A Generative Middleware for Heterogeneous and Distributed Services

Brice Morin and Franck Fleurey
Sintef ICT
Oslo, Norway
firstname.name@sintef.no

Knut Eilif Husa
TellU
Oslo, Norway
knut.eilif.husa@tellu.no

Olivier Barais
INRIA, IRISA, Université de Rennes 1
Rennes, France
barais@irisa.fr

***Abstract*—Modern software-based services increasingly rely on a highly heterogeneous and dynamic interconnection of platforms and devices offering a wide diversity of capabilities ranging from cloud server with virtually unlimited resources down to micro-controllers with only a few KB of RAM. This paper motivates the fact that no single software framework or software engineering approach is suited to span across this range, and proposes an approach which leverages the latest advances in model-driven engineering, generative techniques and models@runtime in order to tame this tremendous heterogeneity. This paper presents a set of languages dedicated to the integration, deployment and continuous operation of existing libraries and components already available and implemented in various languages. The proposed approach is validated on an industrial case study in the eHealth domain, implemented by an industrial partner that provide an qualitative evaluation of the approach. This case study involves a large number of sensors, devices and gateways based on Rasperry Pi, Intel Edison and Arduino.**

Keywords. *Heterogeneity, Distribution, Model-Driven Engineering, Dynamic Component Model.*

I. Introduction

Modern software-based services increasingly rely on a highly heterogeneous and dynamic 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 services, for example in the eHealth domain, the heterogeneity, diversity and vast distribution represent daunting challenges.

To implement such distributed services, which will be deployed on an heterogeneous infrastructure, software developers have access to a plethora of components and libraries (both open and closed source) that they can use to rapidly build up applications and services. Even though programming languages tend to become more and more versatile (for example JavaScript evolving from a client-side scripting language to a server-side programming language with Node.JS), a study of a large number of open-source projects [1] indicates that no programming language is able to stretch so that they can fully cover the whole range of platforms. C/C++ remains the de-facto language for embedded systems as it gives programmers control on every bits and bytes, while Java (Android) and JavaScript/HTML5 is the winning duo for mobile applications. For developing large-scale, distributed algorithms and systems, the abstraction providing by Java seems to prevail over the performances provided by lower-level languages like C/C++. Beside the very popular languages (Java, JavaScript, C/C++), many languages are being used in specific niches e.g. Lua is used as a scripting language in several home automation gateways or game engines.

In any large-scale distributed system, it should be expected that several programming languages will be used to implement or reuse different components. On the one hand, the number of languages should be kept as low and as coherent as possible to keep the development team coherent and enable experts in one language to still understand expert in another language and the code they wrote, contributing to reducing the overall complexity of the system. On the other hand, the ”cost” of sticking with a language that is ”not right for the job” should be carefully balanced. While Java Servlets and JSP allowed Java developers to get started with Web development, the decadent popularity of those technologies (while Java itself still remains popular) indicates they were ”not right for the job” e.g. all banks in Norway have moved away from those technologies and migrated their system, at a high cost.

This paper presents a set of languages based on well-established formalisms (components with port and messages for the interfaces and state machines for the implementation) dedicated to the integration, deployment and continuous operation of existing libraries and components already available and implemented in various languages. This approach is validated on an industrial case study in the eHealth domain and implemented by TellU. This case study involves a large number of sensors, devices and gateways based on Rasperry Pi, Intel Edison and Arduino.

The remainder of this paper is organized as follows. Section II motivates our work, both the particular case of an eHealth application and the associated general software engineering challenges. Section III presents our approach to tackle those challenges. In Section IV, we apply our approach on an eHealth service developed by TellU. Section V presents related work while Section VI concludes this paper.

II. Motivations

A. Challenges engineering a modern eHealth service

eHealth is a growing market in Europe and world-wide, in particular due to the ageing of our societies. 15 millions elderly in Europe are already equipped with a tele-care alarm, typically in the form of a simple necklace button that the elderly can press in case she needs help. This will in turn put the elderly in contact with medical services. While tele-care alarm services are of course very useful services, more and more promoted by public authorities, operating those services is however not an easy task due to the numerous false positive (elderly pushing the button while everything is fine) and false negative alarms (elderly not able to push the button e.g. after a fall). This drastically hinders the safety and cost-efficiency of tele-care services. To improve tele-care services and help elderly to stay home as long as possible, TellU is currently developing a smart-home system to control equipment that are normally present in smart homes to make life more comfortable (automatic light control, door locks, and heater control, etc.) and safer for elderly people. One of the main fears for elderly people is the fear for falling and not being helped. This fear causes elderly to isolate and become less active, which in turn make them more exposed to illness. In addition to integrating home-automation equipment, TellU is developing a fall detection system (patent pending) based on a distributed sensor network for measurements of air pressure from both stationary and wearable sensors. This distributed sensor network in addition provides a way locate elderly within their homes. Physicians, care-givers, family etc. can use this information to monitor if the elderly residents have an adequate activity level during the day. Furthermore, several services can be automated based on measured indoor locations and controllable equipment, e.g. light control, turn of stove when leaving house, etc. TellU also plan to include sensors for measuring physiological parameters like heart-rate and skin temperature. In this way care-givers can monitor if the elderly resident is not feeling well. Fall detection and heart-rate monitoring should also be applicable outdoors to give the elderly the needed confidence that they will get assistance if they need help outside of the home.

eHealth and many other types are services are no longer one-device services but need to leverage a large number of heterogeneous software and hardware platforms to fully complete their goals. Developers are thus facing new Software Engineering challenges as detailed in the next sub-section.

B. Software Engineering Challenges

The infrastructure supporting Heterogeneous and Distributed services (HD services) typically spans across a continuum of devices and platforms ranging from microcontroller-based devices up to Clouds. 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 services on top of such an infrastructure, we queried a popular open-source repository (GitHub) to evaluate the heterogeneity of programming languages across the continuum [1]. 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 Cyber Physical Systems (CPS) have to be distributed across the continuum, and 6) Embedded systems, as a large and important part of the service implementations will run as close as possible to physical world, embedded into sensors, devices and gateways.

This study indicates that no programming language is popular across the whole continuum: Java and JavaScript (and to some extent, Python and Ruby) are popular in the higher-end of the continuum (cloud and mobile) but not popular for the lower end, whereas C (and to some extent, C++) is a clear choice for developers targeting embedded and microcontroller-based systems.

While it might appear that a combination of C/C++, JavaScript and Java should be able to cover the whole continuum of CPS, in practice it does not exclude the need for other programming languages. For example, the Fibaro Home Center 2 (a gateway for home automation based on the Z-Wave protocol) uses Lua as scripting language to define automation rules. Another example is the BlueGiga BlueTooth Smart Module, which can be scripted using BGScript, a proprietary scripting language. This shows that each part of an infrastructure might require the use of a niche language, middleware or library to be exploited to its full potential.

To tackle this heterogeneity, developers typically need to determine a trade-off between alternative solutions (described in details in [1]). A typical solution consists in using Internet-connected devices that simply push all the data to bigger nodes (e.g. in the Cloud). This way, the service can be implemented homogeneously in a high-level language like Java on the larger nodes. However, this requires the devices to have a permanent Internet connection, which rapidly drains the batteries of mobile and wearable devices. Also, the whole service is likely to fail if the Internet connection is lost as the devices do not run any logic, which is not acceptable for a large set of safety-critical services. For this type of services, it is important that the logic can still run even after the failure of multiple nodes or communication channels. Some logic has to be implemented directly in micro-controllers, some logic in intermediate gateways and some logic can possibly be implemented in some backend or cloud servers. This requires a large team of developers with different skills (from a C/assembly coder optimizing bits and bytes to a Java developers implementing large-scale consensus algorithms to extract a coherent overview of the system). This implies high development and integration costs, which is acceptable compared to the price and safety requirements of a plane or a car, but which can significantly hinder the large adoption of e.g. eHealth services.

III. HEADS: A Generative Middleware Approach for the Development of HD Services

Rather than providing yet another programming language supposedly able to address all the concerns needed for HD services, the HEADS approach proposes to rely on abstractions on top of existing programming languages to:

- enable the integration of existing C/C++, Java, JavaScript libraries (and potentially libraries in other languages). In other words, promoting an existing library as a HEADS component, which can then be manipulated by the different tools provided by HEADS, should only require a minimal effort.
- provide expressive constructs to implement the necessary ”glue code” to ensure that different libraries (potentially in different languages) can communicate locally or asynchronously over a network
- ease the reuse of fragments of logic across different platforms and languages
- ease the deployment, operation and maintenance of large-scale and distributed assemblies of heterogeneous component.

HEADS also intensively rely on generative techniques to produce C/C++, Java, JavaScript code, which:

- aims at being as efficient and as readable as code written by experienced programmers. In particular, the generated code is idiomatic e.g. proper Object-Oriented Java code for the JVM versus optimized C code with no dynamic allocation for small micro-controllers.
- provides clear public APIs to enable any programmer to interact with it with no need for her to use any HEADS-specific tool, just her favorite C/C++, Java, JavaScript editor or IDE.
- has few (ideally none) dependencies to any HEADS runtime libraries

The HEADS Design Language and Transformation Framework are available as an open-source projects on GitHub[1].

A. HEADS Design Language

The ambition of the HEADS design language is to support the implementation and integration of the different parts of HD-services, including the integration of legacy and of-the-shelf components and libraries. The HEADS 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 HEADS modeling 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-modeling or re-implementing it. In the design of the HEADS 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 artifacts.

For the components developed from scratch or for which the target runtime environment might change, the HEADS design language provides with all the required expressiveness to fully specify the behavior 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 Model-Driven Engineering platform-independent models with a set of code generators (or compilers) for different platforms. At the other end of the spectrum, for the integration of an existing component, the HEADS approach allow modeling 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 HEADS 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 existing libraries, platform features and application logic. In practice most of the components of a HD-service fall under this category and efficiently being able to integrate these different elements a key goal. The way the HEADS approach implements such capabilities is two-fold. First, a set of special constructs and actions are included in the HEADS action languages in order to seamlessly interleave platforms specific code and platform independent code. Second, the HEADS approach relies on a highly customizable code generation framework which can be tailored to specific target languages, middleware, operating systems, libraries and even build systems, as described in the next sub-section.

1) Defining the interfaces of components: The HEADS Design Language provides constructs to implement lightweight components which communicate asynchronously with other components. In the case of a distributed system running on top of a heterogeneous infrastructure, stronger assumptions regarding communication are simply not realistic. The API of a component basically follows a format well-established in the CBSE community: a set or ports specifying which message can be sent and received by the component.

For example, a timer component would describe the following API:

```
thing fragment TimerMsgs {
  message start(delay : Integer); // Start the Timer
  message cancel(); // Cancel the Timer
  message timeout(); // Notification
}
thing fragment Timer includes TimerMsgs {
  provided port timer {
    sends timeout
    receives start, cancel
  }
}
```

Note that ports are bi-directional as they can both send and receive message. Our notion of port should be understood as an atomic service, which is either provided or required. The service itself typically requires exchanging a set of

[1]https://github.com/SINTEF-9012/ThingML

message, typically a request (like `start`) and a response (like `timeout`).

2) Wrapping existing libraries: A timer component is typically useful in any language. The API we have just defined in fully platform-independent, as it contains no Java, JavaScript or C/C++. As all those languages already providing native timing facilities, we simply need to wrap those facilities into platform-specific components.

The script below shows how the timer is implemented in JavaScript, simply by relying on JavaScript timers:

```
thing TimerJS includes Timer {
  function cancel() do 'clearTimeout(this.timer);'
    end
  function start(delay : Integer) do
    'this.timer = setTimeout(function(){'
    timer!timeout()
    '},' & delay & ');'
  end
  statechart SoftTimer init default {
    state default {
      internal start event m : timer?start
      guard m.delay > 0
      action start(m.delay)
      internal cancel event m : timer?cancel
      action cancel()
    }
  }
}
```

Basically, anytime a start message is received on the timer port, we first check that the delay is positive. If so, we call the start function, which is a simple wrapper around the native `setTimeout` function provided in JavaScript. Native code *i.e.* code that is directly expressed in the target language (here JavaScript) should be written between simple quotes. Native code can be interleaved with actions and expression of the HEADS Modeling Language. This way, the developer does not need to know how the generated code will look like to be able to wrap her library. No matter if the delay variable is renamed in the generated e.g. to avoid name conflict or if the action of sending the timeout actually implemented in a generated function called `sendtimeoutOnTimer`. She just can simply refer to the delay variable and send a message with no knowledge of the underlying implementation of those mechanisms.

In the same way, the timer can be mapped to Java:

```
object JThread @java_type "Thread"
property timer : JThread

function start(delay : Integer) do
  timer = 'new Thread() {
    public void run() {
    sleep(' & delay & ');'
    timer!timer_timeout()
  '};'
  '' & timer & '.start();'
end
```

3) Implementing components: To orchestrate messages and define the behavior of components, beyond the wrapping of existing libraries of components, the HEADS Design Language provides a set of facilities:

- Imperative programming to implement simple procedures, either directly using the actions and expression languages provided by the HEADS Modeling Language or by using the features of the target languages (or a mix of both)
- Event-Condition-Action (ECA) rules, which basically allows reacting to incoming events, independently from any state e.g. unconditionally shut down the system if emergency button has been pressed.
- Composite State Machine, also including parallel regions, which allows reacting to a sequence of incoming events e.g. to make sure all drivers are initialized in the proper order before accepting any other command
- Complex Event Processing (CEP), which allows reacting to patterns in large flows of events, without explicitly defining all possible sequences of events.

The combination of those four paradigms enables service developers to express advanced behavior, which can efficiently tame large flows and data and process them to infer relevant information, as illustrated in Figure 1. A typical use-case is to use imperative programming to write drivers interacting with the physical world (sensors, actuators) or 3rd party software systems. As existing programmatic imperative APIs already exist, they can easily be integrated with the imperative approach.

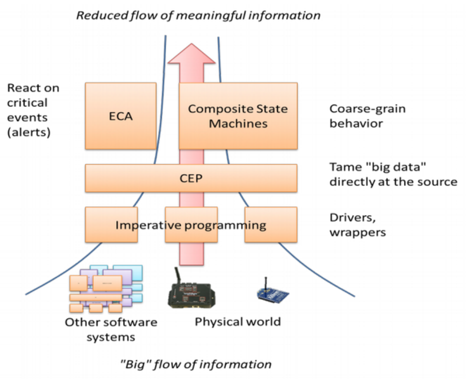

Reduced flow of meaningful information
React on critical events (alerts)
ECA
Composite State Machines
Coarse-grain behavior
CEP
Tame "big data" directly at the source
Imperative programming
Drivers, wrappers
Other software systems
Physical world
"Big" flow of information

Fig. 1. Event Processing

As sensors might generate a large amount of data, a CEP layer close to the sensors enables service designers to express rules able to handle those data and to extract some relevant information (e.g. alert, status). The ECA paradigm is then particularly suited to handle alerts as they enable to enact some reflex-like actions (similarly to the way the spine ”shortcuts” the brain to enact a fast movement e.g., when one burns his finger). Finally, composite state machines enable to describe advanced behavior, orchestrating and adapting to a set of events coming from the CEP.

The HEADS Design Language is conceptually a usable sub-set of the UML (Composite Statecharts and Component Diagrams) with a syntactical notation rather than a graphical notation, so that it remains closer what programmers already know. In addition to the introduction of CEP in state machines, a key difference with UML is that our language comes with a first class action language so that models are not polluted with so-called ”Opaque Behavior” (basically a String where code from the target language is directly outputted). This action language is basically the common subset of what is found in most languages, including numerical and Boolean algebra, control structures, functions, variable declaration and assignments, etc. The only action that is not typically found in programming language is the ability to asynchronously send a message through a port (rather than a simple method call).

ECA rules are basically internal transitions, as in the JavaScript timer. Those rules basically react on an event and can be optionally guarded with any boolean expression, e.g. involving properties of the component and/or parameters of the message. Any action can then be triggered, either fully expressed at a platform-independent level or mixed with the target language (as in the JavaScript timer). As for the state machine, we support all the concepts present in the UML, including composite states and concurrent regions.

The CEP concepts currently included in the HEADS Modeling Language is a sub-set of the concepts of ReactiveX (or other CEP language such a EPL available in Esper). We provide support for joining and merging flows of events, time or length windows and arbitrary filters.

```
stream lengthW
  from e : [weather1?temp | weather2?temp]::keep if
    inRange(e)::during 1000*60 by 1000*60
  select avg : average(e.t[]),
         min : min(e.t[]),
         max : max(e.t[])
action report!temp(avg, min, max)
```

This CEP stream illustrate most of the concepts currently integrated. It performs a merge between two temperature sensors (connected on ports weather1 and weather2). Basically, all temperature measurements coming from both sensor will be piped in a new stream. All these measurements are then filtered using a custom `inRange` operator defined by the developer, which will discard values that are outside a range and which are most likely due to an erroneous measurement rather than a correct measurement being too cold or too warn. The merging and the filtering happens on a time window during one minute which progress by slices of one minute. While such kind of CEP query could be implemented directly in Java, JavaScript or C, or at a more abstract level using the HEADS Modeling Language (state machine), it alleviates in any case the developer from writing a lot of ”plumbing code” (timers, buffers, etc), which if not implemented correctly can have dramatic impact on memory and performances.

B. HEADS Transformation Framework

The HEADS Transformation Framework is responsible for “compiling” HEADS Design Models into source code for a large variety of languages. Rather than implementing one monolithic compiler for each language we target, the HEADS Transformation Framework instead is architected as a modular object-oriented framework. This framework promotes reuse of code among different compilers. A total of 8 formal extension points have been identified in the HEADS code generation framework in order to allow the developer to easily and efficiently customizing some parts of the code generation while reusing the rest.

1) Role of the extension points:

a) Actions / Expressions / Functions : The implementation of this extension point consists of a visitor on the Actions and Expressions part of the HEADS Design Language. New code generators can be created by inheriting from that abstract visitor and implementing all its methods. Alternatively, if only a minor modification of an existing code generator is needed, it is possible to inherit from the existing visitor and only override a subset of its methods. Most of the actions and expressions (26 out of 35 concepts) is actually defined in the framework, as most of the programming languages have similar syntax for numerical and Boolean algebra, control structures, etc. The script below shows how a if condition with optional else is compiled. The same code is reused in all Java, JavaScript and C compilers.

```
@Override
public void generate(ConditionalAction action,
    StringBuilder builder, Context ctx) {
  builder.append(”if(”);
  generate(action.getCondition(), builder, ctx);
  builder.append(”) {\n”);
  generate(action.getAction(), builder, ctx);
  builder.append(”\n}”);
  if (action.getElseAction() != null) {
    builder.append(” else {\n”);
    generate(action.getElseAction(), builder, ctx);
    builder.append(”\n}”);
  }
  builder.append(”\n”);
}
```

b) Behavior implementation: This part of the code generator corresponds to the code generated from the state machine structures, ECA and CEP rules contained in Things. There is basically two main strategies to compile the behavior: target frameworks that are able to execute state machines and CEP or generate the whole logic to keep full control of what is executing at runtime. The first option is typically chosen for high-level languages not intended to run on resource-constrained devices, as it simplifies the code generation process (less code to generate) and generally improve the readability and maintainability of the generated code (less code to read and maintain). On resource-constrained devices however (down to 2 KB RAM) it is of primary importance to have control of every byte allocated and the overhead of embedding a framework is usually to heavy. The HEADS Transformation Framework does not impose nor favors any of those approaches and both can be implemented. The Java and JavaScript compilers use a framework approach whereas the family of C compilers use a full generative approach. In both cases, the framework provides support in the form of a set of helpers which pre-process the state machine e.g. to provide the list of all outgoing transitions for a given state, or the list of all messages being actually used by a component.

c) Ports / Messages /APIs: This part of the code generator corresponds to the wrapping of the generated code into reusable components on the target platform. Depending on the target platform, the language and the context in which the application is deployed, the code generated can be tailored to generate either custom modules or to fit particular coding constraints or middleware to be used on the target platform. As a best practice, the generated modules and APIs for things should be manually usable in case the rest of the system (or part of it) is written directly in the target language. For example, in object oriented languages, a facade and the observer pattern can be used to provide an easy to use API for the generated code. In C, a module with the proper header with structures and call-backs should be generated.

The following code for example generates a Java interfaces that any Java programmer can use to send messages to a generated component.

```
for (Port p : thing.allPorts()) {
  StringBuilder builder = ctx.getNewBuilder(thing.
    getName() + "_" + p.getName() + ".java");
  builder.append("public interface " + "I" +
  thing.getName() + "_" + p.getName() + "{\n");
  for (Message m : p.getReceives()) {
    builder.append("void " + m.getName() + "_via_" +
     p.getName() + "(");
    generateParameter(m, builder, ctx);
    builder.append(");\n");
  }
  builder.append("}");
}
```

For example, for the Java timer component, this would generate this simple API:

```
public interface ITimerJava_timer{
  void start_via_timer(short delay);
  void cancel_via_timer();
}
```

d) Main and Build: These two extension points are responsible for providing users with a turn-key solution to compile and run the generated code. In the main file, all components and instantiated and connected. The build files automate all tasks needed to properly compile e.g. fetching dependency. For the users, compiling and running the code is as simple as:

```
mvn install exec:java #for Java
npm install && node main.js #for JavaScript
make && ./myProgram #for C
```

This extension point makes it easy to for example switch from Maven to Gradle for the build of Java project.

e) Message Queing, Scheduling and Dispatching: Those extension points are related to the internal management of messages within a node. By default, messages are queued in a FIFO, typically reusing already structures. On some very constrained platforms (microcontrollers) the code for the FIFO also needs to be generated as the standard library is typically more limited.For the scheduling and dispatching of messages we typically rely on facilities available on the OS (threads) to ensure a fair distribution of messages among components and avoid starvation and race conditions. Again, on very constrained platforms that are too limited to run an OS, custom schedulers should be generated.

f) Connectors: This extension point is concerned with the serialization and transport of messages among components distributed over the network. For the serialization a default serialization of messages into arrays of bytes is provided [2], but developers can implement their own serialization. For example, we are currently implementing a generator to integrate with MessagePack [3]. For the transport itself, we usually rely on the large collections of communication channels already available in the HEADS Runtime platform (see next section): MQTT, WebSocket, Serial, etc.

2) Benefits of the HEADS Transformation Framework: While the HEADS Transformation Framework can be seen as a family of compilers (producing code in Java, JavaScript and C/C++), those compilers have a different nature than a compiler like GCC, producing machine code out of C source code. Working at a higher level of abstraction, by transforming a model into source code rather than source code into machine code, drastically reduced the cost of writing a HEADS compiler. While GCC alone is about 14 millions LoC, the whole HEADS Transformation Framework, also including the compilers targeting Java, JavaScript and C/C++ is less than 25,000 LoC (560 times less). Those LoC are distributed (approx.) as follows:

- 6000 LoC in the framework itself, that all compilers reuse. This includes facilities to manage files, generate consistent variable names, etc. It also includes the code for compiling most of the actions and expressions (26 out of 35) of the HEADS Design Language. New compiler (*e.g.* targeting Go, Lua, PHP) will benefit from those 6000 LoC, unless the targeted language is radically different, for example if it uses a Polish notation like Lisp where "a + b" is expressed as "+ a b".
- 3000 LoC for the Java compiler that is able to generate fully working Java code. The generated code targets a framework for the execution of the state machines and another for the execution of CEP streams, hence most of the "tricky" code does not need to be generated as it is handled directly in those frameworks. In addition, 400 lines of code are needed to generate wrappers able to "merge" fine-grained implementation components into coarser-grained deployment components (as explained in the next sub-section)
- 3000 LoC for the JavaScript, which strictly follow the same approach as for the Java compiler. About 400 LoC

are also needed to perform the wrapping.
- 11000 LoC for the family of C compilers, distributed (approx.) as follows:
  - 5000 for a generic C transformation framework shared by all C compilers
  - 6000 LoC shared among a POSIX compiler for Linux, an AVR 8-bit compiler (*e.g.* for Arduino) and an ARM 32-bit compiler (*e.g.* for Cypress PSoC5).

Basically, supporting a new high-level language (like Java) with available libraries for state machines and CEP and having an infix (standard) notation should in most cases be limited to writing about 3000 LoC. Redifining the compilation of actions and expressions to support Polish (prefix) or postfix notation should be about 500 additional LoC. Supporting a lower-level language (like C) where libraries are not available (or do not provide enough control on the memory to be allocated, etc) is a slightly more complex endeavor, but still accessible to most programmers. Writing the POSIX C compiler for Linux was about 7000 LoC (as the code for execution of the state machine needs to be generated, etc), including the generic C transformation framework. However, supporting different "dialects" of C required about 2000 LoC for AVR 8-bit and 2000 LoC for ARM 32-bit, which is a limited effort.

C. HEADS Deployment Language and Runtime platform

By default, the HEADS transformation framework generates standalone code that can be executed without any dependencies to HEADS tools and platforms. However, this code cannot easily be updated at runtime e.g. to substitute one component by another, though it is programmatically feasible to instantiate new components and connectors. In the HEADS Design language, components are indeed implementation units, in a way similar to Java classes. Wrapping each individual implementation unit in a deployable component having its own lifecycle at runtime would result in a large number of components that needs to be deployed and administrated at runtime. For example, in an implementation unit dealing with serial communication would typically be packed together with implementation units dealing with serialization/deserialization of messages in the same deploy unit. This way at runtime, the service operator would only need to deploy a single component responsible for serial communication, the (de)serialization aspect being hidden as an implementation detail. Figure 2 shows how HEADS implementation units can be wrapped into Heads deploy units.

Once wrapped into deploy units, components can be manipulated by the HEADS Deployment Language and actually deployed on the HEADS runtime platform, which evolves the Kevoree platform [4] initially developed for Java in order to also support JavaScript and more recently, .NET. The HEADS Deployment Language provide a way to write scripts describing which components to instantiate, how to configure them (set values of parameters), and how to connect components through communication channels. A large set of components and channels is already available off the shelf. At deployment time, the script, describing the components and

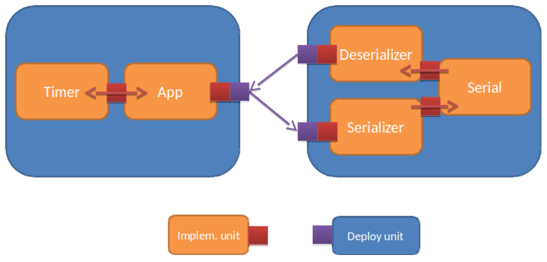


Fig. 2. Heads implementation and deploy units integration

channels deployed on the different nodes of the system, will be sent to the different nodes, which will interpret this script and actually deploy the necessary components, etc.

The HEADS Deployment Language and Runtime platforms are available as a set of open-source projects on GitHub[2].

IV. Engineering a Real-life eHealth Service with HEADS

This section describes how the HEADS approach has been applied to the TellU's eHealth system.

A. Overall Architecture

The overall architecture of the eHealth system is depicted in Figure 3. The system is composed of a home gateway which runs on a Raspberry Pi 2 (1 GHz ARMv7, 1GB RAM, Linux). This gateway is connected to a number of field nodes (typically one per main room in the house) via WiFi. Field nodes run on an Intel Edison (400 MHz x86, 1GB RAM, WiFi and Bluetooth Low Energy (BLE)). Each field node has a pressure cell integrated in order to provide an accurate measurement of the pressure in each room. A wearable sensor node running on a low power resource-constrained ARM Cortex M3 (80 MHz, 256 KB RAM) also integrates a pressure cell and regularly broadcasts air pressure measurement to all field node that are in the BLE range (typically 10m indoor). Based on these pressure measurements and the intensity of the BLE signal, field node can determine the position of the person in the house and if the person has feel (by computing an air pressure differential between the person's sensor and the fixed pressure in the field node). In addition, a set of sensor nodes are deployed in different rooms to measure temperature and light. Those nodes run on an Arduino Yn, which is composed of a resource-constrained microcontroller (16 MHz AVR-8bit, 2.5 KB RAM, no OS) and an embedded Linux processor (400 MHz MIPS, 64 MB RAM, WiFi, OpenWRT). The microcontroller part of the Yn is used to interact with the physical temperature and light sensors while the MIPS CPU and its embedded WiFi is used to communicate with the Gateway. Finally, the gateway also integrates a Z-Wave radio chip that can control and interact with a set of devices (switches, etc).

This particular HD-Service thus relies on an heterogeneous infrastructure composed from rather powerful 32-bit X86

[2]https://github.com/dukeboard/kevoree
https://github.com/kevoree/kevoree-js

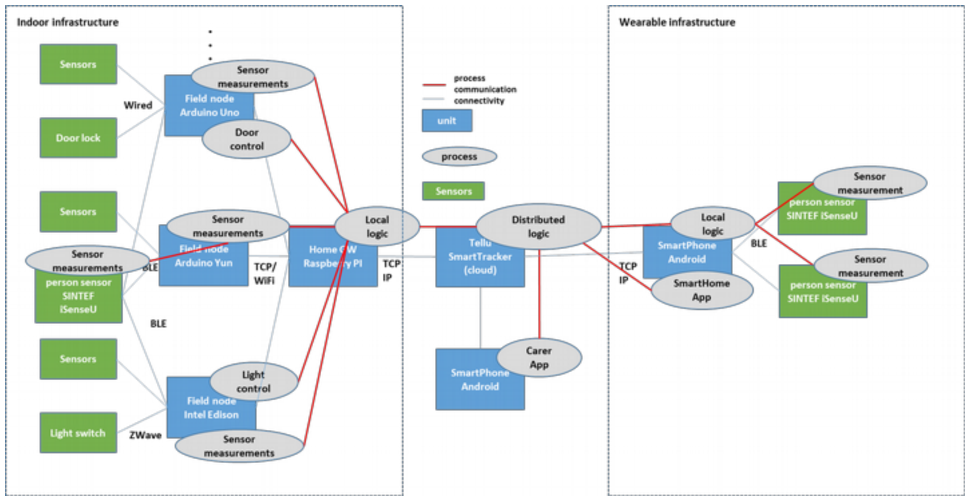


Fig. 3. Safe@Home system architecture

and ARM processors with 1 GB RAM down to 8-bit AVR microcontroller running at 16 MHz (about 60 times slower) and embedding only 2.5 KB RAM (about 400 000 times less). It also integrates a variety of radio protocols such as WiFi, BlueTooth or Z-Wave.

B. *Implementation of the Field Node*

At design/implementation time, the Field Node is composed of 8 core components implementing the logic of the node, and 2 other components that allows the field node to push information to the gateway via MQTT. Each of the implementation component is available as:

- A Platform-Independent Component (PIC), which describes the interfaces of the component i.e., the port and messages it exposes. Some PICs can also provide a full implementation of their behavior, if this behavior does not imply using low-level libraries/drivers that directly interfaces with the hardware or system APIs.
- A Platform-Specific Component (PSC), which describes the full behavior of a component including the access to hardware drivers and system APIs.

For example, the PIC associated to the BLE component describes messages related to the status of the BLE connection, such as stateChange (st : String); discover (peripheral : String); scanStart (); scanStop (). Those messages are orchestrated in the BLE PSC by a state machine (see Figure 4) interfacing with the ”noble” JavaScript module to deal with low-level details related to BlueTooth.

Overall, the Field Node is implemented with 300 LoC for the PICs and 1100 LoC for the PSCs. It produces 2000 LoC of JavaScript source code. This code can simply be run by executing the main.js file with Node.JS. However, this default implementation is rather monolithic at runtime and does not provide any dynamic reconfiguration capabilities. In order to provide such capabilities, this code is automatically wrapped into a component that can be manipulated and adapted at runtime. This additional component is fully generated and contained within 170 LoC. The automated wrapping is illustrated in Figure 5.

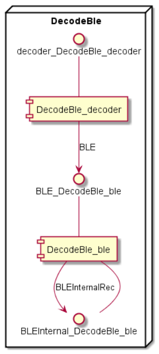


Fig. 4. Component diagram of BLE and Decoder

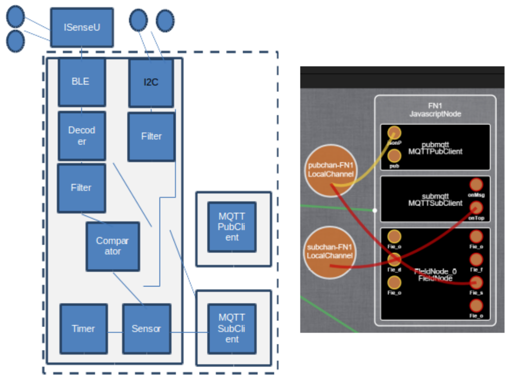
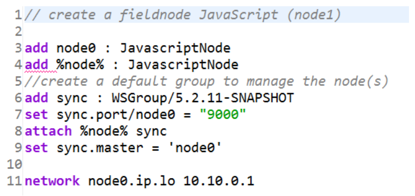


Fig. 5. Field-Node component structure and actual Deployment

C. *Deployment and Operation of the Fall Detection service*

All the field node are described by 3 scripts. The first one is a root script basically describing the default configuration of the node, containing information about the network and how to connect to the rest of the system. It is shown in Figure 6



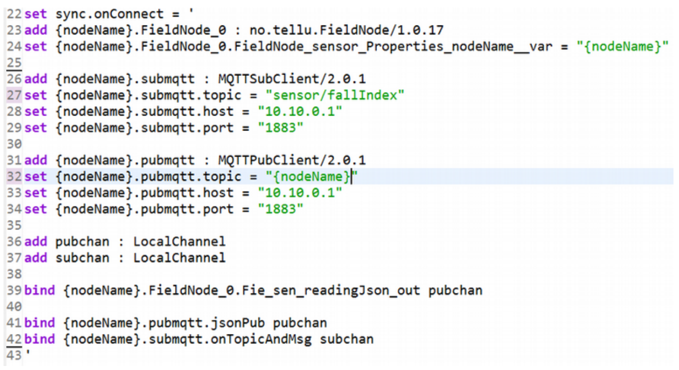
```
// create a fieldnode JavaScript (node1)

add node0 : JavascriptNode
add %node% : JavascriptNode
//create a default group to manage the node(s)
add sync : WSGroup/5.2.11-SNAPSHOT
set sync.port/node0 = "9000"
attach %node% sync
set sync.master = 'node0'

network node0.ip.lo 10.10.0.1
```

Fig. 6. Default configuration of Field-Node

The second one is a script that will be executed whenever the field node connects to the network. This script adds a few components and connectors so that the field node can execute its logic and connect to the gateway via MQTT. This script is shown in Figure 7. Finally, the third script simply remove the whole FieldNode from the model when the field node disappears from the network. Using those three scripts, the model and the running system are always in sync. Only field nodes that are actually connected to the network will appear in the model.



```
set sync.onConnect = '
add {nodeName}.FieldNode_0 : no.tellu.FieldNode/1.0.17
set {nodeName}.FieldNode_0.FieldNode_sensor_Properties_nodeName__var = "{nodeName}"

add {nodeName}.submqtt : MQTTSubClient/2.0.1
set {nodeName}.submqtt.topic = "sensor/fallIndex"
set {nodeName}.submqtt.host = "10.10.0.1"
set {nodeName}.submqtt.port = "1883"

add {nodeName}.pubmqtt : MQTTPubClient/2.0.1
set {nodeName}.pubmqtt.topic = "{nodeName}"
set {nodeName}.pubmqtt.host = "10.10.0.1"
set {nodeName}.pubmqtt.port = "1883"

add pubchan : LocalChannel
add subchan : LocalChannel

bind {nodeName}.FieldNode_0.Fie_sen_readingJson_out pubchan

bind {nodeName}.pubmqtt.jsonPub pubchan
bind {nodeName}.submqtt.onTopicAndMsg subchan
'
```

Fig. 7. onConnect fragment of Home-GW inHEADS deployment model

D. Summary

The eHealth service implemented by the TellU company involves are rather heterogeneous infrastructure composed of X86, ARM, MIPS and AVR nodes, ranging from 1GHz CPU with 1GB RAM down to 16MHz microcontroller with 2.5 KB RAM. All the code generated for this service (in JavaScript for the larger nodes and C for the smaller nodes) worked out of the box, without any manual modifications. All the libraries TellU needed to use (to interact with Z-Wave, BlueTooth, GPIO on the Intel Edison, etc or to communicate over MQTT) could be integrated with no major issues, either in the HEADS Modelling Language, or directly in the target language as a HEADS component.

A few limitations were noted by TellU. Deploying code on resource-constrained devices can sometimes be cumbersome, as the generated code first needs to be cross-compiled and then uploaded to the device. This limitation could easily be addressed by extending the extension point related to build scripts so that it could also run the scripts using a command line the user can override. By default the C compiler would just run make. When compiling and uploading to Arduino it would execute: `avrdude -CD:avrdude.conf -v -v -v -v -patmega328p -carduino -P.COM22 -b57600 -D -Uflash:w:myProgram.cpp.hex:i`.

Regarding the tooling, TellU was rather satisfied with the integration of the different languages and tools within the Eclipse IDE and simply noted some specific cases where code completion does not yet work on par with e.g. Java code completion. Other than that, the languages and tools are usable.

E. Lessons learned

Several developers at TellU, well acquainted with Java, were able to rapidly get started with the concepts and tools described in this approach. A series of tutorials covering the different concepts, and including exercises to be completed by the participants, was completed within a couple of days [3]. Before implementing the eHealth service, TellU developers have been working on the past few years on the backend service, collecting data and performing analytics. Integrating directly with sensors and gateways was thus a new activity. Most of the learning curve was related to learning how to interact directly with hardware. The HEADS approach actually speed up the process by allowing integrating with C and JavaScript libraries (not popular languages at TellU) without writing advanced C and JavaScript, as most of the logic could be expressed in a platform independent way. In particular, our generative and modular component-based approach for heterogeneous and distributed system allowed and efficient co-development and continuous evolution of this eHealth service. Once interfaces have been defined, it was easy for TellU to define mockups for the components that were not yet implemented and still be able to run the whole system. Mockups were replaced in an iterative way.

Regarding the integration with new languages and platforms, most of the compilers (POSIX C for Linux, Java and JavaScript) have been developed by people heavily involved in the definition of the HEADS Design Language and the Transformation Framework. It is thus hard to conclude on how difficult it is to implement a completely new compiler, other than it took 3000 LoC for Java and JavaScript and about 7000 LoC for POSIX C, which is far way less than writing a "real" compiler (GCC being 14 millions LoC). An experiment conducted with another department at SINTEF with people not involved in the development of the language and framework showed it is possible to extend the C compiler to support another "dialect" for ARM microcontrollers. This took about 2000 LoC and a couple of weeks for the embedded C expert (with only basic Java knowledge) to get this compiler fully functional. While the time to write this compiler could have been reduced e.g. with better documentation, the two weeks spend here were almost entirely saved by developers at TellU who could just get started with this new platform

[3]https://github.com/HEADS-project/training

without building a detailed knowledge of C development on this platform.

V. Related Work

Some attempts of creating an Esperanto of the programming languages have emerged, aiming at replacing the need for using multiple programming languages. For example, Haxe [5] is a programming language that cross-compile to most of the popular programming languages (Java, JavaScript, C++ and others). A significant development effort in Haxe, beyond the development of the language and compilers, is put in the development of a standard library (mostly wrapping the standard libraries of the targeted languages). This standard library needs to be embedded at runtime with a too large overhead to run on the most constrained platforms our approach is able to target.

In the CBSE domain, some component platforms are available for different languages. For example, Fractal [6] is implemented in Java and C. However, no significant effort has been put to make the different platforms to interoperate, those different implementations being isolated rather than being a versatile Fractal. The HEADS approach facilitates, both at design-time and runtime the integration and interoperability of components expressed in different languages.

In the modeling domain, using a multi-viewpoints approaches proved to be profitable. Schmidt *et al* [7] use a modeling approach to generate Distributed Real-time and Embedded Component Middleware and Applications. More recently, in [8], Dabholkar et al highlights the benefits of providing a Generative Middleware Specialization Process for Distributed Real-Time and Embedded Systems. The HEADS approach mainly focuses in enabling a simple integrating of some viewpoints with existing programming language to simplify the complex integration of modern systems.

In the system Engineering community, the Eclipse Polarsys Capella project [4] propose a native support for viewpoint extensions, allowing to extend and/or specialize the core environment to address particular engineering concerns (performance, operating safety, security, cost, weight, product line, etc.), combined with the possibility to carry out multi-criteria analysis of target architectures to help find the best trade-offs [9]. In that direction, the HEADS approach can be compared to the Polarsys project with a focus on the integration with existing programming language.

In the Mobile domain, Cepa *et al* [10] present MobCon a top-down generative approach to create Middleware for Java Mobile Applications, which shows that generation techniques can be effectively used to develop mobile application. In the same direction, Cassou *et al* [11] presents a generative approaches based on software architecture model, associated with verification strategies, to create pervasive applications. Contrary to this work, we do not provide a unique abstract viewpoint to design the system. The HEADS approach provides a specific viewpoint for each stakeholder or each software building stage.

[4]https://www.polarsys.org/capella/

VI. Conclusion

This paper mainly relates an experiment in using a multi-view point approach for designing complex eHealth system. This approach proposes the use of several viewpoints and several modeling languages based on well-established formalisms (a component and state machine-based language for design and implementation, and a configuration language for deployment) dedicated to the integration, deployment and continuous operation of existing libraries and components already available in various languages.

The experiment shows that this approach can be used in an industrial context by a small team of engineering (TellU being an SME with about 10 developers) to create a real eHealth system. This experiment also shows that the use of these modeling languages associated with an extensible generative framework to support new platforms provide a solution to create modular applications that can be developed iteratively and easily reconfigured at runtime.

Acknowledgment

This work was funded by EU FP7/2007-2013 grant n611337, HEADS project (www.heads-project.eu)

References

[1] B. Morin, F. Fleurey, and O. Barais, “Taming heterogeneity and distribution in scps,” in *1st IEEE/ACM International ICSE Workshop on Software Engineering for Smart Cyber-Physical Systems, Florence, Italy, May 17, 2015*, T. Bures, D. Weyns, M. Klein, and R. E. Haber, Eds. IEEE, 2015, pp. 40–43.

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

[3] S. Furuhashi, “Messagepack: Its like json. but fast and small, 2014,” *URL http://msgpack. org*.

[4] F. Fouquet, B. Morin, F. Fleurey, O. Barais, N. Plouzeau, and J. Jézéquel, “A dynamic component model for cyber physical systems,” in *15th ACM SIGSOFT Symposium on Component Based Software Engineering, CBSE 2012, Bertinoro, Italy, June 25-28, 2012*, V. Grassi, R. Mirandola, N. Medvidovic, and M. Larsson, Eds. ACM, 2012, pp. 135–144.

[5] B. Dasnois, *HaXe 2 Beginner's Guide: Develop Exciting Applications with this Multi-platform Programming Language*. Packt Publishing Ltd, 2011.

[6] E. Bruneton, T. Coupaye, M. Leclercq, V. Quéma, and J.-B. Stefani, “The fractal component model and its support in java,” *Software-Practice and Experience*, vol. 36, no. 11, pp. 1257–1284, 2006.

[7] A. S. Gokhale, D. C. Schmidt, T. Lu, B. Natarajan, and N. Wang, “Cosmic: An MDA generative tool for distributed real-time and embedded applications,” in *International Middleware Conference, Workshop Proceedings, June 16-20, 2003, Rio de Janeiro, Brazil*, 2003, pp. 300–306.

[8] A. Dabholkar and A. Gokhale, “A generative middleware specialization process for distributed real-time and embedded systems,” in *Object/Component/Service-Oriented Real-Time Distributed Computing (ISORC), 2011 14th IEEE International Symposium on*, March 2011, pp. 197–204.

[9] J.-L. Voirin and S. Bonnet, “Arcadia: model-based collaboration for system, software and hardware engineering,” in *Complex Systems Design & Management, poster workshop (CSD&M 2013)*, 2013.

[10] V. Cepa and M. Mezini, “Mobcon: A generative middleware framework for java mobile applications,” in *HICSS: 38th Annual Hawaii International Conference on System Sciences*, Jan 2005, pp. 283b–283b.

[11] D. Cassou, E. Balland, C. Consel, and J. Lawall, “Leveraging software architectures to guide and verify the development of sense/compute/-control applications,” in *ICSE'11: 33rd International Conference on Software Engineering*. ACM, 2011, pp. 431–440.