Final version available at IEEEXplore : http://dx.doi.org/10.1109/MS.2017.11

Focus: Software Engineering for the Internet of Things

Model-Based Software Engineering to Tame the Internet-of-Things Jungle

Brice Morin, Nicolas Harrand, and Franck Fleurey, SINTEF Information and Communication Technology

The ThingML approach, which was inspired by UML, addresses the challenges of distribution and heterogeneity in the Internet of Things. This model-driven, generative approach has been continuously evolved and applied to cases in different domains, including a commercial e-health solution.

The challenge of writing, deploying, and evolving software for the Internet of Things (IoT) is often underestimated.[1] From a software engineering viewpoint, IoT applications have two main characteristics. The first is distribution over a large range of processing nodes. The second is high heterogeneity of the processing nodes and the protocols used between them. A rich collection of software engineering literature exists for each characteristic. However, not much research has addressed the combination of the two, which presents three specific challenges.

First, in large IoT applications, distribution occurs across many heterogeneous nodes, which play different complementary roles to process data close to the sources and respond in a decentralized way. Most research in distributed systems, telecommunications, or even sensor networks has focused on scaling to very high numbers of nodes but has dealt with only a few types and roles among those nodes (for example, client-server, peer tracker, sensor gateway server, and so on).[2] This homogeneity keeps the software complexity manageable because it allows developing and deploying similar code on large sets of nodes. But this isn't the case with IoT software, which must scale in not only the number of nodes but also the number of types of nodes and platforms, from microcontrollers up to the cloud.

Second, for IoT systems to deliver their full potential, developers must leverage the diverse resources and decentralized computing power that heterogeneous nodes provide. Many tools and techniques hide this heterogeneity to enable developers to write applications that execute the same way on different platforms (OSs, web browsers, mobile devices, and so on).[3] In IoT applications, part of the heterogeneity isn't accidental and must remain explicit and exploitable by developers. This way, developers can, for example, take advantage of IoT nodes' deep-sleep modes, which typically differ among platforms, to save power.

Third, in the IoT vision, applications are no longer isolated, proprietary silos of devices and software.[4] Rather, they must combine things readily available in the environment: some generic, some application-specific, and some legacy things. So, IoT applications are meant to run on shared hardware, and developers should pay special attention to avoiding unwanted interactions between applications.

Consequently, industrializing IoT applications proves challenging at all development phases and requires teams of developers with a broad range of competences from electronics and microcontrollers to cloud and data mining. The ThingML (Internet-of-Things Modeling Language) approach aims to facilitate collaboration between service developers and platform experts so that together they can produce value-added IoT-based services. Over the past six years, this approach has been continuously evolved and applied to cases in different domains, including a commercial e-health application.

The ThingML Approach

Our approach is the result of an iterative process with industry partners and practitioners (see the sidebar), in which we have improved it on the basis of the collected feedback.

The ThingML approach comprises a modeling language, a set of tools, and a methodology. The language (ThingML) combines well-proven software-modeling constructs aligned with UML (statecharts and components), an imperative platform-independent action language, and constructs targeted at IoT applications. The tools include editors, transformations (for example, exporting to UML), and an advanced multiplatform code generation framework[5] that support multiple target programming languages. The methodology documents the development processes and tools used by both IoT service developers and platform experts. The language, tools, and methodologies are released as part of the open source HEADS IDE (heads-project.eu).

Modeling IoT software

The ThingML approach's first goal is to allow abstracting from the heterogeneous platforms and IoT devices to model the desired IoT system's architecture. In practice, platforms and devices, as well as the final distribution of software components, typically aren't known during the early design phases. The architecture model consists of components, ports, connectors, and asynchronous messages.

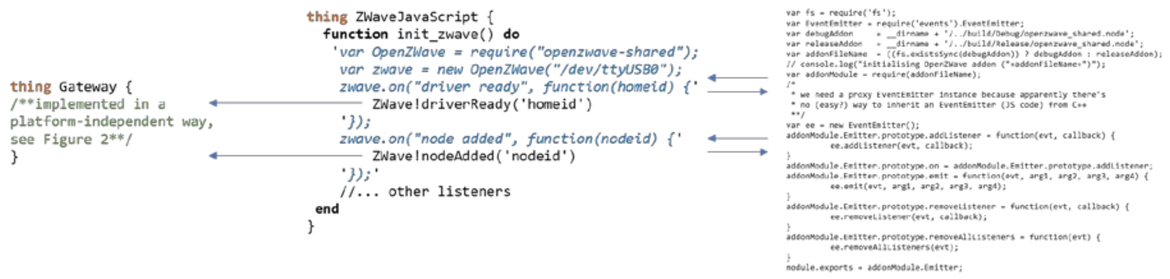
Once the general architecture is defined, our approach allows for specification of the components' business logic in a platform-independent way using statecharts and the action language. ThingML statecharts include composites, parallel regions, and history states. The state machines typically react to events corresponding to incoming messages on a component's port. The action language lets developers specify the guards, actions, variables, and functions in the component and lets the state machine send and receive messages on the component's port.

Facilitating Application Development

The ThingML approach also integrates three features that facilitate IoT application development. The first is a mechanism to rapidly wrap existing libraries by blending ThingML code with target code. The second is complex event processing (CEP) to handle complex reactive patterns. The third is dynamic sessions to handle dynamically discovered sensors.

Blending ThingML and native code. A platform-independent specification is advantageous in early development stages and for certain components that don't depend directly on low-level system or hardware features. However, once developers choose a concrete platform, some components must be implemented efficiently and must take advantage of the hardware features and any available suitable software library. To offer the required flexibility, our approach allows arbitrarily blending model actions with target-language actions. Mechanisms allow easily sharing variables and implementing calls and callbacks in both directions.

Figure 1 illustrates the implementation of a component linking to an existing JavaScript Z-Wave library. The blue code is plain JavaScript code in which we initialize the Z-Wave driver and register callbacks. We've woven ThingML code into these callbacks to emit ThingML events when the callbacks are invoked. This way, the gateway remains platform-independent, simply reacting to these plain ThingML events.



```
thing Gateway {
/**implemented in a
platform-independent way,
see Figure 2**/
}
```

```
thing ZWaveJavaScript {
  function init_zwave() do
   'var OpenZWave = require("openzwave-shared");
    var zwave = new OpenZWave("/dev/ttyUSB0");
    zwave.on("driver ready", function(homeid) {'
        ZWave!driverReady('homeid')
    '});
    zwave.on("node added", function(nodeid) {'
        ZWave!nodeAdded('nodeid')
    '});'
    //... other listeners
 end
}
```

```
var fs = require('fs');
var EventEmitter = require('events').EventEmitter;
var debugAddon     = __dirname + '/../build/Debug/openzwave_shared.node';
var releaseAddon   = __dirname + '/../build/Release/openzwave_shared.node';
var addonFileName  = ((fs.existsSync(debugAddon)) ? debugAddon : releaseAddon);
// console.log("initialising OpenZWave addon ("+addonFileName+")");
var addonModule = require(addonFileName);
/*
 * we need a proxy EventEmitter instance because apparently there's
 * no (easy?) way to inherit an EventEmitter (JS code) from C++
 **/
var ee = new EventEmitter();
addonModule.Emitter.prototype.addListener = function(evt, callback) {
        ee.addListener(evt, callback);
}
addonModule.Emitter.prototype.on = addonModule.Emitter.prototype.addListener;
addonModule.Emitter.prototype.emit = function(evt, arg1, arg2, arg3, arg4) {
        ee.emit(evt, arg1, arg2, arg3, arg4);
}
addonModule.Emitter.prototype.removeListener = function(evt, callback) {
        ee.removeListener(evt, callback);
}
addonModule.Emitter.prototype.removeAllListeners = function(evt) {
        ee.removeAllListeners(evt);
}
module.exports = addonModule.Emitter;
```

Figure 1. Platform-independent and platform-specific components in ThingML. The blue code is plain JavaScript code in which we initialize the Z-Wave driver and register callbacks. These callbacks contain ThingML code to emit ThingML events when the callbacks are invoked.

CEP. The IoT business logic typically reacts to streams of events from different sources and devices. **//7. Author: I reworked the next sentence; is it acceptable. I suggest to talk about "accidental complexity", see after?//** In our experience, when this logic is implemented with traditional state machines (or directly in the target language), accidental complexity arises from buffering sets of events and combining events from different sources. CEP languages and libraries such as ReactiveX[6] provide declarative ways to specify how to process events from different streams and combine them into new event streams.

Our approach defines CEP queries at the same level as state machines. Like a state machine, a CEP query processes a set of input messages and produces outputs messages. However, a CEP query's specification is fully declarative. The ThingML CEP includes operators to join and merge streams of messages and to process messages over windows, defined by time or the number of messages. Later, we'll examine a CEP query from the e-health application we mentioned earlier.

Sessions. IoT gateway components dynamically handle connections and interactions with devices coming and going in the gateway's network. Handling such dynamicity is similar to user sessions in web applications. To implement such behavior, we could have used ad hoc solutions. Instead, to provide a systematic solution and avoid memory management pitfalls, ThingML state machines employ sessions.

A session is a dynamically instantiated parallel region, initialized with a copy of the context (set of properties) of its parent, at fork time. A session executes its own behavior (state machine, CEP, or functions), which can only access and modify its own context. The parent and sessions can communicate only through asynchronous messages, like any other component. A session terminates when it reaches a final state. Later, we'll give an example of using sessions in the e-health application.

Generating Code for Heterogeneous Platforms

Our key lesson learned from applying model-based development with practitioners is that code matters. Whereas architects and managers are often more comfortable with models and systematic code generation, experienced developers typically are concerned with the quality, maintainability, and integration of code generated from models. All successful uses of the ThingML approach in nontrivial projects have required tailoring the code generators to fit not only the target platform but also the project's coding styles, libraries, APIs, and legacy components.

To accommodate this reality, the ThingML code generation framework empowers developers to adapt and extend code generators to fit the needs of their domain and projects. This framework consists of an abstract framework and a set of ready-made code generators for three languages (C/C++, Java, and JavaScript) and several libraries and open platforms (Arduino, Raspberry Pi, Intel Edison, Linux, and so on). In an IoT application, ThingML components are distributed on different heterogeneous platforms for which different code generators are used. Specific code generator plug-ins allow generating communication and message exchange between components on different platforms.

To allow efficient customization of any part of the generated code, the framework is organized around 10 extension points, each generating a particular aspect of the source code. For example, one generates public APIs, whereas another generates a component's implementation. In an IoT system, each component can use a different combination of code generation plug-ins.

An important issue when using, extending, and customizing code generators is to ensure they all provide the same execution semantics for the ThingML constructs. To assist developers, a testing framework consisting of more than 100 test cases and a set of test harnesses (which can be extended for new target platforms) allows automatic diagnostics of any code generator. The testing framework generates code for all test cases, executing code on the target platforms, collecting the execution traces, and comparing them with a set of reference traces.

In practice, customizations can range from simple tweaks of an existing generator to full development of a generator for a new platform and target programming language. Following are three examples together with the approximate effort required.

First, customizing a given action—for example, having ThingML print to save logs in a file rather than display text in the console—takes 5 to 50 LOC.

Another common customization is to change how the generated projects are structured and built—for example, changing from the default Maven to Gradle to manage and build Java projects. This takes approximately 100 LOC.

Finally, in a slightly more complex customization, the framework allows building plug-ins to handle message exchange between components on different platforms. For example, enabling any Java component to communicate with any C component through MQTT (Message Queuing Telemetry Transport) takes approximately 800 LOC—approximately 400 LOC for Java and 400 LOC for C. Of those 400 LOC, roughly half handle message marshaling and unmarshaling; the other half handle transporting the payload using an MQTT library. By default, a binary serialization[7] and a JSON (JavaScript Object Notation) serialization are available for the supported languages, as well as MQTT, WebSocket, and serial transport protocols.

Implementing a compiler from scratch, without reusing existing plug-ins, requires approximately 10,000 LOC. This task isn't straightforward but is still accessible to most developers, or at least companies.

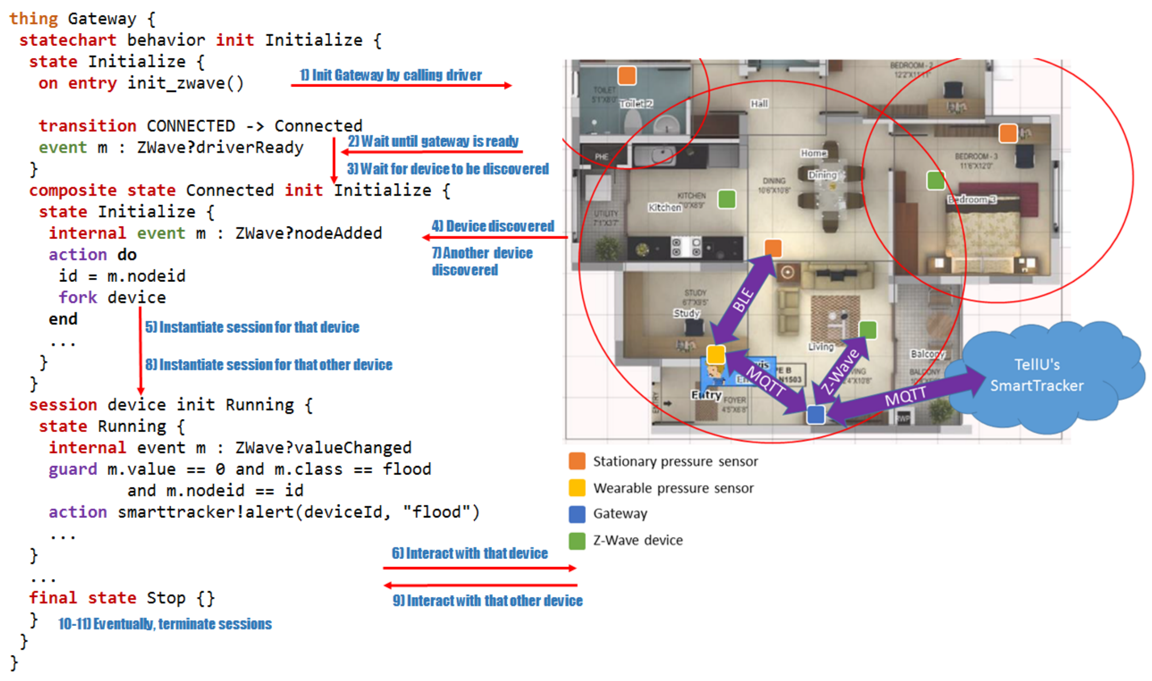
Developing an E-Health Fall Detection System

Tellu is a Norwegian IT company that developed SmartTracker, an IoT platform that lets users register sensors, store data, and define business rules using JBoss Drools rules executed in cloud servers. Commercial applications include geofencing and monitoring of goods and persons. Although such a centralized approach has been successful in certain domains, Tellu realized its limitations when developing e-health services, which require high-throughput data, low latency, and time-critical, reliable reactions. In this context, the business logic must be distributed throughout the IoT infrastructure to build a safe-and-sound system that doesn't need to rely on Internet connectivity or cloud services for critical functionality.

The core of Tellu's e-health solution is an air-pressure-based fall detection system. This system outperforms its competitors, which use wearable inertial sensors. However, in addition to a wearable sensor, the system requires a cluster of stationary pressure sensors in the environment.[8] This extra infrastructure also provides an accurate indoor positioning system that can trigger home automation tasks, such as turning lights on in a room without needing a motion sensor. The application's logic and software must be tuned to individual homes, patients, and devices.

The stationary sensors detect falls in a decentralized way on the basis of processed data sent by the wearable sensor over BLE (Bluetooth low energy, also called Bluetooth Smart). The stationary sensors are connected to a Wi-Fi network, allowing peer-to-peer communication for fall detection and communication with a gateway using an MQTT broker.

Tellu uses ThingML to implement the logic on the sensors and gateway. Figure 2 presents an overview of the infrastructure and deployment in a home and an excerpt of the state machine deployed in the gateway.



```
thing Gateway {
 statechart behavior init Initialize {
  state Initialize {
   on entry init_zwave()

   transition CONNECTED -> Connected
   event m : ZWave?driverReady
  }
  composite state Connected init Initialize {
   state Initialize {
    internal event m : ZWave?nodeAdded
    action do
     id = m.nodeid
     fork device
    end
    ...
   }
  }
  session device init Running {
   state Running {
    internal event m : ZWave?valueChanged
    guard m.value == 0 and m.class == flood
           and m.nodeid == id
    action smarttracker!alert(deviceId, "flood")
    ...
  }
  ...
  final state Stop {}
  }
 }
}
```

1) Init Gateway by calling driver
2) Wait until gateway is ready
3) Wait for device to be discovered
4) Device discovered
7) Another device discovered
5) Instantiate session for that device
8) Instantiate session for that other device
6) Interact with that device
9) Interact with that other device
10-11) Eventually, terminate sessions
Toilet 2
Hall
Bedroom-2
Home
Dining
Kitchen
Bedroom-3
Study
Living
Balcony
Entry
BLE
MQTT
Z-Wave
MQTT
TellU's SmartTracker
🟧 Stationary pressure sensor
🟨 Wearable pressure sensor
🟦 Gateway
🟩 Z-Wave device

Figure 2. A snippet of Tellu's e-health and home automation solution. (a) An overview of the infrastructure and deployment. (b) An excerpt of the state machine employed in the gateway. The gateway uses dynamic

sessions to manage the dynamic discovery and management of devices. BLE is Bluetooth low energy; MQTT is Message Queuing Telemetry Transport.

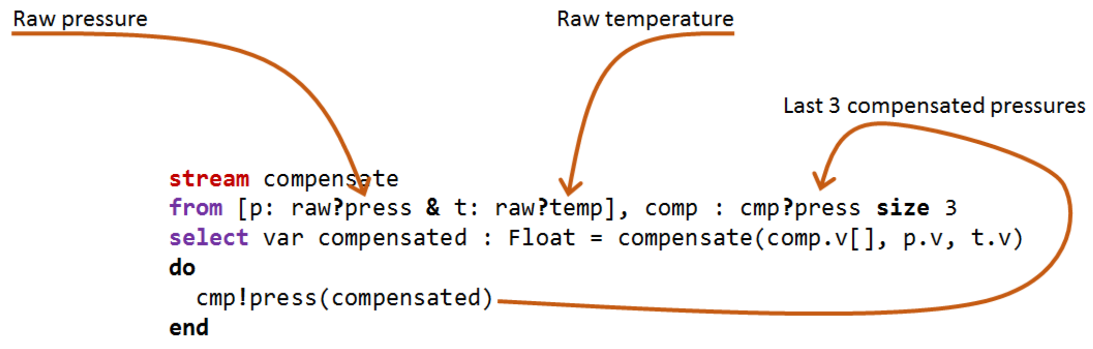
The system comprises more than 25 ThingML components running on four types of embedded nodes:

- ARM Cortex M3 microcontroller-based wearable sensors,
- Intel Edison-based stationary pressure sensor nodes,
- a Raspberry Pi-based gateway for the initial system prototypes, and
- a MIPS-based proprietary gateway for the production-ready system.

Regarding protocols, the system relies mostly on BLE, MQTT (over Wi-Fi and Ethernet), and Z-Wave.

To manage the dynamic device discovery and management, the gateway uses sessions (see Figure 2). When the system discovers a new device (`Z-Wave?nodeAdded`), the gateway forks a device session (`fork device`), with an up-to-date ID, that will handle the device-related messages.

The sensors leverage CEP to elegantly implement and easily tune the pressure compensation algorithm required for precise measurements (see Figure 3). The stream is triggered when the system receives a raw temperature and raw pressure at the same time. To customize the time interval during which two events can be joined, users can enter an annotation. The stream also merges the last three compensated pressures, which it actually produces. To compute the actual compensated pressure, the `compensate` function applies a polynomial function of the buffer of the last compensated values and the current pressure and temperature.



```
stream compensate
from [p: raw?press & t: raw?temp], comp : cmp?press size 3
select var compensated : Float = compensate(comp.v[], p.v, t.v)
do
  cmp!press(compensated)
end
```

Figure 3. A pressure compensation algorithm expressed as a ThingML CEP (complex event processing) stream. The pressure sensors leverage CEP to implement and tune the pressure compensation algorithm required for precise measurements.

Although Tellu's e-health system is still under development, the ThingML approach has already provided four key benefits. First, ThingML components, and the strict asynchronous message-passing communication semantics, force the system to elicit all interactions between components. This greatly facilitates system integration and testing because it's now simple to mock components.

Second, much of the technical code dealing with buffer and timer allocation and management, which had been expressed in ThingML or the target language, was simply removed from the models and delegated to the compilers after we introduced CEP and sessions.

Third, the ThingML approach has greatly facilitated the transition from prototyping to production. As we mentioned before, instead of the initial Raspberry Pi-based home gateway, the system now uses a proprietary MIPS-based gateway. Even though this gateway contains some platform-specific parts (see Figure 1), migration to a new platform is straightforward because all platform-specific references are explicit in the ThingML model. For a software company such as Tellu, this ability to rapidly change platforms is important to avoid being locked into any hardware solution.

Finally, developers can support their own target platforms and protocols by both wrapping existing libraries in ThingML and customizing the code generators using the ThingML framework, including support for the ARM Cortex M3 microcontroller.

The development of Tellu's e-health system is a good example of the process and problems involved in developing commercial IoT systems: heterogeneity, resource constraints, distribution, reliability and privacy requirements, and hardware and software codevelopment. In this context, the ThingML approach is a first step toward managing system complexity and the jungle of underlying IoT platforms. Our experience applying our approach to e-health as well as robotics, sensor networks, media, and home automation shows that it works in practice and benefits practitioners.

Nevertheless, important software engineering challenges remain, particularly regarding software deployment and updates and reliable, predictable sharing of computational resources and devices among IoT applications. In the context of cloud computing, approaches such as Docker (www.docker.com)[9] or LXC (linux containers) allow containerizing components and applications to handle their deployment and "sandbox" their execution on mutualized servers. In terms of the IoT, Docker and LXC has been ported to platforms as constrained as the Raspberry Pi,[10] making it available on the intermediate IoT nodes. However, different approaches are needed for smaller IoT nodes. For constrained platforms, scheduling algorithms to handle mixed-criticality systems[11] could provide a baseline for the predictable execution of competing processes. Models offer an abstraction that can provide a sound, efficient basis for analyzing emergent behaviors.

Acknowledgments

The EU FP7 HEADS project (grant agreement 611337) has supported this research.

References

1. G.P. Picco, "Software Engineering and Wireless Sensor Networks: Happy Marriage or Consensual Divorce?," *Proc. 2005 FSE/SDP Workshop Future of Software Eng. Research* (FoSER 10), 2005, pp. 283–286.
2. I. Stoica et al., "Chord: A Scalable Peer-to-Peer Lookup Service for Internet Applications," *ACM SIGCOMM Computer Communication Rev.*, vol. 4, no. 31, 2001, pp. 149–160.
3. P. Grace, G. Blair, and S. Samuel, "A Reflective Framework for Discovery and Interaction in Heterogeneous Mobile Environments," *ACM SIGMOBILE Mobile Computing and Communications Rev.*, vol. 1, no. 9, 2005, pp. 2–14.
4. A. Marinos and G. Briscoe, "Community Cloud Computing," *Cloud Computing*, LNCS 5931, Springer, 2009, pp. 472–484.
5. N. Harrand et al., "ThingML: A Language and Code Generation Framework for Heterogeneous Targets," *Proc. ACM/IEEE 19th Int'l Conf. Model-Driven Engineering Languages and Systems* (MODELS 16), 2016, pp. 125–135.
6. E. Meijer, "Reactive Extensions (Rx): Curing Your Asynchronous Programming Blues," *Proc. 2010 ACM SIGPLAN Workshop Commercial Users of Functional Programming* (CUFP 10), 2010, article 11.
7. F. Fleurey et al., "MDE to Manage Communications with and between Resource-Constrained Systems," *Model Driven Engineering Languages and Systems*, LNCS 6981, Springer, 2011, pp. 349–363.
8. A. Liverud, "New Sensor Will Make Life Safer for the Elderly," *Gemini*, 9 Feb. 2016; gemini.no/en/2016/02/new-sensor-will-make-life-safer-for-the-elderly.
9. F. Fouquet et al., "Dissemination of Reconfiguration Policies on Mesh Networks," *Distributed Applications and Interoperable Systems*, LNCS 7272, Springer, 2016, pp. 16–30.
10. F.P. Tso et al. "The Glasgow raspberry pi cloud: a scale model for cloud computing infrastructures". *First International Workshop on Resource Management of Cloud Computing* (CCRM) *at IEEE ICDCS*, 2013, Philadelphia, PA, USA.
11. S. Baruah, H. Li, and L. Stougie, "Towards the Design of Certifiable Mixed-Criticality Systems," *Proc. 16th IEEE Real-Time and Embedded Technology and Applications Symp.* (RTAS 10), 2010, pp. 13–22.

Brice Morin *is a research scientist at SINTEF Information and Communication Technology in Oslo. He's* *interested in providing strong, yet practical modeling foundations for the Internet of Things through the ThingML language. Morin received a PhD in computer science from the University of Rennes 1. Contact him at brice.morin@sintef.no.*

***Nicolas Harrand** is an engineer at SINTEF Information and Communication Technology in Oslo. He's interested in applying model-driven engineering to heterogeneous and distributed systems. Harrand received an engineering degree from École Nationale Supérieure D'informatique et de Mathématiques Appliquées de Grenoble. Contact him at nicolas.harrand@sintef.no.*

***Franck Fleurey** is a senior research scientist at SINTEF Information and Communication Technology in Oslo. His research interests include model-driven software engineering, embedded systems, and dynamic adaptive systems. Fleurey received a PhD in computer science from the University of Rennes 1. Contact him at franck.fleurey@sintef.no.*

A History of the ThingML Approach

The ThingML (Internet-of-Things Modeling Language) approach aims to transfer the promises of academic model-based software development to the industry. Over the past six years, three major iterations have shaped the ThingML language and tools.

The first iteration exploited existing UML tools. We developed a minimalistic language to describe sensor-based architectures and support integration of the code generated from commercial state-machine tools with existing APIs. The development (with the oil-and-gas industry) of a large sensor network was hindered by the difficulty of using a graphical specification (beyond toy examples), resource-constrained platforms' harsh coding requirements, and the importance of legacy and third-party components.

The second iteration introduced a textual syntax for the state machines together with a first-class action language. Three independent compilers targeted microcontrollers, Posix C, and Java, respectively. Developing a medical-rehabilitation robot in the CORBYS (Cognitive Control Framework for Robotic Systems; corbys.eu) project and a smart-home gateway in the Arrowhead project (arrowhead.eu) confirmed this textual language's utility. It also showed the limitations of having a fixed set of rather monolithic compilers, which had to be "hacked" to fulfil those projects' requirements.

A key lesson learned is that what ultimately matters is code. Model-based techniques should attempt not to replace or hide code but to support practitioners in producing the code they need, more systematically and more efficiently.

The third iteration, developed in the HEADS project (heads-project.eu), applies that lesson and is being practically implemented in two commercial products in the media and e-health domains. The project's main contribution is a code generation framework and an associated methodology to give practitioners full control of the code by letting them easily customize compilers for their needs. This third iteration also extends the ThingML language with additional concepts such as Complex-Event Processing and dynamic sessions (see the section "Modeling IoT Software" in the main article).