MDE to Manage Communications with and between Resource-Constrained Systems

Franck Fleurey[1], Brice Morin[1], Arnor Solberg[1], and Olivier Barais[2]

[1] SINTEF IKT, Oslo, Norway
[2] IRISA, University of Rennes1 and INRIA

Abstract. With the emergence of Internet of Things (IoT), many things which typically used to be isolated or operated in small local networks, will be interconnected through the Internet. One main challenge to tackle in IoT is efficient management of communication between things, since things can be very different in terms of available resources, size and communication protocols. Current Internet-enabled devices are typically powerful enough to rely on common operating systems, standard network protocols and middlewares. In IoT many devices will be too constrained to rely on such resource-consuming infrastructures; they run ad-hoc proprietary protocols. The contribution of this paper is a model-based approach for the efficient provisioning and management of the communication between heterogeneous resource-constrained devices. It includes a DSML which compiles to a set of interoperable communication libraries providing an abstract communication layer that can integrate both powerful and resource-constrained devices. The approach is implemented in an IDE for the development resource-constrained Things.

1 Introduction

We exploit, interact and rely on things in our everyday life (e.g., house-hold appliances, clothing, cars, lights, buildings, mobile phones, etc). Currently, more and more of these things are equipped with sensors, computing power and communication capabilities, leading to the emergence of the Internet of Things (IoT). A vast number of independent or embedded sensors and sensor networks will form the basis of the IoT infrastructure. New innovative applications exploiting the IoT paradigm are already emerging in domains such as Ambient-Assisted Living (AAL), intelligent transport systems and environmental monitoring.

Building advanced services that involve heterogeneous sensors and things that need to communicate and collaborate can be complex and time consuming for the following reasons:

– **Resource Constraints**: While the business logic of each individual thing is often rather simple, the resource constraints (CPU, memory, power, etc) make it challenging to efficiently implement this logic. Typically, these things have to run autonomously for an extended time, which require minimizing their power consumption while still providing quality of service. For example, sensors monitoring the environment are spread in locations not easily

accessible, making such requirements of high importance to maintain the system services.

- **Heterogeneity**: The IoT includes a wide variety of different nodes, ranging from powerful servers to small things operated by micro-controllers. Powerful nodes can run common operating systems and rely on common frameworks and protocols to implement software engineering best practices. However, IoT will include a vast number of resource-constrained things: some can run light operating systems like TinyOS, some can only run C or C-like languages, some can only run low level assembly code. In addition there is a wide range of (wireless) communication technologies (e.g., WiFi, Bluetooth, ZigBee) which provide different trade-offs regarding power-consumption, range, reliability, etc, and which should be combined in a sensor network.
- **Independent development of things and services**: Things (e.g., physical devices operated by a C-based micro-controller) and advanced services (e.g., Java application running on a server) relying on a set of networked things are usually developed concurrently, by different teams having different competencies. This could lead to misalignments and inconsistencies.
- **Interoperability**: Different IoT-related standards have emerged, such as the ones proposed by the Sensor Web Enablement Working Group [3], to provide interoperability support in the context of sensor networks. While these standards address clear needs, their realization (usually verbose XML-based document) cannot fit to the most resource-constrained devices, due to the technical overhead they imply. Instead micro-controllers rely on ad-hoc and highly optimized protocols, which better fits the resource constraints and improve the reliability and speed of wireless communications.

For these reasons, developing and IoT application based on a heterogeneous Wireless Sensor Network (WSN) is challenging. However, solution exists for tackling these issues for the powerful nodes: standards define ways to represent and exchange data in a homogeneous way, some middlewares, software frameworks and networking stacks solve part of the heterogeneity challenges and the business logic can be realized using classic development techniques including MDE. Unfortunately, these solutions cannot directly be applied for the most resource-constrained nodes. As a result, software embedded in the most resource-constrained things are in most cases manually developed in C and ASM. Platforms are too small to embeed OS, middlewares or frameworks so the code is often a tangled mix of business logic, communication and hardware drivers.

This paper presents an innovative Model-Driven approach to support the efficient development IoT applications over heterogeneous WSNs. In particular we have developed a DSML called ThingML, which aims at promoting software engineering best practices for the specific case of resource-constrained systems. ThingML comes with editors and checkers and a set of transformations and code generators which currently target Java and several micro-controller platforms (e.g., TI MSP, Atmel AVR and Arduino). The contribution presented in this paper is a model-driven approach for engineering and managing efficient com-

munications with and between heterogeneous and resource-constrained things within the IoT, with proper support for independent development.

The paper is organized as follows. Section 2 introduces our running example and analyzes the problems related to communication with and between resource-constrained things. Section 3 presents our model-driven approach to tackle these issues. Our approach is validated on two case studies in Section 4. Section 5 presents related work and Section 6 concludes and opens some perspectives.

2 Problem analysis and illustrative example

This section presents a running example used throughout the paper. The example both illustrate our approach and motivates our work by detailing challenges related to development of resource constrained IoT services.

CoffeeSpy is an experimental device that monitors a coffee machine using technologies typically found in wireless sensor network (WSN) applications. The left side of Figure 1 presents an overview of the sensor hardware structure. The core of the device is an 8bits AVR micro-controller with 32ko of flash memory and 2ko of RAM memory connected to 3 sensors: *i*) a high-resolution infra-red temperature sensor, *ii*) an infra-red distance sensor, and *iii*) a light sensor.

The wireless communications are realized via an XBee radio chip which implements the ZigBee protocol (http://www.zigbee.org/). This hardware set-up is rather simple, however, the CoffeeSpy device is representative for devices found in application domains such as environmental sensor networks or industrial process control and monitoring systems, as it uses typical off-the-shelf components. The CoffeeSpy application provides real time information about:

- the temperature and freshness of the coffee using the temperature sensor,
- the number of cups which have been pored since the coffee was made using the distance sensor and the temperature sensor, and
- the activity in the coffee room using the light sensor and the distance sensor.

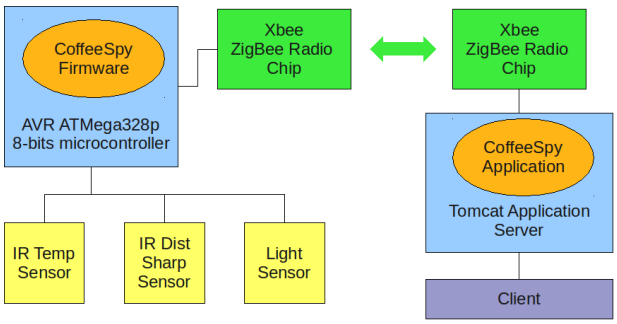

Fig. 1. Architecture of the CoffeeSpy device

As shown in Figure 1, the device communicates with a server equipped with a ZigBee adapter. From a software point of view, the application is composed of two components:

- The server application implemented in Java. The server has a lot of resources available in terms of computing power, memory, software libraries, middlewares, etc. Typical software modeling techniques and development processes can easily be applied in the development of this part of the application.
- The firmware running on the micro-controller, programmed in C. It is constrained in terms of energy, computing power, memory, bandwidth, etc. These constraints make it impossible to rely on standard middlewares and operating systems to implement the device functionalities.

This heterogeneity introduces some accidental complexity for the development of the CoffeeSpy application. Alternatives can be explored to try and avoid this situation, however, they raise other concerns:

- Using a more powerful core for the CoffeSpy which could run a real OS (e.g., embedded Linux) with standard software libraries, network protocols and communication middlewares. This strategy can be applied to some chosen devices but cannot be applied to all the devices of the network: the hardware for such a device would be 50 to 100 times more expensive and its power consumption would be about 1000 times more. Because devices and sensors are deployed in large numbers and often need to be battery-operated and environmentally friendly (green computing), limiting the power consumption is a primary concern.
- Reducing the features of the firmware to the absolute minimum and delegate all the functionalities to large nodes like the server of the CoffeeSpy application. A minimalistic firmware would simply read values from sensors and transmit them periodically. Unfortunately this would not be a good solution for several reasons:
  - It is inefficient. To monitor the coffee, the application can sample the temperature of the coffee every minute, but detecting motion in front of the coffee machine requires more frequent sampling (several times per second). While this can easily be implemented, it is highly inefficient both in terms of power and bandwidth consumption and would thus not scale to any realistic sensor network.
  - It is a threat to reliability. Devices monitoring and controlling equipments, should be able to sustain critical functionalities even if the remote and more powerful nodes are not available. Typically time-critical and safety-critical features should be implemented in the controllers to keep these services available even in case of failure of the other nodes or communication links. The logic of the application should be distributed among all the nodes (including the smallest ones) to avoid critical points of failure.
  - It does not scale. The CoffeeSpy example has only two nodes but real-life sensor networks are typically composed of hundreds or thousands nodes.

> The applications running on these nodes need to cope with the sporadic availability of other nodes, the discovery of new nodes and adapt to their environment to provide the best possible services. Such dynamicity cannot be implemented in a centralized way and require each node of the network to implement behavior which contributes to the overall application.

Designing a distributed system which involves resource-constrained nodes is a complex trade-off between maintainability, hardware costs, reliability, power consumption, etc. Once the hardware is selected, the main instrument to adjust this trade-off is the distribution of the features on the different nodes. For example in the CoffeeSpy application, the freshness of the coffee can either be computed on the micro-controller or on the server by analyzing raw data coming from the sensor. As such this would probably be the most convenient solution for our application since many libraries exist in Java to process data. However if we want to extend the CoffeeSpy device with a display (located on the device itself) which shows the age of the coffee, it would be better to have this computation made locally in order to keep the display properly updated and functional even if the server side is down. This would in addition avoid back-and-forth exchange of messages on the wireless network.

The only realistic solution to keep control on this complex trade-off is to provide efficient support for moving functionalities from nodes to nodes both at design-time and during maintenance and evolution. This actually means co-evolving all the nodes contributing to a function in a consistent way. For example, if the freshness of the coffee is computed on the server, then the CoffeeSpy device only needs to provide a service for reading temperature. However if the device does this computation it has to be refactored to also transmit the age of the coffee. There already exist some elegant solutions to seamlessly communicate both with local objects and remote objects in an homogeneous environment. However, when it comes to micro controller-based devices, such an extension means that both the firmware of the device and its driver on the server has to be extended with the appropriate computation and communication functionalities. Even worse, in a realistic setup, the different nodes of an application are typically developed by different teams and have different life-cycles. The goal of the approach proposed in this paper is to allow defining in a simple way the interfaces between devices and to automatically derive APIs to support communications, and stubs to support the development and testing of nodes in isolation.

3 Approach

The approach proposed in this paper is developed as part of an IDE called ThingML[3] for the development of resource-constrained systems. The idea of our approach is to specify protocols in a ThingML model using a concise and comprehensive syntax, as illustrated in Figure 2, and to fully generate code:

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

– Efficient API for the serialization and de-serialization of messages in/from arrays of bytes, as presented in Section 3.1.
– Handlers for managing message-specific communication features, as presented in Section 3.2.
– Mock-ups and interactive simulators to enable independent development of things and services, as presented in Section 3.3.

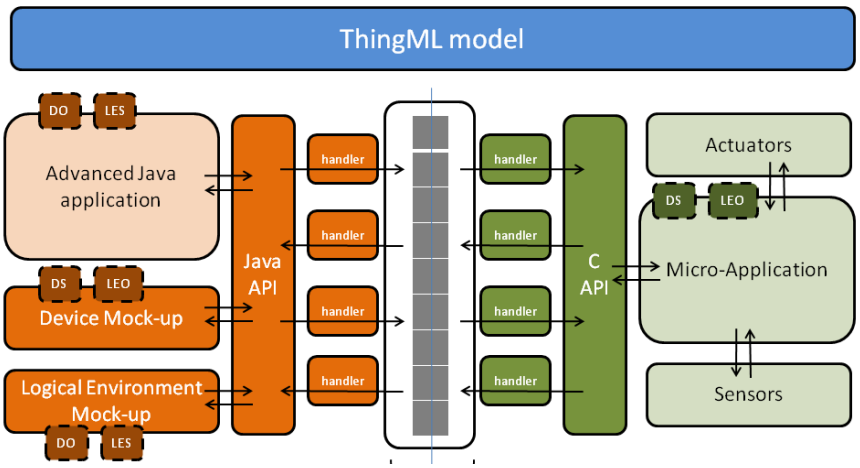

Fig. 2. Overview of the proposed approach

The idea of ThingML is to build a practical development environment and methodology to make typical software engineering good practices available to the development of resource-constrained embedded systems such as micro-controller applications. We believe that model-driven engineering provides the right tools to address this problem because models-based approaches do not need to rely on advanced run-time frameworks, operating systems and middleware to be applicable. Models can be analyzed, tested and verified and exploited to generate optimized code which target resource-constrained platforms. In previous work we have shown how ThingML can be used to produce adaptive firmware using a high-level adaptation DSL and aspect-oriented modeling techniques applied to state machines [7]. In this paper we focus on the generation of interoperable communication APIs to communicate with and between things. To this end, we use a sub-set of the ThingML language to specify messages.

The ThingML metamodel, editors and code generators are available as part of the ThingML open-source project[4].

3.1 Generating interoperable APIs for things and services

An important aspect when setting up an infrastructure for the IoT is the ability to properly connect and interoperate the physical "things" with advanced services relying on standards [3]. Typically, the raw data provided by the sensors

[4] https://github.com/ffleurey/ThingML

(e.g., the actual value of a light sensor) is usually not relevant for end-users. Instead, end-users are more interested, for example, by a qualitative estimation of the light in a room.

To ease the integration and interoperability of (C-based) things and (Java-based) services we generate C APIs for the devices and Java APIs for the more powerful nodes. All these APIs are able to communicate together. Fully functional APIs are generated from ThingML models, such as the one illustrated in Figure 3, which define messages with their parameters, and their directions (sent of received) w.r.t. to the physical device: a message sent by the device is a message received by the service, and vice-versa. In the generated code, each message is clearly reified (e.g. by a Java class) and we provide the designers with a factory to create message either by passing parameters, or from a serialized format. The code related to serialization/deserialization is also fully generated.

Our code generators thus alleviate designers from directly manipulating low-level structures, who use the generated APIs instead. The benefits of this approach are:

- **Performance**: Messages are automatically serialized as arrays of bytes, which are the most concise way of representing data, and by consequence the fastest way of transmitting data on (wireless) networks. They are also much faster to parse than XML data. As a comparison [9], this is 3 to 10 times smaller than XML-based data, and 20 to 100 times faster to parse.
- **Interoperability**: Our approach ensures or facilitates different levels of interoperability:
  - **Programming languages**: Relying on the lowest possible abstraction (bytes) ensures the interoperability at the lower level: Java, C, etc can interpret bytes. However, designer only manipulate higher lever representations: Java POJOs, C structures, etc.
  - **Communication links**: Arrays of bytes are common inputs/outputs accepted by most communication stacks: serial port, ZibBee, Bluetooth, etc. This ensures the interoperability among different links, which is an important point in WSN.
  - **Standards**: The generated high-level APIs provides a good support for interoperability with standards. For example, it is straightforward to instantiate XML templates on values of POJOs.

In addition to the API we also generate the interfaces of two Observer patterns. The first Observer aims at managing and facilitating the communications with the device, and the second one for the communications with clients of the device: other devices, heavy applications running on a server, etc. Using these observers, it is rather easy to develop applications that for example log the data of devices, or drive the devices. We also fully generate such an application (an interactive simulator), as described in Section 3.3.

3.2 Generating Light-weight and message-specific Protocols

Networking is a domain where a large number of techniques, standards and tools are available to assist software development. These techniques very often rely on

the 7 OSI layers (or similar stacks) to implement networking capabilities and offer high-level and reliable communications. The layers are generic components which require the services of the layer below them and offer a set of services to the layer above. This componentization enables easy deployment and reuse in various contexts, but does not suit well resource-constrained systems:

- The generality implies resource penalties. The different layers typically use different memory buffers to store the incoming an outgoing data and perform their tasks, which multiply the memory usage of the stack. Collapsing the layers together would allow building a more efficient networking component requiring only a fraction of the resources.
- The applications running in a resource-constrained environment typically do not need all the features of a fully fledged protocol stack. In most cases the type of messages can be fixed at design time. Depending on the application, protocol features such as acknowledgments, timeouts, routing, error detection, error correction, assembling and disassembling of packets and so on might not all be required.

```
// Raw Data from the sensors
message subscribeRawData( interval : Integer ) @code "19";
message unsubscribeRawData() @code "22";
message getRawData() @code "20";
message rawData(temp : Integer, dist : Integer, light : Integer) @code "21";
receives subscribeRawData, getRawData, unsubscribeRawData
sends rawData

// Simple Ping
message ping() @code "66"
    @sync_ack "pong" @timeout "1000" @retry "3";
message pong() @code "67";
receives ping
sends pong

// Get the data from individual sensors
message GetTemperature() @code "1"
    @sync_response "TemperatureValue#v" @timeout "500" @retry "0";
message TemperatureValue(v : Integer)
    @code "2";
```

Fig. 3. Messages with communication features

In practice, the networking component embedded in micro-controllers is manually implemented as a single optimized component developed for each specific application [15]. This has great advantages in terms of performance and applicability but comes with a large development, testing and maintenance cost. Our approach leverages models to generate these dedicated networking components. Figure 3 presents part of the protocol model for the CoffeeSpy device. ThingML allows modeling a set of communicating devices and a set of messages they can exchange. By default all messages are considered as asynchronous and are sent with a "send on forget" policy (see messages related to raw data in the Figure). This is the simplest message exchange strategy and it can be implemented at a very low cost on top of any kind of physical link. However, even for applications as simple as the CoffeeSpy, more complex networking features are required:

acknowledgments, synchronous message responses, error detection, messages retransmission, encryption, etc. ThingML relies on a set of predefined annotations to further refine the communication semantics of the messages.

The two messages called *ping* and *pong* can be used to check the communication between the server and the CoffeeSpy device. *Ping* can be sent to the device and the annotations specify that this message has a synchronous acknowledgment called *pong*. This acknowledgment is expected to come within 1 second, and if it does not come, 3 attempts can be made at resending the "ping" message. By processing these annotations, the code generator produces the emission of the acknowledgment message in the device code and an operation in the client API which sends the *ping* message, handles timeouts and retries and return a status information specifying whether or not the acknowledgment was received. By default the operation simply returns a Boolean but several options can be used to throw exceptions when communication failures occur. Similarly, it is possible to manage synchronous calls with a return value in a similar way as illustrated by the messages *GetTemperature* and *TemperatureValue* define a synchronous way of reading the temperature sensor of the CoffeeSpy device. The annotation *sync_response* specifies that the parameter *v* of the message *TemperatureValue* is the result of the call.

In all cases, the code generator will only include the code required to implement the networking features on the specified messages. The benefits of the approach are two-fold: First it allows for efficient, specific and compact networking components. Second, it provides a fined grained way of defining the specific ways in which different messages should be handled. For example, in a typical sensor network different types of information are exchanged between nodes for collecting sensor data, managing the network, discovering new sensor nodes, etc. Each feature has different needs in terms of synchronization, response time, bandwidth, reliability, etc. With a classical approach this would require using the most advanced protocol stack for all communications or to embed a collection of different protocol stacks. Using the proposed model-based approach, the type of communication and its quality of service properties can be fine tuned for each message in order to fulfill the domain requirement with no accidental overhead.

3.3 Generating interactive simulators

In practice, micro-controllers and client software systems are usually developed by two separate teams with different competences. For productivity reasons, it is not reasonable to wait that one end ((Java-based) software or (C-based) things) is fully operational before realizing the second end. Rather, both ends are developed and maintained concurrently, often leading to misalignments and inconsistencies. It is thus very important to provide support early in the development cycle to be able to test the different ends of a WSN-based application.

The fully automated generation of APIs to encapsulate and exchange messages (Section 3.1) is a first step to uncouple the development of advanced services from the development of things. Once messages are specified in ThingML,

different teams of developers can rely on the generated API knowing that they will interoperate seamlessly.

Similarly to the development of more classic applications, developing a WSN-based application often requires an iterative process. This is even more important since some constraints are imposed by the low-resource hardware. As motivated in Section 2, it is not always easy to identify a priori the best tradeoff between which part of the application logic should be implemented on the small device, and which part should be executed on more powerful nodes.

To support a more agile development process, we also generate interactive simulators, as illustrated in Figure 4, which respectively mock-up the devices and their logical environment, in order to respectively test the devices or other client devices/services [5].

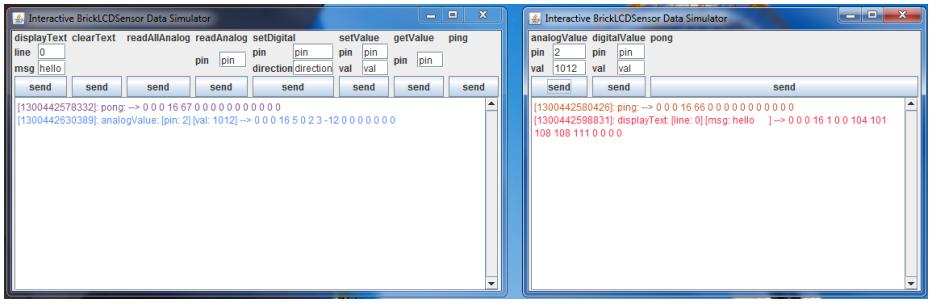

Fig. 4. Interactive simulators to enable independent development

These simulators enables rapid and early testing of the two ends of the system (C-based devices and Java-based services) to identify early in the development cycle potential lacks or mismatches in the set of messages, with no need to wait that one end is fully developed to discover these problems. In the case messages are updated in the ThingML model, API can be re-generated. This would of course imply some refactoring in the client code, facilitated by modern IDEs.

As described in Section 3, ThingML focuses on resource-constrained devices, usually deployed as leaves in a sensor network. However, it is possible to infer some useful information from the ThingML specifications of such devices. Typically, if we reverse the protocol of a given device (i.e, send message that were formerly received, as vice-versa), we can obtain a specification of a client device equivalent to the logical environment of the initial device. Another alternative that we also support is to let the designers specifying other devices (that can potentially run on more powerful nodes, even though ThingML does not specifically address such kind of nodes) that interact with the initial device. These devices partially intersect the logical environment, but might also define other messages. By default we generate an interactive simulator that stubs all the devices specified in the ThingML model, and that also stubs all the logical environment. Both stubs are actually generated using the same code generator, since

we consider the logical environment as a "mirror" device. Each stub is composed of a simple (fully generated) GUI and a controller that implements the Observer pattern generated with the API (Section 3.1). By default, we generate a simple test program which connects both stubs together. The utility of this application is simply to check that message are serialized and deserialized in a consistent way by the generated API. However, it is off course straightforward to connect the stubs to other application implementing our Observer pattern, such as the JArduino [6] application presented in Section 4.2.

4 Application

In order to validate our ThingML IDE, we have implemented two case studies in ThingML, on top of the Arduino platforms, and generated the code to actually run these case studies. Arduino [2] is an open-source (both hardware and software) electronics prototyping platform. The Arduino board can be connected to a set of sensors and actuators and programmed in a language close to C/C++.

The first application is a domain-specific application (the coffee spy) based on a precise set of sensors that we used as a running example in this paper. The second case study is a general purpose application to enable the rapid prototyping of sensor networks on top of the Arduino platform, which we have made available as an open-source project on GitHub [6]. The idea is to gather a community (basically, Java developers who wants to try the Arduino platform) around JArduino and collect feedback on the generated code.

We use the Sonar tool (http://www.sonarsource.org/) to compute various metrics and score (based on simple naming convention, anti-pattern detection, etc) on the generated Java code. The metrics of the generated C code (smaller) are provided manually.

4.1 CoffeeSpy: Domain-specific application in ThingML

The protocol of the coffee spy is described by 24 messages, in 50 lines of code. From this rather synthetic model, we generate 44 classes for a total of 2702 lines of code for the Java side, and almost 400 LoC for the C side. The expansion factor is thus more than 50. The generated code is fully operational and is of good quality: it obtains a score of 83.7% of rules compliance, using the default Sonar settings.

Based on the generated API, we implemented a simple Java client program (<100 LoC) that monitors the activity around the coffee machine only when this is relevant (i.e., when someone is approaching the machine). This way it reduces the traffic on the wireless network and also reduces the amount of data to log. In more details this program:

1. Subscribes to the motion information,
2. Subscribes to the raw data if it receives an approaching motion message,
3. Logs all the raw data,

4. Un-subscribes to the raw data if it receives a leaving motion message, with no sub-sequent approaching motion message in a given time window. Otherwise it keeps its subscription and continues logging raw data.

This simple Java program has successfully been tested with the generated mock-up device and (with no modification) with the physical device.

4.2 JArduino: wrapping the Arduino API in ThingML

The goal of this second case study is to easily integrate sensors and actuators in Java, for rapid prototyping and experimentation. To achieve this goal, we wrapped all the standard Arduino API related to Input/Output[5] in ThingML, as well as other commonly used libraries (to interact with LCD, etc). The result is naturally called JArduino [6], for Java for Arduino.

The protocol of this application is described by 40 messages and 8 enumerations to constrain the parameters of the messages, in about 100 lines of code. From this rather synthetic model, we generate 68 classes for a total of 4708 lines of code for the Java side, and close to 500 LoC for the C side embedded in the micro-controller. The expansion factor is thus more than 50. The generated code is fully operational and is of fairly good quality: it obtains a score of 67.9% of rules compliance, using the default Sonar settings. This score is mostly explained by the naming convention used in the ThingML model describing the Arduino, where several messages contain underscores. In order to keep the public part of the generated API aligned with the ThingML specification, we also generate method names, etc with these underscores. This would however be straightforward to generate code complying with Java conventions and we would reach the same score as the Coffee spy case study, but this would slightly change the alignment of the API w.r.t. its specification.

The generated C-code is uploaded on the micro-controller of the Arduino board. This code actually receives messages (arrays of bytes as described in Section 3.1) and dispatches these messages to appropriate handlers, depending on the types of the messages. These handlers simply delegate to the standard Arduino API also located on the micro-controller.

The generated Java code is an API which matches the Arduino APIs. The Arduino API related to control structures, arithmetic, etc is not mapped since this is a direct sub-set of the Java language or Java standard API. When the generated Java API is invoked, it generates a message which is sent to the Arduino board.

This simple yet fully generated case study makes it possible to write Java programs that seamlessly manage sensors and actuators connected to the Arduino board. We have successfully ported most of the standard examples provided by Arduino on JArduino, and provided and extra example which simply connects the stub that simulates the logical environment to the Arduino. It provides a GUI to easily administrate the Arduino and quickly experiment sensors and actuators. More details are available at [6].

[5] http://www.arduino.cc/en/Reference/HomePage

5 Related Work

5.1 Remote Procedure Calls for micro-controllers

Several solutions exist to enable the seamless collaboration of software components written in different languages and running on different platforms. Corba [10], RMI [12], Web Services [14], WCF [13] are well-established alternatives for distributed computing infrastructure. If these technologies differs for some non functional features [8], they provide the same benefits for building distributed applications: Independence from language and OS, Strong Data Typing, High Tune-ability and Compression. CORBA/e [11] sheds the dynamic and high-resource aspects of CORBA but retains full interoperability. Then, CORBA/e Micro Profile shrinks the footprint small enough to fit low-powered microprocessors: only tens of kilobytes. Other researches exist to support Corba on top of micro-controllers [16, 4], nevertheless these approaches always use a layer model to hide the communication medium (Bluetooth, RS232, XBee, etc). ThingML use model information to generate the most suitable communication layer for each application depending of the feature required in each of them. In particular, it allows customizing communication features for each specific message.

5.2 Abstractions over sensors and micro-controllers

Using higher level of abstraction than the C language is a common approach for building software on top of micro-controller. Then, it is common to find small Java, Processing[6] or Lua[7] implementation for lightweight environment. For example, S4A[8] is a Scratch [1] modification for Arduino, which provides a high level interface to Arduino programmers with functionalities such as interacting with a set of boards through user events. In the same trend, ThingML integrate a language inspired by state machines to define the behavior of devices [7]. In this paper we presented another sub-set of ThingML, which can be seen as an Interface Description Language used to describe the interface of sensors. ThingML describes an interface in a language-neutral way, enabling communication between piece of software written using several programming language. In that sense, ThingML can be combined to high-level micro-controller programming language to manage communication.

Google Protocol Buffer is a DSL which offers abstractions to implement protocols based on efficient serialization/deserialization of structured data [9]. The sub-set of the ThingML metamodel dedicated to protocols is aligned with Protocol Buffer. While Protocol Buffer was formerly designed to handle web protocols, we handle the communication of wireless sensor networks, where nodes communicate via ZigBee, Bluetooth, etc. We also generate the code related to the behavior of the protocols, using synchronous or asynchronous message exchange,

[6] http://processing.org/
[7] http://www.tecgraf.puc-rio.br/ maia/oil/
[8] http://seaside.citilab.eu/scratch/arduino

timeout, retries, etc, as well as code which enable independent development of the different ends of a communication protocol.

OGC's Sensor Web Enablement (SWE) framework defines a set of web service interfaces and communication protocols abstracting from the heterogeneity of sensor network communication and enabling their discovery, access, tasking, as well as eventing and alerting [3]. It is an infrastructure enabling access to sensor networks using standard protocols and API. No specific effort is made in SWE to provide the link between the web-services and the real sensor. ThingML provides interoperability at a lower level and the high-level APIs generated from ThingML can be used to bridge the gap with standards like SWE.

5.3 Networking for micro-controllers

Network library based on Ethernet, IP, ARP, TCP, UDP and HTTP can run on resource-constrained micro-controllers [15]. To provide such a compact library many features of the protocols have been stripped out. For example it does not support assembling and disassembling packets which means that the size of each messages is limited to a few hundred bytes. This is perfectly acceptable on a micro-controller which should just process a set of commands and transmit the data from a few sensors. Thanks to these limitations not only the library fits on micro-controllers but it also outperforms many computer based protocol stacks in terms of response time. Obviously such an approach is very costly in terms of development, testing, maintenance and evolutions. The more specific the protocol component is made, the more optimized it can be but the less reusable it is. ThingML by generating fully functional code to deal with communication can significantly reduces the burden of developing such libraries.

6 Conclusion and Future Work

This paper presented a Model-Driven approach to generate efficient communication APIs to exchange messages with and between resource-constrained devices. Based on a concise ThingML description of the messages sent and received by a device, we fully generate:

- Efficient API for the serialization and deserialization of messages in/from arrays of bytes, as presented in Section 3.1.
- Handlers for managing the communication features (asynchonous/synchronous messages, timeout, retry, etc) specific to each message, as presented in Section 3.2.
- Mock-ups and interactive simulators to enable independent development of things and services, as presented in Section 3.3.

We have validated our approach on two case studies. All the code related to communication has been fully generated. The CoffeeSpy application is a toy example which however relies on standard technologies used in state-of-the-practice wireless sensor networks. JArduino is a medium-sized application that we have made available as an open-source project.

In future works, we plan to extend ThingML both for a technical and a research point of views. From a technical point of view, we will implement a bi-directional bridge between ThingML and Google Protocol Buffer, and extend Protocol Buffer with our C code generator targeting low-resource devices. We will also include the feedback provided by the community on the JArduino project [6] to improve and extend our code-generators, so that all the applications generated from ThingML would benefit from these improvements. From a research point of view, we will continue to investigate how best practices in "classic" software development can be applied to micro-controllers: separation of concerns, self-adaptation [7], variability management and reuse, etc.

References

1. Mitchel Resnick amd John Maloney, Andrs Monroy-Hernndez, Natalie Rusk, Evelyn Eastmond, Karen Brennan, Amon Millner, Eric Rosenbaum, Jay Silver, Brian Silverman, and Yasmin Kafai. Scratch programming for all. *Commun. ACM*, 52(11), 2009.
2. Arduino. Arduino home page. http://www.arduino.cc/.
3. Mike Botts, George Percivall, Carl Reed, and John Davidson. Geosensor networks. chapter OGC® Sensor Web Enablement: Overview and High Level Architecture, pages 175–190. Springer-Verlag, Berlin, Heidelberg, 2008.
4. Carlo Curino, Matteo Giani, Marco Giorgetta, Ro Giusti, Amy L. Murphy, and Gian Pietro Picco. Tinylime: Bridging mobile and sensor networks through middleware. pages 61–72, 2005.
5. eviware. Feature overview - soapUI. http://www.eviware.com/soapUI/features.html.
6. Franck Fleurey and Brice Morin. JArduino Repository on GitHub. https://github.com/ffleurey/JArduino/.
7. Franck Fleurey, Brice Morin, and Arnor Solberg. A Model-Driven Approach to Develop Adaptive Firmwares. In *SEAMS11: 6th International Symposium on Software Engineering for Adaptive and Self-Managing Systems. Waikiki, Honolulu, Hawaii, USA*, 2011.
8. N. A. B. Gray. Comparison of web services, java-rmi, and corba service implementation. In *Fifth Australasian Workshop on Software and System Architectures*, 2004.
9. Google Inc. Protocol buffers. http://code.google.com/apis/protocolbuffers/.
10. OMG. Corba / iiop specification 3.1., January 2008.
11. OMG. Corba for embedded (corbae) specification 1.0. formal/2008-11-06, November 2008.
12. ORACLE. *Java Remote Method Invocation Specification, JDK 1.5*. Oracle, Mountain View, Calif., October 2004.
13. Steve Resnick, Richard Crane, and Chris Bowen. *Essential windows communication foundation: for .net framework 3.5*. Addison-Wesley Professional, first edition, 2008.
14. James Snell, Doug Tidwell, and Pavel Kulchenko. *Programming Web services with SOAP*. O'Reilly & Associates, Inc., Sebastopol, CA, USA, 2002.
15. tuxgraphics.org. HTTP/TCP with an atmega88 microcontroller (AVR web server). http://www.tuxgraphics.org/electronics/200611/embedded-webserver.shtml,.
16. F. J. Villanueva, D. Villa, F. Moya, J. Barba, F. Rincón, and J. C. López. Lightweight middleware for seamless HW-SW interoperability, with application to wireless sensor networks. In *DATE07: Conference on Design, automation and test in Europe*, pages 1042–1047, San Jose, CA, USA, 2007. EDA Consortium.