Deliverable reference:	Date:	Version:	Responsible partner:
D2.1	09.11.2017	V1.0	SINTEF

DiversIoT

Project co-funded by the Norwegian Research Council
IKTPluss

http://its-wiki.no/wiki/DiversIoT:Home

Title:

D2.1 Proof of Concept case study report

Editor(s):
Brice Morin
Jakob Høgenes

Approved by:
Project Technical Manager: Franck Fleury

Classification:
Unrestricted

Abstract / Executive summary:

The goal of DiversIoT is to enable automatic diversification of IoT software distributed across various IoT nodes: devices, gateways, servers, etc. We propose to achieve this by building on and extending ThingML, a modelling language and toolset for the IoT.

This report gives a brief introduction to ThingML, a baseline technology for DiversIoT, and then details the experiments we have conducted in DiversIoT during Phase 1.

Document URL:

ISBN:

The Research Council of Norway
With funding from
The Research Council of Norway

Content

Table of Figures

Version History

Version	Description	Date	Who
0.1	First complete version		Brice Morin, Jakob Høgenes

1 ThingML, an enabler for IoT Software Diversification

ThingML is a baseline tool that is brought to and will be further developed in the DiversIoT project. All partners in DiversIoT have experience with ThingML:

- SINTEF has developed ThingML over the past 10 years, through a set of European, national and SINTEF internal projects.
- UiO uses ThingML in courses related to Model-Driven Software Engineering and IoT.
- Tellu IoT has recently acquired ThingML, as a mean to reinforce its IoT strategy and business development. Tellu IoT intensively used ThingML in the context of FP7 HEADS project, together with SINTEF.

It is important to note that ThingML is open-source (Apache 2.0 license) and freely available on GitHub[1] and that the contributions of DiversIoT to ThingML are and will also be open-source.

ThingML is based on a combination of well-proven software modelling constructs aligned with the UML (state charts and components), an imperative platform-independent action language and specific constructs targeted at IoT applications. ThingML is supported by a set of tools which include editors, transformations (for example to export UML models), an advanced multi-platform code generation framework which support multiple target programming languages. The figure below shows the ThingML editor integrated into the Eclipse IDE as well as a UML state chart diagrams automatically generated.

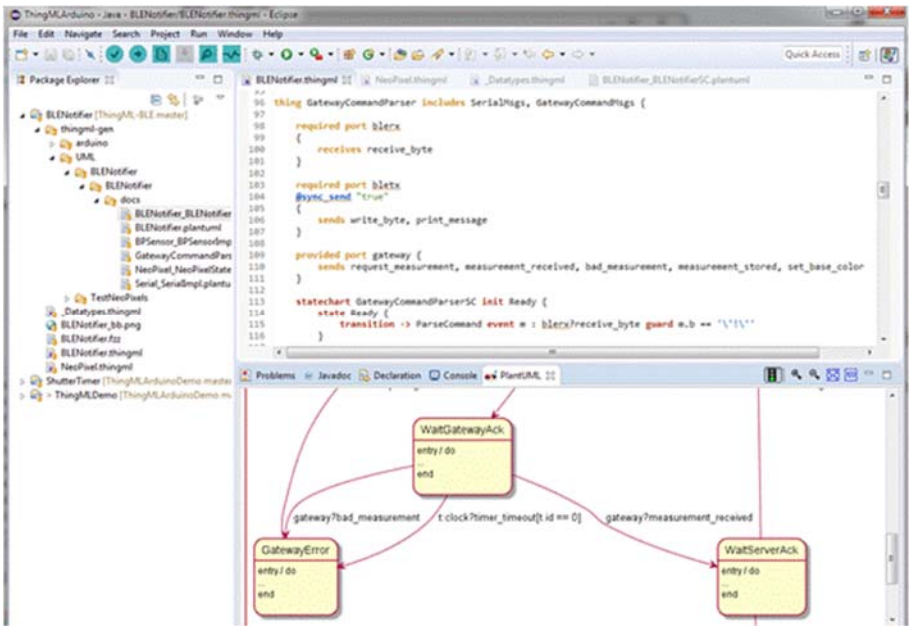
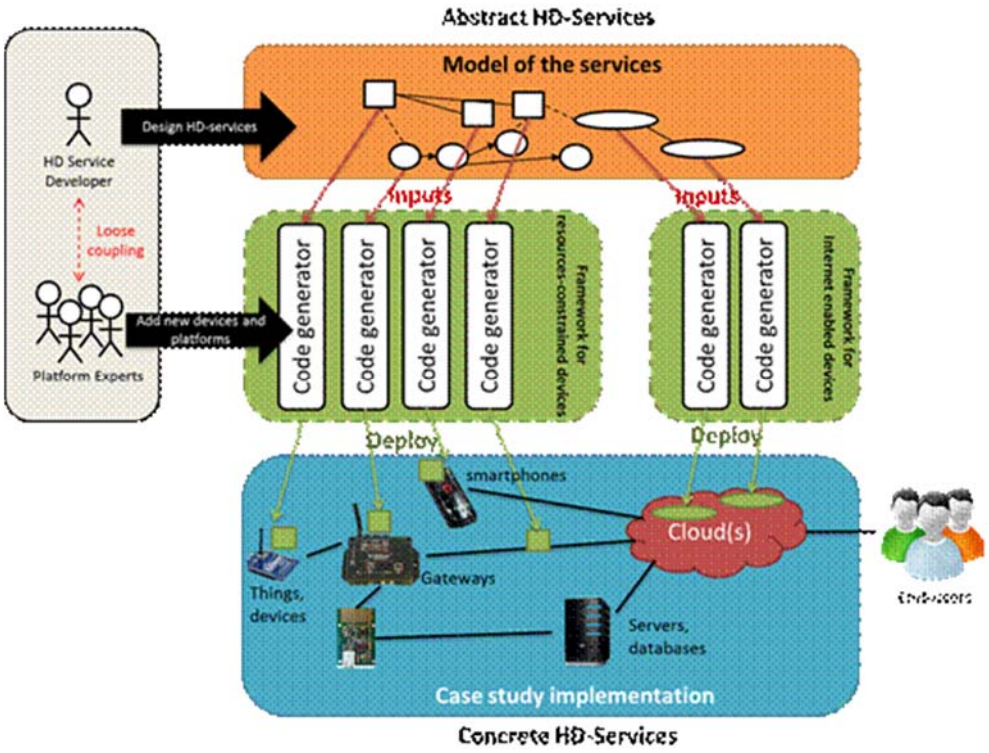


Figure 1: ThingML IDE integration in Eclipse

[1] https://github.com/TelluIoT/ThingML

A key feature that makes ThingML particularly relevant for DiversIoT is its code generation framework. Rather than targeting a specific runtime platform, this code generation framework provides a set of extension points that can be customized to target different languages, platforms, communication protocols, etc. The figure below gives a high-level overview of this code generation framework. Basically, different software components composing the whole IoT system can be generated by different compilers.



Each compiler (or code generator in the figure above) is actually not monolithic, but rather a composition of a set of extension points, as illustrated in the figure below. The ThingML code generation framework empowers the developers to tailor and extend the code generators in order to fit the need of their domain and projects. The ThingML code generation framework is composed of an abstract framework and a set of ready-made code generators for three different languages (C/C++, Java and JavaScript (both server and browser)) and a number of libraries and open platforms (Arduino, Raspberry Pi, Intel Edison, Linux, etc.). Within a single IoT application, ThingML components are distributed on different heterogeneous platforms for which different code generators are used. Specific code generator plugins allow generating the communication and message exchange between components running on different platforms.

To allow customizing any part of the generated code in an efficient way, the framework is organized around 11 extension points, each responsible for the generation of a particular aspect of the source code[2]. For example, one is dedicated to the generation of public APIs while another

[2] ThingML, A Language and Code Generation Framework for Heterogeneous Targets -- N. Harrand, F. Fleurey, B. Morin and K.E. Husa. -- MODELS'16: ACM/IEEE 19th International Conference on Model Driven Engineering Languages and Systems. Practice and Innovation track

is dedicated to the generating of the implementation of component. In an IoT system, each component can use a different combination of code generation plugins.

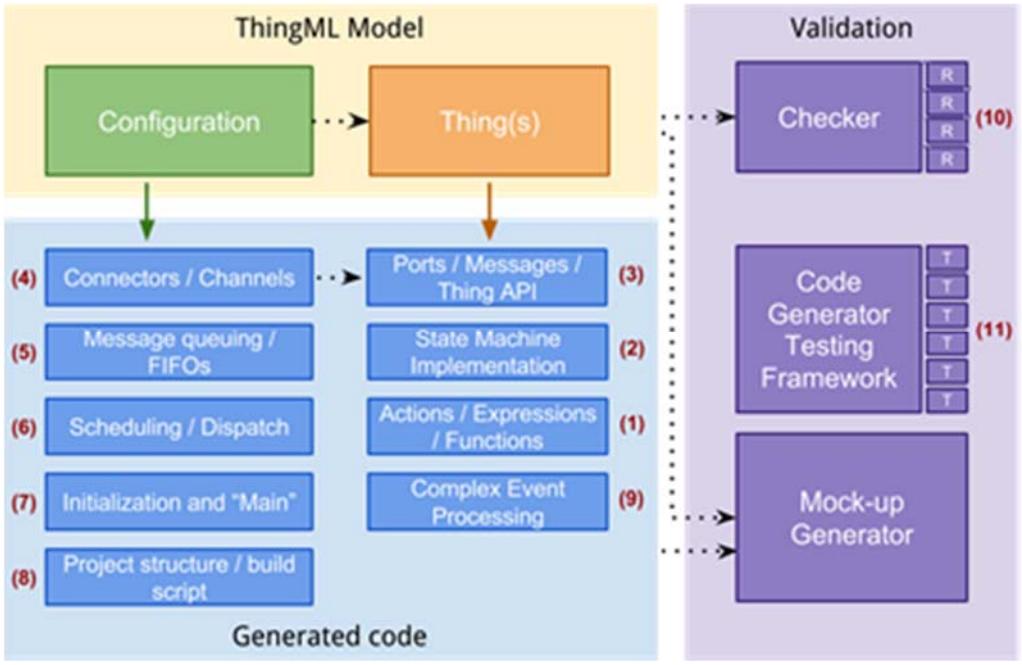


Figure 2: Overview of the ThingML code generation framework, with extension points (numbered)

ThingML, and its modular code generation framework, provide a solid baseline to apprehend the diversity of the IoT. Though ThingML as of today can produce diversified implementations from a platform-independent specification, for example an implementation in Java, another in JavaScript and another in C, the choice of which compiler and which concrete extension points to use is left to the developer.

In DiversIoT, we have implemented a lightweight extension to ThingML during Phase 1 in order to more automatically drive the diversification process. This extension is presented in the remainder of this document.

2 Experimental Setup

The experimental setup, depicted in the figure below, mirrors a typical IoT deployment, with a set of devices communicating with gateways (in this experiment, one gateway per device, but the experiment can easily scale to n devices per gateway), themselves communicating with a cloud backend. This is for example a setup use by Tellu when providing and operating eHealth services.

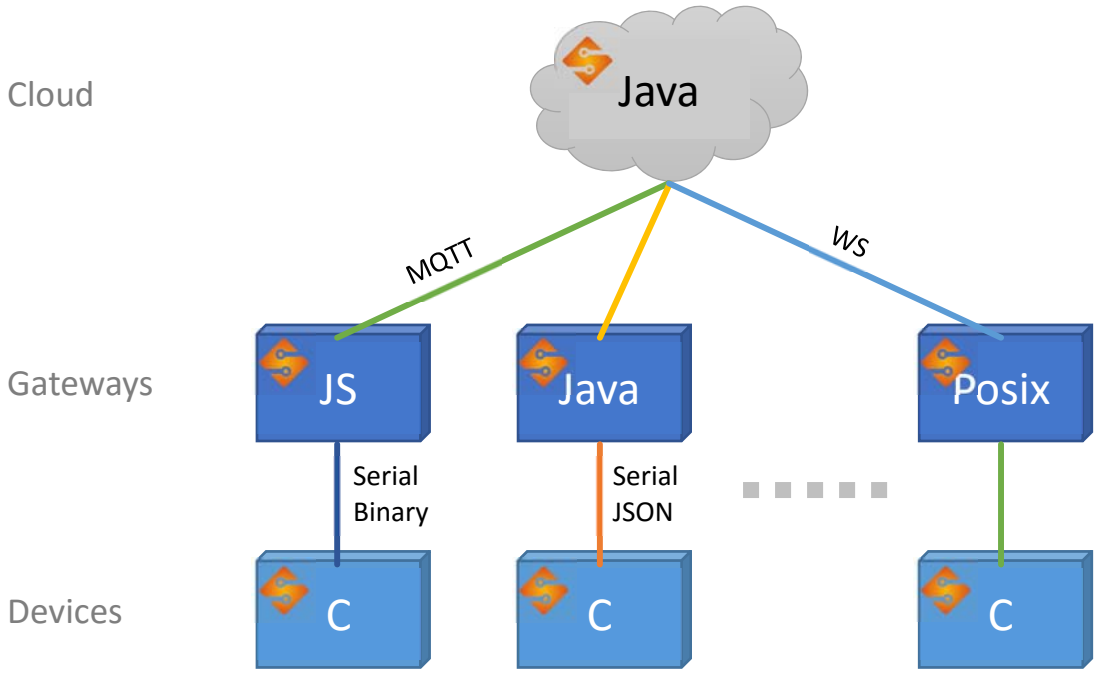


Figure 3: eHealth case study component overview

In this experiment, devices run C code, as devices are typically too constrained to run advanced programming languages such as Java or Javascript. The gateways run either C (Posix), JavaScript (Node.JS) or Java code as modern gateways are typically powerful enough (about 1GHz CPU with 1-2 GB RAM and fully-fledged Linux OS). The cloud backend is a mock-up implemented in Java, similarly to the Tellu cloud backend, but could in principle be implemented in any language, as the cloud provide virtually unlimited resources. The gateway thus represents a first point of diversity, as they are implemented in different languages.

2.1 Inlined diversification

We implemented a lightweight mechanism on top of ThingML to enable some parts of the ThingML compiler to generate different versions of target platform code so that:

1. different versions of the same software component are semantically equivalent as seen from the outside, while the internal implementation varies
2. the communication protocol between pairs of software components differs, but the implementation of each component supports the protocol of the other component in its pair.

The experimental platform is based upon the concept of *diversification points* (similar to variation points in software product lines, and to some extent to #ifdef macro in C), which are written as comments in the ThingML code to stay within the allowed ThingML syntax. A line containing

```
//DIVERSIFICATION <ID> “<Name>”
```

defines the start of a diversification point. The following line defines the first possible ThingML code for that point, while subsequent lines starting with

```
//DIVERSIFICATION
```

defines alternative lines of code for the diversification point.

The following piece of ThingML code defines two diversification points: (1) “Request message” and (2) “Response message”, each having two alternatives. (1) is used two places within the code, which means that the different alternatives will be used together.

```
3  thing fragment Messages {
4      //DIVERSIFICATION 1 "Request message"
5      message Request(Num : Byte)
6      //DIVERSIFICATION message Request(Num : Int)
7      //DIVERSIFICATION 1
8          @code "1"
9      //DIVERSIFICATION @code "10"
10
11     message Response(Num : Byte)
12     //DIVERSIFICATION 2 "Response message"
13         @code "2"
14     //DIVERSIFICATION @code "17"
15 }
```

For this ThingML code, the experimental platform will produce 4 (2*2) diversified versions of the code:

```
3 thing fragment Messages {
4     message Request(Num : Byte)
5         @code "1"
6
7     message Response(Num : Byte)
8         @code "2"
9 }
```

```
3 thing fragment Messages {
4     message Request(Num : Byte)
5         @code "1"
6
7     message Response(Num : Byte)
8         @code "17"
9 }
```

```
3 thing fragment Messages {
4     message Request(Num : Int)
5         @code "10"
6
7     message Response(Num : Byte)
8         @code "2"
9 }
```

```
3 thing fragment Messages {
4     message Request(Num : Int)
5         @code "10"
6
7     message Response(Num : Byte)
8         @code "17"
9 }
```

Compiling these four pieces of ThingML code will result in four different versions of platform code, which all defines the two messages “Request” and “Response”.

On the gateways, those 4 versions of this simple request/response protocol can be generated for 3 different languages: C, Java and Node.JS, yielding 4*3 = 12 possible concrete implementations.

In general, this inlining strategy allows developer to define alternative, yet semantically equivalent, behaviours for the different components. Some strategies are discussed in the surveys also delivered in Phase 1.

2.2 Generation of executables

Since ThingML code is platform independent, the experimental platform makes use of another special comment to specify to which target platform to generate code:

```
//LANGUAGE <ThingML-compiler-id>
```

This comment can also be placed within a diversification point to generate executables to multiple platforms.

Along with the diversified versions of the ThingML code, the platform will then generate *Dockerfiles* that describe how to build Docker containers that contain a fully built executable for the specified target platform. E.g.:

```
1 FROM diverseiot/base
2
3 COPY medical-device.thingml /root/thingml/
4 RUN java -jar /root/thingmlcli.jar --compiler <COMPILER> && \
5       --source /root/thingml/medical-device.thingml      && \
6       --output /root/target/
7 RUN <COMMAND COMPILE THINGML TO PLATFORM CODE>
8 RUN <COMMAND TO COMPILE PLATFORM CODE>
9 ENV THINGML_CMD="<COMMAND TO RUN COMPILED CODE>"
```

The parts within angle brackets <...> are automatically generated by the experimental platform.

2.3 Composing software components

For the different components to actually work together, we use readily available network plugins provided by ThingML:

- Device to gateway communication is done over a simulated serial port using using two different serializations of three messages, yielding 2*2*2 = 8 variants
- Gateway to cloud communication is done over MQTT or WebSocket, using two variants of two messages, serialized with either JSON or binary formats, yielding 2*4*2 = 16 variants

In the experimental setup, a single cloud component communicates with all the generated gateway components, while the device and gateway components communicate in pairs. This means that the experimental platform has to ensure that the components that send messages to each other use the same serialization of each message. For the cloud component (that is not diversified), the experimental platform generates handling of all possible serializations. The gateway and device components are generated in pairs that use the same serialization variant. This is performed by selecting the same option for each diversification point that is shared between the pairs of gateway-device code that is generated.

Alongside the generated cloud component and pairs of gateway-device components, the experimental platform also generates a *Docker Compose* configuration file that links the correct Docker containers generated for each component together at runtime. The following excerpt from a generated configuration file (a "Docker stack") shows the configuration for the cloud component and a single pair of gateway-device components:

```
1version: '3'
2services:
3  cloud:
4    image: diverseiot/cloud:static
5    build:
6      context: diversified
7      dockerfile: Dockerfile.medical-cloud
8    command: bash -c "$$THINGML_CMD"
9  gateway_0-0-0:
10   image: diverseiot/gateway:0-0-0
11   build:
12     context: diversified\0-0-0
13     dockerfile: Dockerfile.medical-gateway
14   command: bash -c "$$THINGML_CMD"
15   links:
16    - cloud
17    - device_0-0-0:device
18 device_0-0-0:
19   image: diverseiot/device:0-0-0
20   build:
21     context: diversified\0-0-0
22     dockerfile: Dockerfile.medical-device
23   command: bash -c "$$THINGML_CMD"
24   links:
25    - gateway_0-0-0:gateway
```

The actual deployment has been done on SINTEF's private experimental cloud, composed of 10 PCs, accounting for 72 cores, 400 GB RAM, 4.5 TB SSD storage (including 1.3TB in NVMe), 50 TB HDD, communicating through a Gigabit network and set up as a Docker Swarm.

3 Summary and Conclusion

In this experiment, we have automatically derived 8 implementations for the devices, which uses a diverse set of advertise/request/response messages. The gateways further implement 4 versions of the request/response messages to the cloud, uses either JSON or binary serialization over MQTT or WebSocket, and supports 3 different languages (C, Node.JS or Java), yielding 4*2*2*3*8 = 384 possible concrete implementations for the gateways.

Altogether, 384 possible working combinations of device/gateway can automatically be obtained and successfully push the cloud backend, by defining no more than 8 diversification points with 2-3 alternative implementations.

To facilitate the actual deployment of those generated artefacts, each artefact (e.g. the software for a device or the software for a gateway) is wrapped into a Docker container, and each concrete deployment is specified as a Docker stack, which can then fully automate the deployment on SINTEF's experimental cloud, or on any cloud setup as a Docker Swarm (e.g. Google's GKE, Amazon AWS Docker Containers or Docker Cloud).

This overall setup, based on generative techniques to produce code and container technologies to package it and deploy it, promotes repeatability and allows setting up further experiments rather easily. This experimental framework will be a major asset for Phase 2.