Final version available at IEEEXplore : http://dx.doi.org/10.1109/ICSAW.2017.63

ThingML: A Generative Approach to Engineer Heterogeneous and Distributed Systems

Franck Fleurey, Brice morin
SINTEF Digital
Email: first.last@sintef.no

I. Topic

A. Summary

Cyber Physical Systems (CPS) typically rely on a highly heterogeneous interconnection of platforms and devices offering a diversity of complementary capabilities: from cloud server with their virtually unlimited resources to tiny micro-controllers supporting the connection to the physical world. This tutorial presents ThingML, a tool-supported Model-Driven Software Engineering (MDSE) approach targeting the heterogeneity and distribution challenges associated with the development of CPS. ThingML is based on a domain specific modelling languages integrating state-of-the-art concepts for modeling distributed systems, and comes with a set of compilers targeting a large set of platforms and communication protocols. ThingML has been iteratively elaborated over the past years based on a set of experiences and projects aiming at applying the state of the art in MDSE in practical contexts and with different industry partners.

B. Goals and Take-away messages

The goals of the tutorial are to:

1) reflect on these previous experiences to motivate the current ThingML approach and its implementation,
2) describe the ThingML approach and its usage by the actors involved in the development of CPS, and
3) provide hands-on experience with the associated tools.

The take-away messages the audience should leave with:

1) ThingML: UML components and state machines made practical
2) ThingML: Generative techniques that just work (compiles to C, JS, Java)

C. Intended audience

The tutorial is targeted both at researchers and practitioners. Average programming (*e.g.* Java, JavaScript or C) and modelling skills (state machines, components) are required. Participation to the hands-on exercises requires a laptop with Oracle Java JDK 8 installed and capable of running Eclipse (Windows, Linux and Mac are supported). All the tools, samples and exercises used during the tutorial are open-source and will be distributed through USB sticks and installed during the tutorial. The installation is rather lightweight: simply execute Eclipse from the USB stick or copy Eclipse to the PC and execute Eclipse from the PC.

Ideally, participants can visit *https://github.com/SINTEF-9012/ThingML* and **read the README file so as to get started before the tutorial.**

II. Implementation

A. Length

The tutorial is the standard 1/2 day.

B. Level

Beginner / Introduction, with possibilities to approach more advanced topics depending on requests from the audience.

C. Content and Outline

This tutorial targets the development of software systems for which the software has to be distributed within an infrastructure composed of a set of heterogeneous interconnected computational nodes. Most Cyber Physical Systems (CPS) and Internet of Things (IoT) applications fall under this category. In the following we only refer to CPS as the target systems but (i) the approach applies more generally to any software system with similar requirement in terms of heterogeneity and distribution (ii) the sub-set of CPS which can be implemented with a centralized approach and/or a single platform is not within scope.

The next sections follow the outline of the tutorial and provide indicative timing for each part of the tutorial. Questions and remarks from the audience will be handled as they come during the sessions.

1) SE Challenges in CPS development [15 min]: The typical infrastructure of a CPS is composed by a highly heterogeneous interconnection of platforms and devices offering a wide diversity of capabilities. On the one end of the continuum, cloud platforms provide virtually unlimited and "on-demand" resources in terms of computation power, storage and bandwidth. On the other end, the already vast and rapidly increasing number of smart objects, sensors, embedded systems and mobile devices connected to the Internet offers the connection to the users and to the physical world. While offering great potential for innovative CPS, the heterogeneity, diversity and vast distribution of the computing continuum represent daunting challenges.

Current software engineering approaches tend to provide support for managing and exploiting only parts of the continuum. For example, current cloud computing and service-oriented software engineering practices provide efficient abstractions for virtualizing the infrastructure in order for the

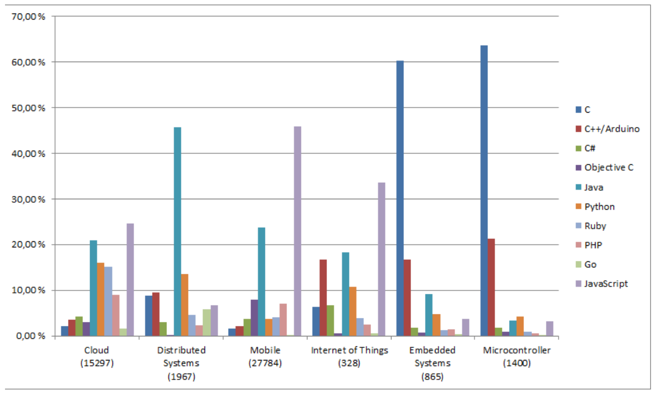

Fig. 1. Popularity of 10 programming languages for the different areas related to CPS development

software engineer to concentrate on the business logic of the applications. However these techniques merely support integration of mobile devices, sensors and actuators "as-a-service" and lack specific support for advanced exploitation of these small devices.

Furthermore, although sensor network nodes, gateways, smart-phones, and most smart-objects are provided with APIs and development environment which allow advanced exploitation of their capabilities. There are no common practices to expose the level of flexibility, languages, APIs and customization supported by different devices and platforms at the level of what is provided in cloud and service oriented approaches.

Software for the different classes of devices are typically built using different approaches and languages. In order to understand the skills and capabilities required to develop a CPS on top of such an infrastructure, we queried a popular open-source repository (GitHub) to evaluate the heterogeneity of programming languages across the continuum. The following sets of keywords were used: 1) Cloud: server with virtually unlimited resources, 2) Microcontroller: resource constrained node (few KB RAM, few MHz), 3) Mobile: an intermediate node, typically a smartphone, 4) Internet of Things: Internet-enabled devices, 5) Distributed systems, as services exploiting CPS have to be distributed across the continuum, and 6) Embedded systems, as a large and important part of the CPS implementations will run as close as possible to physical world, embedded into sensors, devices and gateways.

Figure 1 presents the results of those queries. The queried keywords are presented on the x axis together with the number of matches for that keyword. For each keyword, the y axis represents the popularity (in percent of the total number of matches) of each of the 10 most popular programming languages that we encountered.

This simple study indicates that no programming language is popular across the whole continuum. A general trend indicates that Java and JavaScript (and to some extent, Python and Ruby) are popular in the higher-end of the continuum (cloud and mobile), whereas C (and to some extent, C++) is a clear choice for developers targeting embedded and microcontroller-based systems. Other languages do not score more 10% for any of the keywords. For all keywords except “Cloud”, the combined popularity of Java, JavaScript and C/C++ (*i.e*, the sum of the percentages) is above 70%. For “Cloud” we observe a certain homogeneity with Python, Ruby also being very popular, so the combined popularity of Java, JS and C/C++ is only 50%. It is also worth noticing that the most popular language for a given keyword scores very poorly (less than 5%) for at least another keyword.

2) *State of the Art / Practice [15 min]:* The following subsections review a set of typical software engineering practices used to manage (or avoid) the heterogeneity problem when developing CPS.

a) Centralized system which uses devices "as-is": This is a common approach for simple systems for which all the logic can be implemented in a central program. Most simple Internet of Things applications rely on dumb things, devices and sensors which constantly communicate to a central unit running the application. The central unit can be a server in the cloud, a home automation gateway or in some cases a mobile phone. This centralized design makes development, deployment and maintenance easy but it is also restrictive in term of applications functionalities, quality of service and scalability. This approach is not suited for most CPS.

b) Avoid problems by carefully selecting platforms: When designing an application and its infrastructure from scratch, the designer can select a set of platforms which are "compatible" and for which an integrated software framework or middleware already exists. For example, when building sensor networks, a set of motes, gateways and back-end servers can be selected based on software capabilities which are compatible with each other’s. Another possibility is selecting popular generic platforms (for example Arduino) for which large set of libraries and software components are already available and build the application around those components. This approach is only suited when building a CPS from the ground up and restricts the choice of platform which can be used.

c) Hide behind an homogeneous software layer: Deploy a common runtime platform or middleware to all different devices in order the then easily deploy software components to any node of the system. This solution, for example advocated by Java’s "*write once, run everywhere*" motto, consists in hiding the heterogeneity behind a virtual machine. When applicable, this is a good approach to eliminate the accidental heterogeneity between platforms. In the case of CPS the range of platforms is too wide to be fully covered by a single language or middleware. In addition only the common subset of platforms features can be exposed and some runtime overhead is inevitable. The greater the range of covered platforms is stretched, the less features are can be exposed and the higher the overhead. This approach also restricts the choice of platforms which can be used.

d) Custom develop manually all pieces of software: In some domain (automotive, aeronautics, etc) where systems of mixed criticalities have to be implemented, the heterogeneity cannot be avoided or hidden. Some critical part of the software has to run in real-time on some specific hardware (typically resource-constrained hardware, approved to run in harsh conditions), while some less critical software (e.g. related to entertainment) can run on more permissive and less constrained hardware. In this case, the only solution is to have a large team of developers, each specialized in a given platform and using platform specific techniques. This results in high development costs [**?**] (the cost of software in a car ranges from 15 to 45% of its total cost) which can only be acceptable in specific situations (e.g. critical systems, mass production). This is the only solution which currently allows exploiting every piece of the infrastructure of a CPS to its full potential.

None of the solutions presented above allow exploiting the CPS continuum platforms to its full potential.

3) MDE: Promises, Strengths and Weaknesses [15 min]: The model based software engineering community has proposed the Model-Driven Architecture (MDA) (and a number of related techniques) for abstracting from heterogeneous platforms. The idea of those approaches is to describe the logic of the system in platform independent models which can then be transformed into different platform specific executable code. The benefit of those approaches is to allow producing sets of implementation fitting different target platform. Another advantage is that it does not impose a runtime overhead and can allow for the generation of optimized native code. While some of these approaches have been adopted in specific domains (e.g. telecommunications or critical embedded systems) their practical adoption remains limited. Some of the main reasons are that systematically describing behaviour in a platform independent way introduce an important accidental overhead, the difficulty of creating and maintaining high quality code generators, the difficulty of integrating with legacy and other components developed using other approaches. Another barrier to the wide adoption of model-driven development is the high complexity of the UML (both syntactic and semantic) which typically limits its practical use beyond the design phase.

4) ThingML Approach [30 min]: The objective of the ThingML approach is not to compete or replace any of existing languages, practices or frameworks which are used on individual target platforms. The idea is to provide a way to practically combine software components using these different heterogeneous techniques and platforms to build a complete CPS. The ambition of the approach is to support the complete life-cycle of the software components, from design time to implementation and evolutions.

The first phase in the realization of a CPS is the design phase where the main components of the system are defined and the infrastructure on which these components will run needs to be selected. In practice, as for any realistic systems, some of the system components and some of the infrastructure will be based on of-the-shelf and legacy systems. This is especially true for CPS which typically result from a combination of pre-existing hardware and software. For the design and modelling of CPS, the software engineering literature and practices include some well adopted abstractions. The ThingML approach integrates popular abstractions, which have been adopted across the range of CPS platforms, into the ThingML language. The purpose of this domain-specific modelling language is to capture the architecture of the CPS and the communication required between the different components. In terms of modelling components, the ThingML language provides a flexible solution either to model the complete behaviour of components, or integrate off-the-shelf or legacy components as black boxes.

The ThingML language includes:

- **Component types with ports and asynchronous messaging**: this is the base for the approach; all parts of the system have to be described as components with asynchronous messaging interface. This is a well adopted model for all kinds of distributed system.
- **Event-based reactive programming**: the behaviour of components can be expressed using event processing rules. Our current implementation is limited to event-condition-action rules but planned extensions will include complex event processing constructs allowing combining event or constructing complex event from the events occurring within a certain timeframe.
- **Composite State Machines**: the behaviour of components can be structured in a state machine. The state machines in the ThingML languages are aligned with UML2 state charts and include composite states, regions and history states.
- **Action language**: An imperative action language and expressions allow fully modelling all conditions and actions within event processing rules and within state machine in a platform independent way. This action and expression language also includes a template language for easily embedding or linking platform specific code.

The rationale for creating a new domain-specific modelling language for the ThingML language is (*i*) no existing modelling language provide the exact set of concepts needed (for example UML 2.x contains numerous additional concepts but still lacks a few, in particular a practical platform independent action language) and (*ii*) in order to allow supporting all subsequent phases of the CPS life-cycle with practical tools based on the same concepts and the same well defined semantics. In order to ease the adoption of the ThingML language, all concepts and their semantics are aligned to a subset of the UML 2.x standard. One key tooling difference with the UML is that the ThingML language primary concrete syntax is lexical and not graphical.

5) Break: It will be a break during the tutorial, aligned with the official schedule of ICSA. The presenters might slightly adapt the schedule to fit the official ICSA schedule.

6) Tooling and Methodology [25 min]: The ambition of the ThingML language is to support the implementation and integration of the different parts of the CPS, including the integration of legacy and of-the-shelf components. The ThingML approach has a cost that comes as a consequence of this flexibility. At a certain abstraction level, all components interfaces need to be described in terms of the ThingML language in order to allow for their integration in the system. However, it is important for any existing platform, library, framework or middleware to be usable without re-inventing, re-modelling or re-implementing it. In the design of the ThingML approach and code generation framework, a special attention is put to avoid introducing any accidental overhead beyond what is strictly necessary for the integration of the implementation artefacts.

For the components developed from scratch or for which the target runtime environment might change, the ThingML language provides with all the required expressiveness to fully specify the behaviour of a component in a platform independent way. Different code generators can then be used to produce code for different platforms (currently Java, Javascript and C/C++). This is similar to typical MDE platform independent models with a set of code generators (or compilers) for different platforms. For the integration of an existing component, the ThingML approach allows modelling only the required part of interface of the component and mapping to its public platform specific API. This is similar to typical wrapping of external components and libraries such as for example Java Native Interface for Java to interact with a native library.

The contribution of the ThingML approach is to give the flexibility to develop components which are neither fully platform independent nor direct wrapping around existing components, but rather an arbitrary combination of exiting libraries, platform features and application logic. In practice, most of the components of a CPS fall under this category and being able to efficiently integrate these different elements is a key goal. The way the ThingML approach implements such capabilities is two-fold. First, a set of special constructs and actions are included in the ThingML action languages in order to seamlessly interleave platforms specific code and platform independent code. Second, the ThingML approach relies on a highly customizable framework code generation framework which can be tailored to specific target languages, middleware, operating systems, libraries and even build systems. A total of 8 formal extension points have been identified in the ThingML code generation framework in order to allow the developer to easily and efficiently customize parts of the code generation process.

7) Exercise 1: ThingML for CPS Developers [30 min]: The CPS developer is using ThingML to create the models of the CPS and can create and/or use some re-usable ThingML libraries. Most of the contributions of the CPS developer have to remain at the platform independent level. This first hands-on exercise demonstrates the usage of the ThingML language and the possibility to generate executable code for different platforms.

CPS developers are the primary users of the ThingML approach. The CPS developer has to interact with a number of platform experts in order to select (or learn about) the platforms which will be used in the infrastructure. In the event that some of the some platforms are not supported by the ThingML code generation framework, the platform experts needs to create new plugins for the in order to support and expose the capabilities of the required platforms.

8) Exercise 2: ThingML for Platform Expert [30 min]: The platform experts are responsible for creating new libraries and code generation plugins for ThingML. These libraries allow capturing the expert knowledge in order to support new platform and their specific capabilities and to make them available to CPS developers. The platform expert can use a number of different extension points in order to support both the design time and the runtime support of a given platform. This second hands-on session will illustrate these different extension points on concrete examples.

III. Presenters' Backgroud

Dr. Franck FLEUREY is a senior research scientist at SINTEF Digital in Oslo, Norway. He received a PhD degree in Computer Science from the University of Rennes 1 (France) in 2006. One contribution of his thesis was the Kermeta meta-modelling language and framework. His research interests include model-driven software engineering, embedded systems, product lines, adaptive systems and software validation. He has been active in the software modeling community for more than 10 years with a focus on developing and using MDE approaches in both academic and industry-driven projects. In particular, he is the initiator and one of the main contributor of ThingML. He is the author of more than 90 peer-reviewed publications totalizing over 4000 citations. He has given tutorials about Kermeta and ThingML in several reputed venues including ICSE and MODELS conferences.

Dr. Brice MORIN is a research scientist at SINTEF Digital in Oslo, Norway. He holds a PhD degree iin Computer Science from the University of Rennes, France. His research focuses on investigating sound and practical modelling foundations for software systems (ranging from the cloud to the Internet of Things), available throughout their lifecycles. Together with his colleagues, he pioneered the models@runtime paradigm in the context of EU project DiVA. He is one of the main contributor of ThingML. Since 2007, Brice published more than 60 peer-reviewed papers totalizing more than 1600 citations. Together with Franck Fleurey, he has given a ThingML tutorial at MODELS 2015.

The presenters are the two core developers of the ThingML approach and its associated open-source toolset developed in the context of the HEADS EU FP7 research project.

IV. Tutorial's Background

Some of the material of the tutorial has been presented as journal papers, conference papers and university lectures.

A previous version of this tutorial has been presented at ACM/IEEE MODELS 2015 conference to 30 attendants.

V. Acknowledgements

The research leading to these results has received funding from the European Union Seventh Framework Programme (FP7) under grant agreement no 611337, the HEADS project (www.heads-project.eu).