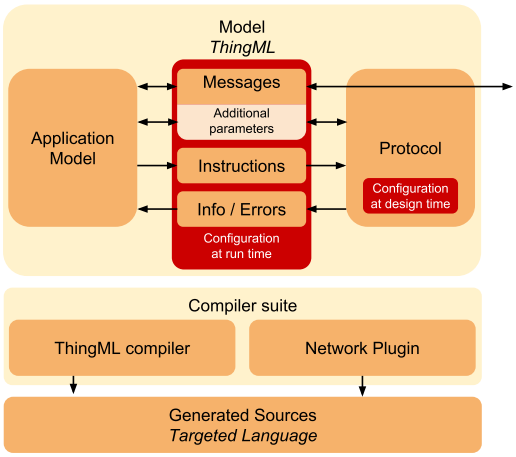

Figure 3: Overview of generated network library

Two broad types of annotations can be used to control more precisely a generated library at run time. Annotated messages can be transformed into instructions addressed at the generated network library and not forwarded (for example, a re-connection instruction). Similarly annotated messages can become feedback originating from the network library and transmitted to the application (for example if the connection has been lost). But additional parameters (that will not be forwarded) can also be added to messages in order to customize at run time the way those messages are forwarded (for example destination information).

```
thing fragment myMsgs { //Control at run time
  message reconnect()
    @websocket_instruction "reconnect";
  message connection_lost()
    @websocket_feedback "connection_lost";
  message msg1(Param : Float, ClientID : UInt16)
    @code "101" //ID for serialization purposes
    @websocket_client_id "ClientID";
}

protocol Websocket //Configuration at Design time
  @websocket_server "true"
  @websocket_max_client "16"
  @websocket_enable_unicast "true"
  @serialization "msgpack"
;

configuration myCfg {
  instance i : myThing // The thing myThing can send
                       // messages through websocket in a
  connector i.myPort over Websocket //transparent way
}
```

This architecture offers a trade-off between abstraction and customizability. It allows to hide or expose parts of the communication paradigm on demand, depending on what the application requires. It provides a way to extend the ThingML code generation framework for message exchanges, while relying if necessary on serialization plugins described in the following section.

4.3 Serialization Plugins / Encoding

Serialization plugins are in charge of generating both the serialization and parsing of a set of ThingML messages. Note that serialization and parsing are not necessarily perfect mirrors, as ports can be asymmetrical.

Binary: Efficiency, both in term of messages size and parsability, is required when communicating with resource-constrained devices. The simplest way to serialize data is to write them in forms of raw bytes. This kind of serialization is very similar to the way message are en-queued in ThingML generated FIFOs in C, so it can be generated with very little modification to the code generator.

Message Pack: MessagePack[3] is a lightweight data serialization format inspired by JSON but binary-coded. As in JSON the size depends on its value and not only on its type. While implementations of a message pack library existed for most common languages, we had to implement our own library to cope with the constraints of our platform for this experiment (an AVR 8 bits microcontroller)[4].

JSON, SenML, XML: SenML[5] defines media types for representing simple sensor measurements in JSON. SenML native libraries for the targeted languages already exist, but the maintenance of a serialization / de-serialization is tedious. Meanwhile, they can often be wrapped in code generator in a straightforward way, which greatly lessen the maintenance effort. Nonetheless, for resource constrained devices library implementing this standard must be both chosen and used carefully in order to be viable, as a straightforward implementation for those rather verbose formats can quickly use up the available RAM of a microcontroller.

[3]http://msgpack.org/
[4]http://github.com/HEADS-project/arduino_msgpack
[5]http://tools.ietf.org/html/draft-jennings-senml-10

Proprietary/Custom: In order to integrate proprietary (or non customizable) devices, adopting a fixed set of messages in addition to a specific serialization is often required, as showed our experimentation with some Bluetooth devices, or with the Multiwii Serial Protocol for some drone flight controller. In this case in addition to the serialization aspect, it is necessary to model the set of pre-existing messages in ThingML in order to make them usable.

```
//ThingML definition
message myMessage(UInt8 u, Int32 i) @code "6";
//ThingML use
port!myMessage(3,-257)
//Raw byte array
00 06 03 ff ff fe ff
//Message Pack
81 a9 6d 79 4d 65 73 73 61 67 65 92 03 d1 fe ff
//Json
{"myMessage":{"code":6, "u":3, "i":-257}}
```

In the end, in order to be flexible enough, the code generation framework must provide a way to integrate rapidly existing native library to enable the use of standard serialization schemes. But it also needs to support custom/proprietary communication protocols, by modeling their existing messages in ThingML and generating code compatible with non-modifiable software and hardware components.

Moreover, to meet the various needs of compatibility, a serialization plugin needs to be usable in two different ways:

- It can either be provided to a network plugin in order to generate fully executable code for the ThingML end.
- It can also be used to generate the methods separately in order to integrate them directly in sources written in the targeted language.

These different usages offer more flexibility on the choice of the end to adapt. One can either adapt the ThingML end of a link, but also the end running a software component written in another language. While enabling this choice, it fits the maintainability requirement as it allows regeneration of code after modification of the set of messages.

5. THINGML IN PRACTICE

This section provides a brief overview of three projects in different domains developed with ThingML (Version 3), which all take advantage of the code generation framework.

5.1 TellU Safe@Home System

TellU's Safe@Home system [14] is built around a home gateway which runs on a Raspberry Pi 2 (1 GHz ARMv7, 1GB RAM, Linux). This gateway is connected to a number of field nodes (typically one per main room in the house) via WiFi, and also to the TellU Cloud via Ethernet. The gateway is currently implemented in Node.JS, as most of the technical libraries were already available (Z-Wave, Bluetooth Low Energy, MQTT, etc) and could rapidly be integrated by TellU to implement their gateway.

The cloud back-end is implemented in Java and has been developed by TellU over the past ten years. This represents an important legacy for TellU. The back-end communicates with the rest of the system through edge connectors, currently offering APIs over HTTP/REST and MQTT.

Field nodes run on an Intel Edison (400 MHz x86, 1GB RAM, WiFi and Bluetooth Low Energy (BLE)). Each field node has a pressure cell integrated in order to provide an accurate measurement of the pressure in each room. Fields nodes are also currently implemented in Node.JS using Intel's MRAA library to control the Edison's GPIOs.

A wearable sensor node running on a low power resource-constrained ARM Cortex M3 (80 MHz, 256 KB RAM) also integrates a pressure cell and regularly broadcasts air pressure measurement to all field nodes that are in the BLE range (typically 10m indoor). Based on these pressure measurements and the intensity of the BLE signal, field nodes can determine the position of the person in the house and if the person has fallen (by computing an air pressure differential between the person's sensor and the fixed pressure in the field node). Due to the stringent resource constraints on this device (driven by the fact the wearable sensors should be able to run several days on a single charge), this device has to be implemented in C. The only alternative is assembly.

In addition, a set of sensor nodes is deployed in different rooms to measure temperature and light. Those nodes run on an Arduino Yùn, which is composed of a resource-constrained microcontroller (16 MHz AVR-8bit, 2.5 KB RAM, no OS) and an embedded Linux processor (400 MHz MIPS, 64 MB RAM, WiFi, OpenWRT). The microcontroller part of the Yùn is used to interact with the physical temperature and light sensors while the MIPS CPU and its embedded WiFi is used to communicate with the Gateway.

Finally, the gateway also integrates a Z-Wave radio chip that interacts with a set of devices (switches, etc). Those devices are proprietary and the firmware they run as well as the protocol they use to communicate cannot be modified.

Lessons learned

Many developers at TellU have a previous experience with UML and code generation using the JavaFrame approach [9]. For reasons similar to the ones described in Section 2.1, none of the legacy TellU back-end was developed using UML.As TellU's vision is now to push some of their code as close as possible to the physical world (in gateways and even devices), the ability of ThingML to quickly wrap existing code (either theirs or open-source library) was an accelerator for them to implement this vision. TellU's developers were skilled Java (server-side) and JavaScript (in browser) developers, used to x86 architecture and HTTP/REST protocols. They are now deploying Node.JS and C code on a large set of heterogeneous devices.

5.2 Micro-aerial vehicle platform

The goal of this project was to create a lightweight indoor aerial vehicle platform to support a set of research projects. ThingML was chosen to implement the software framework and provide a way to easily build drone with different sets of sensors and algorithms.

The platform (a quadcopter) is built around a Quanum Pico 32-bit Flight Controller board, including mainly a microcontroller STM32F103CB, an Inertial Measurement Unit (IMU) MPU-6050, and a radio receiver CYRF6910. This board runs an open source firmware (cleanflight).

Our approach consists into providing support for extension of the flight controller through additional an board based on an ATmega32u4 microcontroller where code can be deployed to interact with new sensors. In this example, we will detail the use of a ToF (Time of flight) altitude sensor to create a semi autonomous micro aerial vehicle, self

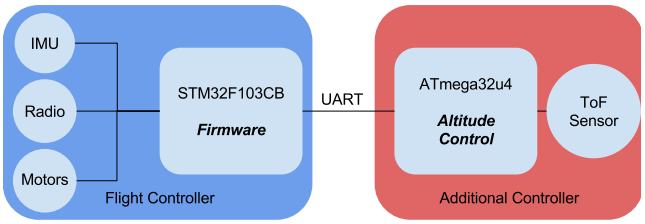

Figure 4: Hardware configuration of the drone

regulating its altitude as illustrated in Figure 4.

The firmware run by the flight controller understand command formatted with MultiWii Serial Protocol over a UART.

```
0   8   16  24  32  40       ... ...
*---*---*---*---*---*---------*---*
| $ | M |Dir|len| T | Payload |CRC|
*---*---*---*---*---*---------*---*
```

MSP is a message-based protocol that allows request / response exchanges with the flight controller. Messages are formed (as shown above) of a header (containing the payload length), a payload of parameters, and ends with a CRC.

Before being able to generate MSP messages from ThingML, three things need to done: (*i*) MSP messages need to be model in ThingML, (*ii*) the ThingML UART plugin must be configured to read MSP header and extract the size of the payload and then build a message-oriented system on top of the stream-oriented system that is UART, and (*iii*) an MSP serialization plugin must be written.

```
thing fragment MSPMsgs {
    message req_MSP_RC() @code "105";
    message MSP_RC(rc1: UInt16, rc2: UInt16, rc3: UInt16,
        rc4: UInt16, rc5: UInt16, rc6: UInt16,
        rc7: UInt16) @code "361";
    message MSP_SET_RC_OFFSETS(roll : UInt16, pitch : UInt16,
        yaw: UInt16, throttle: UInt16) @code "80";
    ...
}
```

As request messages and their response share the same name and ID in MSP, while not the same signature, they must be defined as two different messages, by prefixing request message with 'req_'. MSP message IDs are limited to one byte in length. As ThingML allows for longer IDs, we added a MSP-specific rule to the plugin in order to check that no message with an ID greater than 511 (one byte plus a bit for direction) is used on a connector using MSP.

Lessons learned

The effort to integrate support for MSP in the ThingML code generation framework represents a 130 LoC MSP serialization plugin, and the modeling of the MSP messages with ThingML. As for the communication layer, it could reuse the existing UART plugin with no modification.

All the serialization plugins (binary, JSON, MSP, Message Pack) and communication plugins (UART, WebSocket, MQTT, etc) implemented so far are all within 100-300 LoC. This indicates that integrating a new serialization or a new protocol is rather accessible.

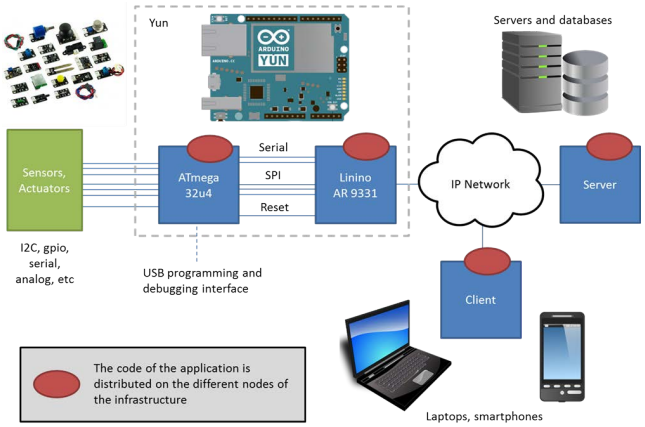

Figure 5: Arduino Yùn Controller

5.3 Arduino Yùn IoT Framework

This experiment consisted of using the ThingML framework to fully generate code targeted at Arduino Yùn[6] and exploiting its networking capabilities.

In this situation, the Arduino Yùn is in charge of controlling in real time actuators and sensors with its microcontroller chip. In addition, its CPU can process the data, store them locally temporarily, forward them to a server regularly, and enable control over actuators and data for clients.

This scenario relies on multiple platforms (Atmega32u4 microcontroller, Atheros AR9331 processor, Server, clinet web browser) and various communications (Serial link between the Yùn two chips, HTTP REST, MQTT, Websocket between the AR933S and the rest of the network). Communications libraries targeting embedded platform need to be integrated to the code generator. The build generator has been customized to enable cross compilation for MIPS.

This adaptation now allows the modeling and generation of diverse applications targeting this kind of network of devices such as a hardware debugger for Arduino Yùn[7] and the ViMoSys a vibration monitoring system[8].

Lessons learned

This experiment involves a large set of protocols, that all could be easily integrated as plugins. In addition, it required to adapt the existing C compiler in order to support the MIPS architecture. As the code generated by ThingML does not rely on any library (other than the ones needed by the network plugins) and only uses well-supported features of the target language, the only extension point that needed to be modified was the build generator in order to generate a different Makefile.

6. RELATED WORK

This section briefly discusses related work regarding the adoption of MDE in practice and approached for implementing modular code generators.

6.1 Adoption of MDE

Over the years, the community has made significant effort to develop the MDE vision. However, there is still relatively

[6]https://www.arduino.cc/en/Guide/ArduinoYun
[7]https://github.com/Lyadis/Yun-Arduino-Debugger
[8]https://github.com/Lyadis/Yun-IMU

limited work reporting on the result and lesson learned from the practical adoption of MDE in the software development process [1]. Among the existing literature on the topic, it is worth mentioning [18] which contains a set of paper reporting on a MDE success stories.

In [19] the authors present the result of a survey built from 39 interviews with MDE practitioners. The paper explores the obstacles to the practical adoption of MDE in the industry and especially focuses on the tooling. The study build on previous work [12] which reports on the industry practice of MDE and [11] which focuses more on the organizational impact of MDE [19].

Our experience applying MDE for the implementation of distributed reactive system corresponds a very specific use of MDE but our findings confirms the general observation presented in [19]: tools should be matched to the users (and not the other way around), MDE should be strategically applied to specific problems and not blindly to the whole development process and finally, the development process remains more important than the tool itself.

In [3] Clarke and Muller discuss their respective experiences in building start-up companies around MDE tools. The paper details the lessons learned from these experiences both in terms from the entrepreneur point a view and from the technical point of view. The paper points 13 technological issues for the adoption of MDD. Many of these issues match our observation in applying ThingML with industry partners. More specifically the adoption of UML, maturity issues, misconceptions about platform independence and practical aspects such as text vs. diagram and code generation confirm our own experience.

6.2 Code generation frameworks

In [17] the authors identify and motivate the need for better practices to develop modular and re-usable code generators. The paper proposes to reuse product line mechanisms to model, structure and compose families of code generators. In [16] the same group proposes a concept for code generator composition applied to a robotics case study. The motivation for this work is similar to our motivations for building the ThingML code generation framework and goes one step further in formalizing the valid extension points and alternative combinations. The ThingML approach is bottom-up and has been built around using a traditional object-oriented approach. However, we do acknowledge the need for additional information, especially in order to add constraints to define the valid combination of plugin across the different extension points.

In [20] the authors propose to use aspects to create modular code generators. The approach allows un-tangling the different aspects of the code generator in order to improve the flexibility and maintainability of code generators. The second version of the ThingML tool was implemented using similar ideas and weaving the code generator components as aspects into the meta-model using Scala case classes. While the design was elegant, this solution made it very difficult for the average developer to modify and customize the code generator to its needs. That is why the code generation framework was fully re-implemented using classic object-oriented design and Java as the programming language.

In [13] the authors propose a code generation framework for CPS. The paper insists on the multi-disciplinary nature of building software for CPS. The proposed model include a set of 3 views: a functional model, platform model and a runtime model. The code generation framework allows generating code for microcontrollers, DSPs and FPGAs. The approach has been applied to the case study of a Lathe CNC System. The notation used are graphical and code generation is template based. As compared to the ThingML framework, it is unclear how customizable the generated code can be and what is the cost of adding support for new platforms.

In [2] the authors propose a DSL for robot programming. The DSL abstract over the programming APIs provided by ROS and allow generating code. The language proposed is very specific to robotics and even to the type of robot developed in the case study. Compared to ThingML, the language is way more specific to the application and its broader applicability remains to be demonstrated.

7. DISCUSSION AND CONCLUSION

The ThingML approach and tool are developed as an open-source project. The implementation of the different versions of the tool as well as the code generators and plugins described in this paper are available on Github [4]. The features of the language and tool are documented through a set of tutorial and exercises available in a separate repository [10]. Over the past 2 years, over 400 students and developers have used ThingML.

At this point, the main lessons we can draw from our experiences applying ThingML are:

1. ThingML builds on a rich MDE state of the art. Several iterations, both on the approach and tools, have however been necessary for being usable and beneficial for practitioners.

2. Having a single integrated language with a clear semantics together with dedicated tool support was a key to succeed. Approaches combining and composing different models and formalisms, turned out impractical [6]. We believe that more research and tooling is needed before such techniques can be practical for the domains we have considered.

3. One benefit of MDE (and ThingML) is the ability to have platform-independent specifications. However, to remain practical MDE approaches should not require more modeling/formalization than is useful/exploited: MDE should not introduce any overhead when no benefit is expected or desired (*e.g.* target platforms are known *a priori*, building on existing libraries/APIs, interfacing with existing/legacy components). This is critical for adoption since no modern software system is developed from scratch, and re-modeling those artifacts (a.k.a re-inventing the wheel) is simply not acceptable.

4. Code generation is not popular among practitioners. This bad reputation is typically based on experiences with tools producing code with low readability, hard to integrate with existing systems and other components and very hard to maintain and/or evolve. Our experience with ThingML is that MDE should produce better code more efficiently rather than propose to replace code with models. We believe that it is critical to provide practitioners with ways to fully customize the code generators to their needs, habits and projects.

Acknowledgements

This work has been supported by EU FP7 HEADS (grant agreement: 611337) project.

8. REFERENCES

[1] J. Aranda, D. Damian, and A. Borici. Transition to model-driven engineering: What is revolutionary, what remains the same? In *Proceedings of the 15th International Conference on Model Driven Engineering Languages and Systems*, MODELS'12, pages 692–708, Berlin, Heidelberg, 2012. Springer-Verlag.

[2] J. Baumgartl, T. Buchmann, D. Henrich, and B. Westfechtel. Towards easy robot programming: Using dsls, code generators and software product lines. In *in Proceedings of the 8th International Conference on Software Paradigm Trends (ICSOFT-PT 2013*, pages 147–157, 2013.

[3] T. Clark and P.-A. Muller. Exploiting model driven technology: a tale of two startups. *Software & Systems Modeling*, 11(4):481–493, 2012.

[4] F. Fleurey, B. Morin, and O. community. Thingml source code repository: https://github.com/sintef-9012/thingml, 2016.

[5] F. Fleurey, B. Morin, and A. Solberg. A model-driven approach to develop adaptive firmwares. In *2011 ICSE Symposium on Software Engineering for Adaptive and Self-Managing Systems, SEAMS 2011, Waikiki, Honolulu , HI, USA, May 23-24, 2011*, pages 168–177, 2011.

[6] F. Fleurey, B. Morin, and A. Solberg. A model-driven approach to develop adaptive firmwares. In *Proceedings of the 6th International Symposium on Software Engineering for Adaptive and Self-Managing Systems*, pages 168–177. ACM, 2011.

[7] F. Fleurey, B. Morin, A. Solberg, and O. Barais. MDE to manage communications with and between resource-constrained systems. In *Model Driven Engineering Languages and Systems, 14th International Conference, MODELS 2011, Wellington, New Zealand, October 16-21, 2011. Proceedings*, pages 349–363, 2011.

[8] R. France. Why johnny can't model. *Software & Systems Modeling*, 8(2):163–164, 2009.

[9] Ø. Haugen and B. Møller-Pedersen. Javaframe: Framework for java-enabled modelling. In *Proceedings of Ericsson Conference on Software Engineering*, 2000.

[10] HEADS. Thingml tutorials: https://github.com/heads-project/training, 2016.

[11] J. Hutchinson, M. Rouncefield, and J. Whittle. Model-driven engineering practices in industry. In *Proceedings of the 33rd International Conference on Software Engineering*, ICSE '11, pages 633–642, New York, NY, USA, 2011. ACM.

[12] J. Hutchinson, J. Whittle, M. Rouncefield, and S. Kristoffersen. Empirical assessment of mde in industry. In *Proceedings of the 33rd International Conference on Software Engineering*, ICSE '11, pages 471–480, New York, NY, USA, 2011. ACM.

[13] S. Li, D. Li, F. Li, and N. Zhou. Cpsicgf. *Microprocess. Microsyst.*, 39(8):1234–1244, Nov. 2015.

[14] B. Morin, F. Fleurey, K.-E. Husa, and O. Barais. A generative middleware for heterogeneous and distributed services. In *19th International ACM Sigsoft Symposium on Component-Based Software Engineering, Venice, Italy, April 5-8, 2016*, 2016.

[15] P. E. Papotti, A. F. Prado, W. L. Souza, C. E. Cirilo, and L. F. Pires. *Advanced Information Systems Engineering: 25th International Conference, CAiSE 2013, Valencia, Spain, June 17-21, 2013. Proceedings*, chapter A Quantitative Analysis of Model-Driven Code Generation through Software Experimentation, pages 321–337. Springer Berlin Heidelberg, Berlin, Heidelberg, 2013.

[16] J. O. Ringert, A. Roth, B. Rumpe, and A. Wortmann. Code generator composition for model-driven engineering of robotics component & connector systems. *CoRR*, abs/1505.00904, 2015.

[17] A. Roth and B. Rumpe. Towards Product Lining Model-Driven Development Code Generators. In *Proceedings of the 3rd International Conference on Model-Driven Engineering and Software Development : Angers, France, 9 - 11 February 2015*, pages 539–545, Aachen, Feb 2015. 3rd International Conference on Model-Driven Engineering and Software Development, Angers (France), 9 Feb 2015 - 11 Feb 2015, SciTePress.

[18] D. D. Ruscio, R. F. Paige, and A. Pierantonio. Guest editorial to the special issue on success stories in model driven engineering. *Science of Computer Programming*, 89, Part B:69 – 70, 2014. Special issue on Success Stories in Model Driven Engineering.

[19] J. Whittle, J. Hutchinson, M. Rouncefield, H. Burden, and R. Heldal. *Model-Driven Engineering Languages and Systems: 16th International Conference, MODELS 2013, Miami, FL, USA, September 29 – October 4, 2013. Proceedings*, chapter Industrial Adoption of Model-Driven Engineering: Are the Tools Really the Problem?, pages 1–17. Springer Berlin Heidelberg, Berlin, Heidelberg, 2013.

[20] S. Zschaler and A. Rashid. Towards modular code generators using symmetric language-aware aspects. In *Proceedings of the 1st International Workshop on Free Composition*, FREECO '11, pages 6:1–6:5, New York, NY, USA, 2011. ACM.