A Modeling Domain-Specific Language for IoT-enabled Operating Systems

T. Gomes*, P. Lopes*, J. Alves*, P. Mestre†, J. Cabral*, J. L. Monteiro*, and A. Tavares*
Corresponding author: mr.gomes@dei.uminho.pt

*Centro ALGORITMI, University of Minho, Portugal
†Engineering Department, University of Trás-os-Montes e Alto Douro, Portugal

***Abstract*—With the increased complexity of low-end devices in the Internet of Things (IoT), mainly due to the connectivity and interoperability requirements, the development and configuration of embedded operating systems (OSes) for such devices is not straight forward. The complexity of the communication requirements is usually mitigated by the OS, e.g., the Contiki-OS, as it already incorporates an IoT-compliant network stack. Yet, the configuration of such stack requires major knowledge on the code structure, leading to additional development time, particularly when the network comprises several wireless nodes and individual configurations with subsequent firmware that needs to be generated. Based on a developed software modeling domain-specific language, this paper presents the EL4IoT framework. It aims to reduce and ease the development time by promoting a design automation tool that can configure, and automatically generate code (ready to compile) for low-end IoT devices running the Contiki-OS. Although leveraging the whole Contiki-OS modeling, this work only refactored and modeled the network stack while approaching the OS itself as one big building block or component. The proposed approach can be extended to other IoT-enabled OSes as well as integrated in other design automation tools.**

***Index Terms*—Domain-Specific Language (DSL), Internet of Things (IoT), embedded Operating Systems, model-driven development**

I. Introduction

The Internet of Things (IoT) materializes the concept of a world wide network, enabling devices and sensors to be wirelessly connected to the Internet [1]. Providing wireless connectivity to ever-growing number of such smart devices, which already integrate a plethora of applications and scenarios, such as smart cities [2], structural health monitoring [3] and smart health care systems [4], while serving a variety of requirements, can be quite challenging. This is due, mainly, to the hardware heterogeneity (which can range from small 8-bit microcontrollers to more complex and energy-efficient 32-bit processors) and the scarce available resources (e.g., processing capabilities, available memory and constrained energy sources). These limitations, as well as the increased complexity of connecting smart things to the Internet seamlessly through standard and well-known protocols, requires a standardized and adaptive communication stack [5]. Such protocol stack defines standards for all layers, e.g., the IEEE 802.15.4 standard (layer 2) and IPv6 over Low power Wireless Personal Area Networks (6LoWPAN) protocol (layer 3), promoting interoperability and connectivity among all the participating devices. From the hardware point of view, traditional operating systems (OSes) cannot fulfill the requirements of providing a standardized and lightweight, yet complex, network stack to be deployed on the constrained low-end IoT devices [6]. As an alternative, available embedded OSes (which already integrate an IoT-compliant and standardized network stack), such as Contiki-OS [7] and RIOT-OS [8], provide support to a wide range of hardware platforms and microcontroller (MCU) architectures. They run in traditional homogeneous wireless sensor network (WSN) nodes, as well as in heterogeneous architectures for low-end IoT devices [9], enabling field-programmable gate array (FPGA) technology to be exploited for accelerating network and sensing-related tasks [10].

Despite promoting a lightweight implementation, the configuration and deployment of these embedded OSes is still complex, mainly due to the hardware heterogeneity and the high variability of the OS and network stacks. The task of configuring and customizing network parameters, such as the personal area network (PAN) and device's addresses (MAC and IPv6), as well as OS services and protocols, e.g., 6LoWPAN and Constrained Application Protocol (CoAP), can be mitigated by enabling design automation through the development of a tool that allows full system configuration and code generation according to the user needs and application requirements. The applicability of such tool can be explored in a way that generating firmware for several nodes in an IoT network, while providing mechanisms for code verification and validation, can be performed by automated systems and/or embedded systems designers without deep knowledge of the OS or the IoT communication stack. Several approaches targeting design automation by providing a domain-specific language (DSL) to model a desired system have already been undertaken in the recent years. In [11] it is proposed a low-level DSL for dynamic code-generation in binary translation systems, enabling code snippets to be added during compile time. The code is then generated by the translator on demand at runtime and integrated into the translated application code. SensorScript [12] proposes a business-oriented DSL for sensor networks, that aims to provide a model which avoids to overwhelm any user with all the data gathered from the sensor network, regardless if it is actually required by the user. This allows users to easily search, query and aggregate information from the sensors. Targeting IoT-based applications and aiming

978-1-5386-1127-2/17/$31.00 ©2017 IEEE

to relieve designers from the complexity and the heterogeneity of the WSN nodes, the DSL-4-IoT [13] was also proposed. It mainly provides a visual model based language, which, using a high level of abstraction, allows different configurations to be deployed over a WSN environment. However, at the level of abstraction provided DSL-4-IoT, the low-level configurations such as the network settings and protocols are not specified.

In this paper we propose the EL4IoT framework, a modeling DSL for IoT-enabled OSes, allowing the configuration and automatic generation of code for low-end devices in IoT applications that require an IoT network stack to provide interoperability and seamless connectivity to the Internet. The main contributions of the EL4IoT are: (1) a DSL for modeling embedded OSes targeting the Internet of Things (Contiki-OS); (2) reduced modeling efforts over the Contiki-OS network stack; and (3) the development of a design automation and code verification tool for embedded systems designers, promoting its integration with design automation tools.

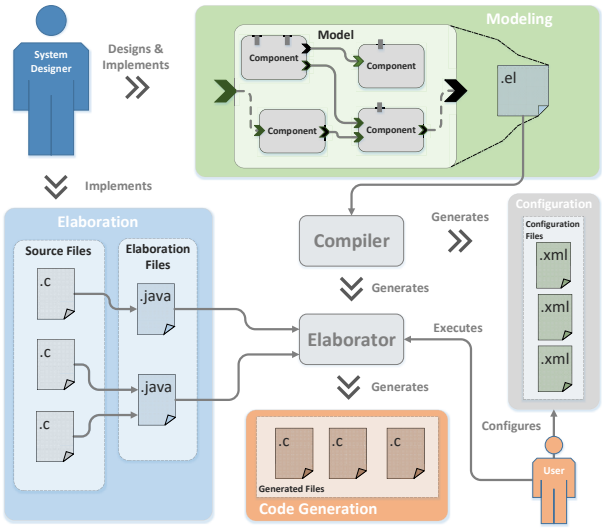

Figure 1. Framework workflow

II. Elaboration Language (EL)

A. Background

A domain-specific language (DSL) is a programming language with limited expressiveness which, in contrast with general purpose languages, targets a specific domain providing constructs to solve its specific problems [14]. Its usage it is quite appealing since it promotes a simpler and faster development, while providing higher gains in expressiveness, ease of use and productivity [15]. Developing DSLs is quite hard, as it requires high levels of domain knowledge and technical expertise, but once well designed and created, it tends to pay off all the inherent development efforts [16]. There is a growing interest in DSLs for generative programming (GP) [17] and model driven development (MDD) [18] programming styles, as they provide higher levels of abstraction, leveraging software reuse and fast software development. While GP targets the automatic system generation according to a defined specifications, MDD is an approach in which extensive models are created (before, during or after source code is written) to describe system's architecture abstracting implementation details, easing development and testing purposes. Both MDD and GP rely on software reuse and complex code generation, thus, a DSL must provide constructs to enable the mapping between models and code that will be generated [19].

The Elaboration Language (EL) is a modeling DSL designed to be an efficient GP tool, while approaching MDD. Based on the Service-Component Architecture (SCA) standard [20], it mainly targets the code generation automation from the source files of a designated system. SCA specifies that various system components may be assembled by the form of service-oriented architecture (SOA) components following a composite pattern. The key elements of this standard are: Composite, Component, Service, Reference, Property and Wire. According to the standard, it is possible to create reference architectures identifying the system components and interactions between them, as well as their configurable properties. Our EL was developed using the Xtext and Xtend frameworks, widely adopted when developing DSLs. Xtext is an Eclipse framework used to create the language grammar, which dictates how the parser and the Abstract Syntax Tree is created, while Xtend is a general purpose programming language that is translated into a comprehensible Java representation [14]. It is interchangeable with Java code, and it is used to implement language validators, code generation software, and some other Eclipse language specific tools (e.g., quick fixes). In this paper, we do not cover the aspects of developing a DSL, as we mainly intend to technically present our EL as a solution for software modeling, validating its usability through a real use case, that is, the modeling of the Contiki-OS communication stack.

B. System overview

The framework workflow, depicted in Figure 1, encompasses four main stages: Modeling, Elaboration, Configuration and Code Generation. During each stage, the artifacts that will be used in the next stages are created. During the Modeling stage, the main goal is to create an architectural model, according to the SCA standard, that will be later used as a reference architecture. During the architectural model creation the system components must be identified, as well as the dependencies between them, allowing the specification of well-defined interfaces and properties. After its creation, it must be compiled. If the compilation process succeeds, the compiler should generate, for each component, its Java representation, elaboration stub class, the configuration XML files, and an architecture-specific Java Elaborator. All these artifacts are then used in the following stages. The Elaboration stage encompasses the provision of annotated source files and the implementation of the elaboration classes' behavior (using the elaboration stub classes). Once implemented, elaboration files dictate how the source code must be generated. For each component, more than one implementation may be available, as well as its respective elaboration. Only one elaboration class

per component will be executed in the Elaborator, as specified by the configuration files. Also, an API is available to ease the annotation process (find and replace), within the respective source code files (e.g., function calls, property values), and also to fetch other properties from the configuration files. Previously generated XML artifacts (configuration files), contain the values for all the component properties, which may be changed during the Configuration stage, to modify the system configuration and its subsequent code generation. These files also specify which component elaboration will be loaded into the Elaborator process. In addition, each elaboration may have its own implementation-specific properties. Since such properties are not available in the reference architecture's configuration files, another XML file should be provided by the elaboration developer. Once properly configured, the generated elaborator must be executed. This process will fetch each component's properties and will load elaboration classes through Java reflexion, according to which rules are specified in the XML files. As the result, the Code Generation should be according to corresponding Elaboration Classes.

Three different actors, that interact with the system at different stages of the code generation process, are identified:

- The Architect: the individual with technical knowledge and specific domain expertise, that is responsible for translating system characteristics into a model;
- The Component Designer: the individual with technical expertise, which provides the annotated source code files and implements the elaboration classes.
- The End User: the final user that will benefit from the provided resources to configure and generate application-specific code. Usually, the end user only focus in setting up the configuration files (defining properties values and choosing the elaboration file to be imported) before invoking the Elaborator.

C. EL's Constructs and Operations

As previously said, the EL is a DSL that allows the description of an architectural model according to SCA specified elements. An EL file (depicted by Listing 1), defined with .el extension, contains three types of top level constructs: components, interfaces and language descriptions. Each component is described as an aggregation of subcomponents, properties, and its relations with other components in the form of services and/or references. It is also described as having one language type, which should be its own implementation language.

1) Language construct: The Language construct specifies the implementation language of components (e.g., C, C++ or Python), imposing restrictions to a given component, which can only relate with others described in the same language.

2) Interface construct: The Interface construct describes interfaces for which other components will associate. Components connect through bindings of services and references that follow the same interface type, using the keyword `bind`. An interface declaration must always state the services it can provide.

3) Component construct: The Component construct is the most important and it consists of four sections:

- Properties: EL enables properties declaration for the basic programming types (i.e., char, int, string), in which it is possible to impose restrictions and to perform assignment operations.
- Subcomponents: Where the aggregated subcomponents are stated.
- Services and References: Here, interfaces are instantiated as services or references. The interfaces belonging to the Services section are those implemented by the component, while the ones inherent to the references section are dependencies from a given component that implements that interface. Services and references must always be connected through binding operations.
- Free: In this section, assignments can be made to properties where subcomponent interfaces are binded and/or promoted. This is done (either for services or references) by using the keyword `promote`. It states that a subcomponent interface is going to be available in the top-level component, and it must be resolved later with a binding to the top-level component. Other important keyword is the `compile`, which states the top-level component on the hierarchy, from where the compilation process is started (in a top-down approach). This also defines the order of the classes invocation in the Elaborator program.

```
1  final ("TransportLayer.java") component TransportLayer (C) {
2      properties:
3          bool Enable_UDP : true
4          bool Enable_TCP : true
5      subcomponents:
6          UDP UDP1
7          TCP TCP1
8      services:
9          I_tcp_ip_input      S_TCP_IP_Input
10         I_TCPIP_UIP_Call    S_TCPIP_UIP_Call
11         I_TCPIP_ICMP6_Call S_TCPIP_ICMP6_Call
12     references:
13         I_Timer_API R_Timer_API
14     /*UDP subcomponent */
15     promote reference UDP1.R_TCPIP_Output as R_TCPIP_Output
16     promote reference UDP1.R_UIPProcess   as R_UIPProcess
17     ...
18 }
```

Listing 1. EL code snippet from the component *TransportLayer*.

III. Modeling the IoT stack

Modeling an IoT network stack is not straight forward, mainly due to its configuration complexity and inherent variability. In this paper, the Contiki-OS network stack (called μIP stack) was selected, however others could have been studied, e.g., RIOT-OS. Due to its complexity, it requires a high level of knowledge on IoT-enabled network stacks for low-end devices as well as the OS itself. The μIP stack is composed by tightly-coupled components which must be priorly identified by performing source code analysis aided by the respective simulation. The identification of component interfaces and their interactions (through function invocation) requires a deep understanding of the stack implementation and its regular behavior. The first steps encompass the creation of

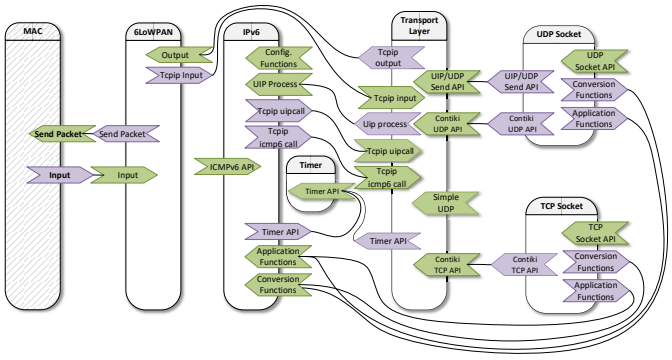

Figure 2. High-level composite model of the Contiki network stack implementation.

a reference architecture of the network stack, followed by a comprehensive description of abstraction, lowering from the model to source code, extending as well the Contiki-OS stack documentation.

Reference Architecture: Figure 2 depicts the top-level view of the resulting model, comprising the top-level components in a composite model, where each block refers to a component with well-known denomination. Such view is expected since we are modeling the network stack, prior further development. The purple and green polygons define references and services, respectively, in a SOA approach. Semantically, the connectors between references and services denote establishment of a function call dependency between components, where green polygons provide the services required by references. The proposed model follows a layered approach, where top components are formed by other components as well. This approach inherently provides a changeable level of abstraction, depending on the embedded designer needs. For the sack of simplicity, we will focus on the *Transport Layer* component, which internals are depicted in Figure 3. This component is composed by a UDP and TCP subcomponents, with their respective interfaces and properties. A promote relationship is depicted by the dashed lines. In this example, we provide visibility to the internal services and references from outside the Transport Layer component. Logically, this component uses all the modeled OS services available (to applications) and, therefore, encompasses all its available references. Properties are application-specific and cannot be represented in the reference model.

IV. Implementation

A. Model Representation

After being conceived, the model is translated to the EL DSL. For instance, the model representation depicted by Figure 3 is translated into EL DSL code present in Listing 1. Its code representation allows the generation of the supporting software, which consists on a framework that accesses model properties and interfaces during the final stage of code generation. This framework aims to promote design automation with code generation, leaving only the components selection

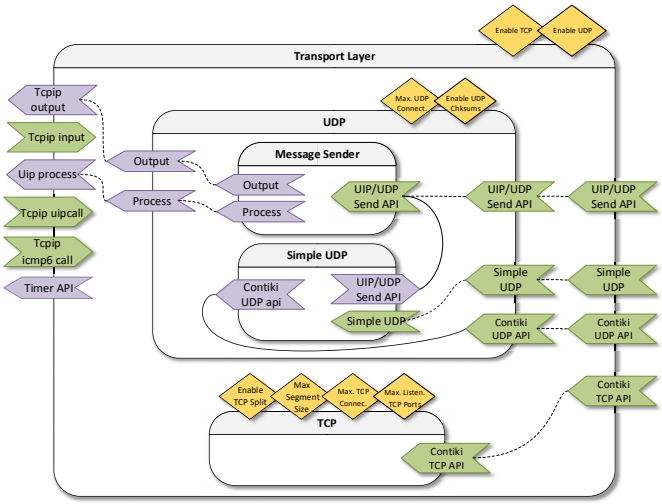

Figure 3. Internals of the Transport Layer component from the High-level composite model.

and properties configuration, e.g., PAN address and TCP/UDP parameters, to the user's choices.

B. Elaboration

1) Properties: The EL seamless code generation process allows a transparent system configuration. The modeled system properties are defined in a configuration XML file, according to the user and application requirements. While EL avoids the traditional need for developers to write code, the generation of the final OS source files according to the model definition and configuration must be automatic and user-friendly. As aforementioned, EL uses software annotations, embedded in the OS source files and defined by the meta-character `'@@'`. This pattern is used to help in finding code where the annotation is later replaced by its corresponding value during the code generation process. Such annotations and associated complexity are dependent on the OS implementation, thus may vary according to the code size and system variability.

```
1 //======================== Transport
      ========================
2 //----------- TCP
3 #define UIP_CONF_MAX_LISTENPORTS
      @@UIP_CONF_MAX_LISTENPORTS@@
4 //----------- TransportLayer
5 #define UIP_CONF_TCP              @@UIP_CONF_TCP@@
6 #define UIP_CONF_UDP              @@UIP_CONF_UDP@@
```

Listing 2. UIP_CONF_TCP and UIP_CONF_UDP annotations.

Listing 2 depicts a code snippet with `UIP_CONF_TCP` and `UIP_CONF_UDP` annotations, which are used to enable TCP and UDP protocols, associated with the `Enable TCP` and `Enable UDP` model properties, respectively. The `UIP_CONF_MAX_LISTENPORTS` is defined by the TCP component inside its parent (Transport Layer) component, as depicted in Figure 3, leveraging the ability of internal components to define their own properties.

The elaboration of components incorporates the logic of their respective code generation process (i.e., annotation sub-

stitutions) in the form of JAVA code. The component Transport Layer only contains the original Contiki-OS implementation, embedded in an elaboration. Further elaborations for this and other components of the stack are still under development. Listing 3 denotes the logic associated with the Transport Layer component elaboration. Briefly, a header (shared between several components) is opened and the `UIP_CONF_TCP` and `UIP_CONF_UDP` annotations are replaced by the values defined in the model, retrieved from the configuration XML file.

```
openAnnotatedSharedSource("contiki-conf-gen.h");

if(target.get_Enable_TCP())
    replaceAnotation("UIP_CONF_TCP", 1);
else
    replaceAnotation("UIP_CONF_TCP", 0);

if(target.get_Enable_UDP())
    replaceAnotation("UIP_CONF_UDP", 1);
else
    replaceAnotation("UIP_CONF_UDP", 0);

openAnnotatedSource("tcpip.c", "./contiki-3.0/core/net/ip");
openAnnotatedSource("tcpip.h", "./contiki-3.0/core/net/ip");
```

Listing 3. Elaboration method of the Transport Layer component in Java.

Next, the source code of the implementation is generated in the final directory by calling the "openAnnotatedSource" method. The new generated file will be create after the annotations' replacement. All these functionalities are provided by an API which eases the elaborator's implementation. The generate method is automatically invoked by the framework, while a fully configured and ready-to-compile Contiki-OS stack is being generated.

2) Interfaces: Contrary to the "Properties" of the code generation (which is entirely a model-based process), interfaces use components' elaborations to provide services (as function calls) to connected references. That is to say, different components' elaborations (for the same component) might provide distinct implementations for the same service, represented by several functions calls. The implementation of Interfaces, including argument's meta-data passing, is still under development.

V. Evaluation

The Contiki-OS network stack model is used to demonstrate the flexibility provided by the modeling tool to automatically generate a full configured system ready to be compiled. While the reference model provides properties' abstraction from implementations by using components, easing the system configuration to users, it requires the specification of a real implementation by the system designer. Listing 4 denotes the XML configuration file for the Transport Layer component. Our implementation of this component is provided by the `MySpecificTransportLayerElaborator`. Each Transport Layer sub-component has its own implementation. For instance, by disabling the TCP protocol, the final generated Contiki-OS code will not contain TCP related code.

The same procedure is used to specify the implementation of the `Application` component to be a UDP Server. This implementation provides its own configuration XML file, allowing the designer to specify implementation-related properties as well. In both XML files, every property requires a default value to seamlessly generate the final (compilable) Contiki-OS source code.

```
<component type="TransportLayer">
  <elaboration
     default="SpecificTransportLayerElaboratorTemplate">
  MySpecificTransportLayerElaborator
  </elaboration>
  <properties>
    <property type="bool" name="Enable_UDP" default="true">
      <value>
        <element></element>
      </value>
    </property>
    <property type="bool" name="Enable_TCP" default="true">
      <value>
        <element>false</element>
      </value>
    </property>
  </properties>
</component>
```

Listing 4. XML configuration file for the Transport Layer component.

The UDP server application is configured to create a connection with any UDP client, listening on the UDP port 4101 (instead of the default port 3001). The default response given by the server to any client was not changed by our tool. While in this example (Listing 5) the server is configured to be listening on UDP port 4101 (TCP related code was disabled from compilation), several distinct configurations can be achieved by using other model properties (not depicted in this test).

After its generation, the source code for the UDP server was compiled and the resulting firmware was deployed on a CC2538EM board from Texas Instruments. This Evaluation Module (EM) was used in a simple client-server setup in order to test our developed tools. The server, using the same EM, is running a UDP client application that periodically exchanges messages with the automatically generated and configured (by our tool) UDP server. As it can be seen from Figure 4, which depicts the UDP server application output, received by the UDP client application. The client connects the specified UDP server ports and receives the default message, which is printed to the EM output.

```
<component type="Application">
    <properties>
        <property type="int" name="LocalPort"
     default="3001">
            <restriction type="range">
                <botValue>1</botValue>
                <topValue>5000</topValue>
            </restriction>
            <value> <element>4101</element> </value>
        <property type="string" name="Message"
     default="Automatically configured server!">
            <value> <element></element> </value>
        </property>
        <property type="string" name="Hostname"
     default="contiki-udp-server">
            <value> <element></element> </value>
        </property>
    </properties>
</component>
```

Listing 5. UDP Server application specific XML configuration file.

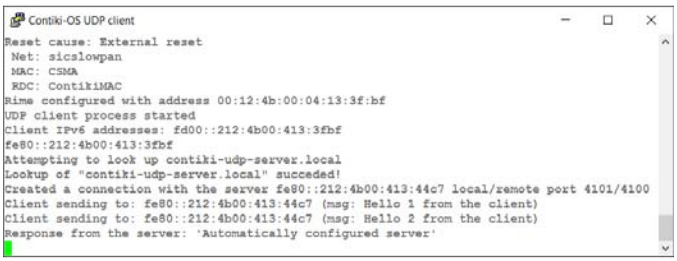

Figure 4. A simple UDP client connecting and exchanging messages with the automatically generated UDP Server.

VI. Conclusions and Future Work

This paper presented the EL4IoT, a modeling DSL for embedded Operating Systems that targets the low-end devices in the Internet of Things. We have modeled the Contiki-OS network stack and developed a DSL that allows the whole OS description, mainly its companion network stack, with a variable abstraction level supported by the composite pattern. Our developed framework aims to promote design automation, mitigating the configuration task of the Contiki-OS when deploying an IoT-compliant network through automatic code generation. For the proof of concept, we have modeled the Contiki-OS network stack into our DSL and, using the framework, generated a UDP server application, easily (re)configured without manually changing the source code. The obtained results show that, despite being hard to create a DSL for a specific domain, the modeling efforts tend to pay off. With the increasing level of complexity and variability of IoT systems, these tools result in higher productivity and lower development time.

The benefits from such approaches in developing embedded software are endless. Hereafter, we plan to improve our EL4IoT framework, namely the code generation tool, with a more user friendly interface that allows code generation and system configuration with reduced number of inputs from the user. Other features can be added to this tool, such as the automatic creation of the final firmware from the generated code. Currently under the development, we are improving our DSL with the usage of semantic technology to describe the domain knowledge, leveraging the development towards a model validation approach and reducing the elaboration development efforts. This will also contribute in improving the system scalability, as semantic technology can upgrade code generation and verification when used to guide the development environment.

VII. Acknowledgments

This work has been supported by COMPETE: POCI-01-0145-FEDER-007043 and FCT - *Fundação para a Ciência e Tecnologia* within the Project Scope: UID/CEC/00319/2013. Tiago Gomes is supported by FCT PhD grant SFRH/BD/90162/2012.

References

[1] L. Atzori, A. Iera, and G. Morabito, "The Internet of Things: A Survey," *Comput. Netw.*, vol. 54, no. 15, pp. 2787–2805, Oct. 2010.

[2] A. Zanella, N. Bui, A. Castellani, L. Vangelista, and M. Zorzi, "Internet of Things for Smart Cities," *IEEE Internet of Things Journal*, vol. 1, no. 1, pp. 22–32, Feb 2014.

[3] T. C. Arcadius, B. Gao, G. Tian, and Y. Yan, "Structural Health Monitoring Framework Based on Internet of Things: A Survey," *IEEE Internet of Things Journal*, vol. PP, no. 99, pp. 1–1, 2017.

[4] L. Catarinucci, D. de Donno, L. Mainetti, L. Palano, L. Patrono, M. L. Stefanizzi, and L. Tarricone, "An IoT-Aware Architecture for Smart Healthcare Systems," *IEEE Internet of Things Journal*, vol. 2, no. 6, pp. 515–526, Dec 2015.

[5] M. R. Palattella, N. Accettura, X. Vilajosana, T. Watteyne, L. A. Grieco, G. Boggia, and M. Dohler, "Standardized Protocol Stack for the Internet of (Important) Things," *IEEE Communications Surveys Tutorials*, vol. 15, no. 3, pp. 1389–1406, Third 2013.

[6] O. Hahm, E. Baccelli, H. Petersen, and N. Tsiftes, "Operating Systems for Low-End Devices in the Internet of Things: A Survey," *IEEE Internet of Things Journal*, vol. 3, no. 5, pp. 720–734, Oct 2016.

[7] A. Dunkels, B. Gronvall, and T. Voigt, "Contiki - a lightweight and flexible operating system for tiny networked sensors," in *Local Computer Networks, 2004. 29th Annual IEEE International Conference on*, Nov 2004, pp. 455–462.

[8] E. Baccelli, O. Hahm, M. Gunes, M. Wahlisch, and T. C. Schmidt, "RIOT OS: Towards an OS for the Internet of Things," in *2013 IEEE Conference on Computer Communications Workshops (INFOCOM WKSHPS)*, April 2013, pp. 79–80.

[9] A. de la Piedra, A. Braeken, and A. Touhafi, "Sensor Systems Based on FPGAs and Their Applications: A Survey," *Sensors*, vol. 12, no. 9, pp. 12 235–12 264, 2012.

[10] T. Gomes, S. Pinto, F. Salgado, A. Tavares, and J. Cabral, "Building IEEE 802.15.4 Accelerators for Heterogeneous Wireless Sensor Nodes," *IEEE Sensors Letters*, vol. 1, no. 1, pp. 1–4, Feb 2017.

[11] M. Payer, B. Bluntschli, and T. R. Gross, "LLDSAL: A Low-level Domain-specific Aspect Language for Dynamic Code-generation and Program Modification," in *Proceedings of the Seventh Workshop on Domain-Specific Aspect Languages*, ser. DSAL '12. New York, NY, USA: ACM, 2012, pp. 15–20.

[12] A. Garnier, J. M. Menaud, and R. Pottier, "SensorScript: A Business-Oriented Domain-Specific Language for Sensor Networks," in *2015 3rd International Conference on Future Internet of Things and Cloud*, Aug 2015, pp. 44–49.

[13] A. Salihbegovic, T. Eterovic, E. Kaljic, and S. Ribic, "Design of a domain specific language and IDE for Internet of things applications," in *2015 38th International Convention on Information and Communication Technology, Electronics and Microelectronics (MIPRO)*, May 2015, pp. 996–1001.

[14] L. Bettini, *Implementing Domain-Specific Languages with Xtext and Xtend*. Packt Publishing, 2013.

[15] M. Mernik, J. Heering, and A. M. Sloane, "When and How to Develop Domain-specific Languages," *ACM Comput. Surv.*, vol. 37, no. 4, pp. 316–344, Dec. 2005.

[16] M. Fowler, *Domain Specific Languages*, 1st ed. Addison-Wesley Professional, 2010.

[17] K. Czarnecki, *Overview of Generative Software Development*, J.-P. Banâtre, P. Fradet, J.-L. Giavitto, and O. Michel, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2005.

[18] U. Zdun, "Concepts for model-driven design and evolution of domain-specific languages."

[19] J.-P. Tolvanen and M. Rossi, "MetaEdit+: Defining and Using Domain-specific Modeling Languages and Code Generators," in *Companion of the 18th Annual ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications*, ser. OOPSLA '03. New York, NY, USA: ACM, 2003, pp. 92–93.

[20] J. Marino and M. Rowley, *Understanding SCA (Service Component Architecture)*, ser. Independent Technology Guides. Pearson Education, 2009.