Model-Driven Engineering Approach for Simulating Virtual Devices in the OSATE 2 Environment

Fáber D. Giraldo and Mónica M. Villegas

***Abstract*—Simulating devices while developing software for embedded systems is advantageous if physical elements such as sensors are not available. In this work, we present the design and implementation of an extension named *Custom Simulation of Virtual Devices (CSVD)* for the Eclipse-based platform that has full support for the *AADL* meta-model, *OSATE 2*. The *CSVD* extension uses a *Model-driven* engineering approach to simulate virtual devices, which allows capturing an *AADL* model of a system that represents the connection between a local network of devices and a gateway through a serial bus, simulating its behavior at a data transmission level by being executed inside the *QEMU* emulator. Also, we present an example of our approach, based in a system modeled in *AADL* and its simulation using the *CSVD* extension.**

***Index Terms*—AADL model, OSATE 2, QEMU, simulation.**

I. Introduction

While developing software for embedded systems, sometimes is required to have devices for testing data transmission, support of a protocol, or some data to validate some software developed. In this work, we describe the design of a plugin named *Custom Simulation of Virtual Devices (CSVD)* developed as an extension to the *OSATE 2* platform, which supports the simulation of serial devices connected to a gateway (emulated using *QEMU*) through a serial bus. This extension allows *OSATE 2* to have a custom simulation system integrated with the *QEMU* emulator, a feature that supports to be extended for future development of another kind of protocols or devices.

The primary approach of the *CSVD* extension developed and added to *OSATE 2* is to analyze an *AADL* model, obtain the configuration of the devices modeled and execute the simulation of those devices inside *QEMU* emulator which emulates an *ARM*-based embedded system.

This paper is organized as follows: Section II explains some of the related work found in the literature. Section III describes the purpose of *AADL* and its usage under *OSATE 2*. Section IV explains how the *CSVD* extension was designed, developed and implemented for its integration with the Eclipse-based platform *OSATE 2*. Section V shows a demonstration of our approach in which a system is designed, modeled in *AADL* and then simulated using the *CSVD* extension. Section VI concludes the paper. And finally, Section VII provides some future work proposed for this project.

Manuscript received September 23, 2017; revised December 12, 2017. This work was supported in part by the SINFOCI Research Group at the University or Quindío, Colombia.

Fáber D. Giraldo and Mónica M. Villegas are with the University of Quindío, Armenia, Quindío, Colombia (e-mail: fdgiraldo@uniquindio.edu.co, mmvillegasa@uqvirtual.edu.co).

II. Related Work

In [1] authors describe an approach for the modeling, verification, and implementation of *ARINC653* systems using *AADL*. It describes a modeling approach exploiting the new features of *AADL* version 2 for the design of *ARINC653* architectures. In [2] authors propose an approach for the verification of the *AADL* architecture, which is assisted by a toolchain and defining a source meta-model for *AADL* and a target meta-model for the timed automata formalism; authors also define a transformation process in two steps, a *Model2Model* transformation and a transformation of a *Model2Text*. In [3] authors present a verification tool called ABV tailored for *AADL* models with a behavioral annex, in which given an architecture defined in *AADL* and its behavior specified in the associated language, their tool model-checks the latter against the requirements specified in Computation Tree Logic (CTL). In [4] authors propose to extend the capabilities of *AADL-OSATE* modeling notation and supporting toolset with Petri nets in a manner that facilitates formal analysis to verify the absence of deadlock and/or livelock phenomenon.

The difference between our project and the related works found is that the extension described in our work is developed to simulate serial devices connected to a gateway and which at the same time is integrated with *QEMU* emulator, features that allow an extensibility for another kind of simulations being executed using *QEMU* emulator.

III. Modeling with AADL

First, we need to have clear what modeling is in this context, a concept explained in [5] and which says that modeling seeks to explain the behavior of a complex system via a model that abstracts it.

The *Architecture Analysis & Design Language* (*AADL*) is a unifying framework for *model-based* software systems engineering. *AADL* can be used to capture the static modular software architecture, the runtime architecture regarding communicating tasks, the computer platform architecture on which the software is deployed, and any physical system or environment with which the system interacts [6]. *AADL* is also an *SAE International (Society of Automotive Engineers) standard [AS5506A]*.

AADL provides concepts such as *threads*, *processes*, and *devices*, which allows doing a formal analysis of systems. In an *AADL* model can be represented the architecture of a

DOI: 10.7763/IJMO.2017.V7.606

system as a hierarchy of interacting components, organizing interface specifications and implementation blueprints of software, hardware, and physical components into packages to support large-scale developments [6].

Models in *AADL* are composed of *AADL components*, *AADL properties*, and *graphical AADL notations*. For components, AADL describes a taxonomy of model elements which can be *component category*, *component type*, *component implementation*, *packages* and *property sets*. Component categories can be of the type *data*, *thread*, *thread group*, *subprogram*, *process*, *memory*, *bus*, *processor*, *device*, *virtual processor*, *virtual bus*, *system* and *abstract*. Components can have features of type *data port*, *event (data) port*, *port direction*, *requires access* or *provides access*. For properties, there are predefined property sets such as *deployment_properties*, *thread_properties*, *timing_properties*, *communication_properties*, *memory_properties*, *programming_properties*, *modeling_properties*, and *aadl_project*. For graphical notations, *AADL* provides the ones illustrated in Fig. 1, which allows representing graphically the model of a system.

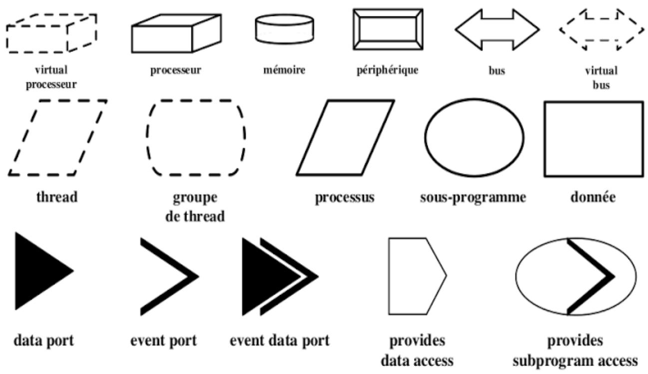

Fig. 1. Types of graphical AADL notations [7].

For the implementation of *AADL* models, *OSATE 2* can be used. *OSATE 2* is an Eclipse-based platform that has a full support of the *AADL* meta-model and which can be extended in functionalities. This platform relies on *EMF*, *UML2* and *XText*, and its core by itself provides the fundamentals to use the basics of *AADL* (textual and hierarchical editor, instantiation of the model) and build *OSATE 2* plugins [8].

One way to extend *OSATE 2* is developing and integrating Eclipse plugins, which can use all the resources from the Eclipse platform and at the same time, use the elements of the *OSATE* core to process and analyze *AADL* models.

IV. Extending OSATE 2

In this work, we extend *OSATE 2* by developing and integrating a plugin that allows processing an *AADL* model with the purpose of generating files and codes from its analysis as resources that are required to launch the simulation of the modeled system by being executed using the *QEMU* emulator. The launch parameters of the *QEMU* emulator, such as *Operating System (OS)* image, kernel image, *CPU* type and *RAM* size need to be configured in the *AADL* model of the system designed.

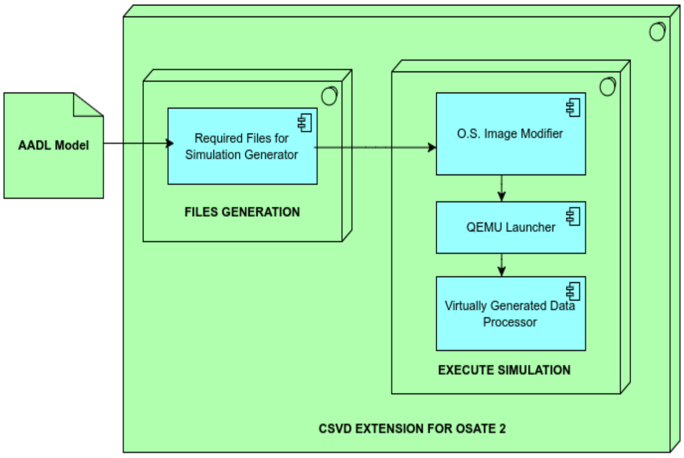

Fig. 2. Structure of the CSVD extension for OSATE 2.

The structure of the *CSVD* plugin is shown in Fig. 2, which also describes the process of executing a simulation. The usage of the *CSVD* plugin consists of loading an *AADL* model through a file search, the execution of the files generation after the *AADL* model is loaded and then, the execution of the simulation if required files were successfully generated. Fig. 3 shows the *graphical user interface (GUI)* of the *CSVD* plugin, allowing to identify the simple usage of the extension.

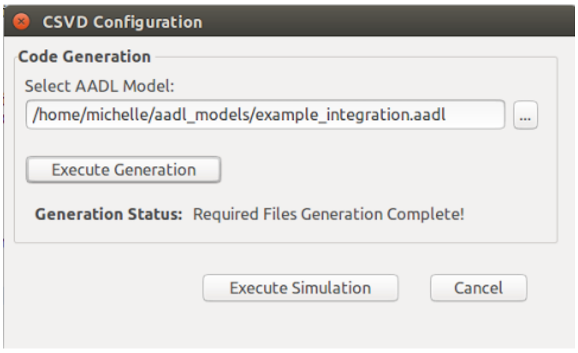

Fig. 3. CSVD GUI.

For the required files generation, the *CSVD* plugin executes a *Python* script that analyzes the *AADL* model and then, based on some rules specified in a *JSON* file which links *AADL* with custom syntax (that can be specific commands or codes) generates the required files as shown in Fig. 4.

On a side, the required files generator, generates a text file containing the specifications of the devices that are going to be simulated, and on the other side, it creates the script that unpacks/configures/repacks the *OS* image, and which contains the command that launches the *QEMU* emulator.

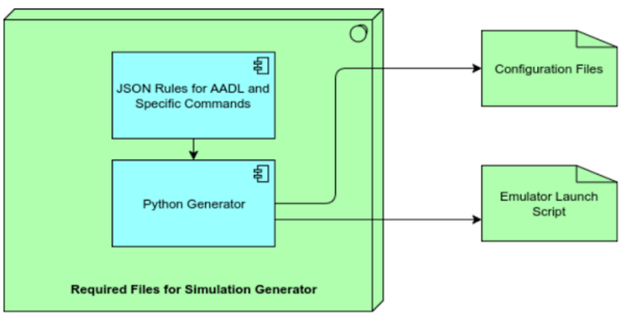

Fig. 4. Structure of the 'required files for simulation' generator.

After the generation of the required files finishes successfully, the execution of the *QEMU* emulator with the simulation system inside is launched.

Before *QEMU* is executed, there is an *OS* image configuration process, which consists of unpacking the image, adding the generated and required files to it, configure it as needed and then, repack it to be executed with *QEMU*. The structure of this process is shown in Fig. 5.

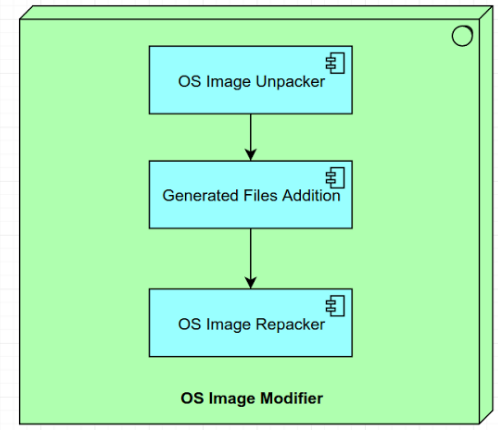

Fig. 5. Structure of the OS image modifier.

After *QEMU* is executed with the modified *OS* image, the simulated devices send data to the serial port that was created in the emulated platform, allowing software processes to capture serial data packages, which contain besides the data, the specified identifiers of each modeled device for proper identification.

V. Demonstration of Our Approach

This demonstration is based on the design of a system that is composed of 3 sensors connected to a serial receiver, which through a serial bus is connected to a gateway sending serial data received from the sensors, as shown in Fig. 6. Fig. 7, 8, 9 and 10 show the *AADL* text model of the system designed. Fig. 7 shows the specification of the entities imported for the usage of memories, *ARM* processors and *UART*.

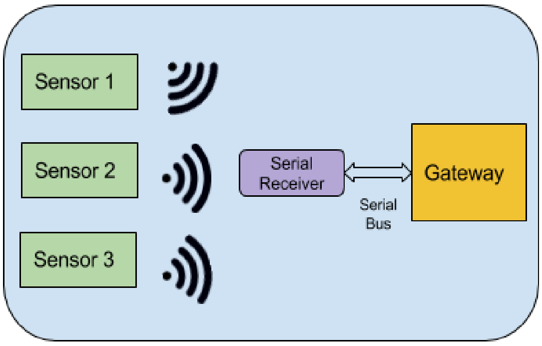

Fig. 6. System for the demonstration.

```
package example_integration
public
    with Memories;
    with Processors::ARM;
    with Buses::UART;
```

Fig. 7. Importing AADL entities.

Fig. 8 shows the declaration of the systems that describe the desired design. For this demonstration, the design is based on two systems, a first system named *top_system* that contains all the components and subsystems, and a second system called *gw_system* which includes the specifications of the gateway. This second system must be specified with the extra parameters of the *OS* image path, *CPU* type, and kernel image path because these are specifications required by *QEMU* and are not currently supported by *AADL*, so one of the ways to specify them is as annex types.

```
system top_system
end top_system;

system gw_system
    features
        bus_serial:requires bus access SerialConnection;
    annex extras {** qemu_kernel : "qemu-kernel/kernel-qemu" ;
        qemu_os : "os_linux_stable.img" ; qemu_cpu : "arm" ;
    **};
end gw_system;
```

Fig. 8. Declaration of the systems for the AADL model designed.

Fig. 9 shows the declaration of the devices and the serial bus. In this case, there are two types of devices, sensors and a serial receiver which is named as serial radio. The sensor devices are configured with a *MAC* address as the identifier; mean and deviation values that will be used during the simulation for the random data generation; and a period value, which represents how often the data will be transmitted to the serial receiver from the sensors. Also, in the declarations must be included the features of each device, in which for this case, it specifies that each sensor will have a data port as output.

For the serial receiver, the features specify the data port connections as inputs, to which the sensors will be connected, and the serial bus requirement used for the connection with the gateway. The data ports specified for all the devices will be used in the implementation of the model for the interconnection between devices and systems.

```
device Sensor1
    features
        data_pk_out:out data port;
    properties
        Period=>2000ms;
    annex extras {** MAC = "20DE55AB2051C123"; mean = "20"; deviation = "20" **};
end Sensor1;

device Sensor2
    features
        data_pk_out:out data port;
    properties
        Period=>2000ms;
    annex extras {** MAC = "20EF50AE2071CF03"; mean = "10"; deviation = "20" **};
end Sensor2;

device Sensor3
    features
        data_pk_out:out data port;
    properties
        Period=>3000ms;
    annex extras {** MAC = "20EF500E4081CF03"; mean = "10"; deviation = "20" **};
end Sensor3;

device SerialRadio
    features
        data_pk_sensor1:in data port;
        data_pk_sensor2:in data port;
        data_pk_sensor3:in data port;
        bus_serial:requires bus access SerialConnection;
end SerialRadio;

bus SerialConnection extends Buses::UART::UART
end SerialConnection;
```

Fig. 9. Declaration of the devices and bus for the AADL model.

Finally, Fig. 10 shows the implementation of the systems and devices, in which all the connections between devices and systems are specified. In this case, the sensors are connected to the serial receiver and the serial receiver to the serial bus that will make the connection to the gateway for the serial data transmission.

```
system implementation top_system.with_devices
    subcomponents
        this_gw:system gw_system;
        this_sensor_1:device Sensor1;
        this_sensor_2:device Sensor2;
        this_sensor_3:device Sensor3;
        this_serial_radio:device SerialRadio;
        this_bus:bus SerialConnection.impl;
    connections
        sensor_1_conn:port this_sensor_1.data_pk_out -> this_serial_radio.data_pk_sensor1;
        sensor_2_conn:port this_sensor_2.data_pk_out -> this_serial_radio.data_pk_sensor2;
        sensor_3_conn:port this_sensor_3.data_pk_out -> this_serial_radio.data_pk_sensor3;
        bus_serial:bus access this_bus -> this_gw.bus_serial;
        bus_radio:bus access this_bus -> this_serial_radio.bus_serial;
end top_system.with_devices;

system implementation gw_system.impl
    subcomponents
            RAM:memory Memories::RAM { Memory_Size => 256 MByte ; };
            CPU:processor Processors::ARM::Generic_ARM;
end gw_system.impl;

device implementation Sensor1.impl
end Sensor1.impl;

device implementation Sensor2.impl
end Sensor2.impl;

device implementation Sensor3.impl
end Sensor3.impl;

device implementation SerialRadio.impl
end SerialRadio.impl;

bus implementation SerialConnection.impl
end SerialConnection.impl;
end example_integration;
```

Fig. 10. Implementation of systems and devices in AADL model.

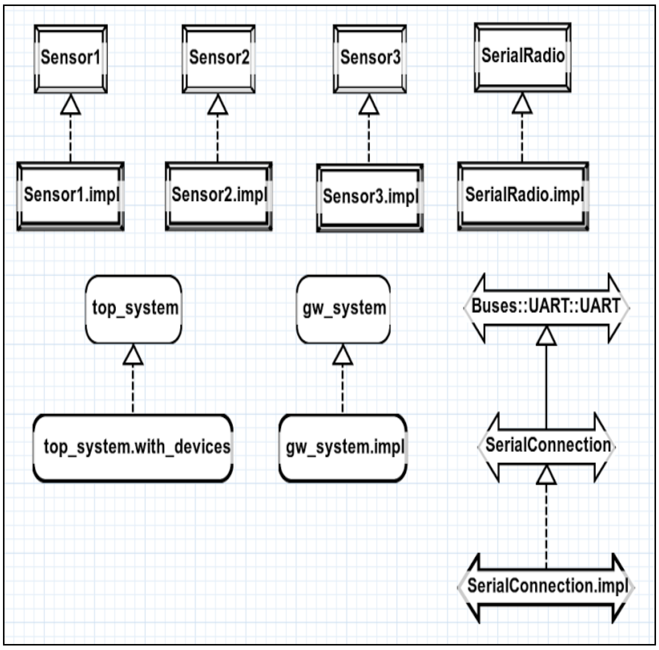

Fig. 11. Diagram of the AADL model.

Besides the *AADL* text files, *AADL* is composed of diagrams and graphical notations. In Fig. 11, is shown the diagram that represents the declarations with the implementations for systems, devices, and buses. This diagram is generated using an *OSATE 2* functionality and allows to observe an abstraction layer of the composition of the *AADL* model. Also, *OSATE 2* allows watching the graphical view of instances and implementations, a view that is corresponding to the *AADL* text model. Fig. 12 and 13 show the graphical view of the *AADL* model described in Fig. 7, 8, 9 and 10, this view allows to have a better view of the system designed and which will be simulated using the *QEMU* emulator, that in this case is represented as the gateway.

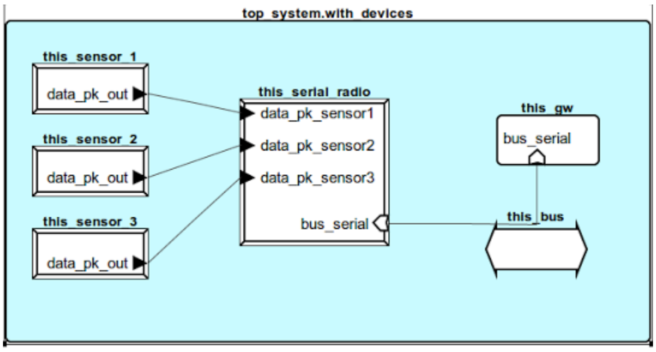

Fig. 12. Graphical view of instances and implementations of the AADL model for the primary system modeled.

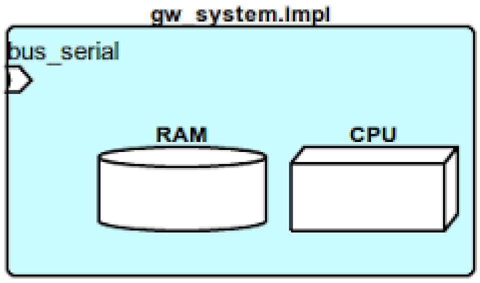

Fig. 13. Graphical view of instances and implementations of the gateway modeled in AADL.

After designing the *AADL* model of the system, the next step is to execute the simulation from the button 'Execute Simulation' at the *CSVD GU*I as shown in Fig. 3. When *QEMU* is executed, it runs the specified *OS* image, that in this case is a Linux-based *OS* image for an '*ARM*' based CPU and with a RAM value of 256 MB as modeled for the system in Fig. 10 and shown in Fig. 13, also, under the *QEMU* emulation will be generated a serial interface to simulate the connection between the serial receiver and the gateway. Fig. 14 shows the serial interface named /dev/ttyV0 and Fig. 15 shows the data being sent to the serial receiver to the gateway through the serial interface.

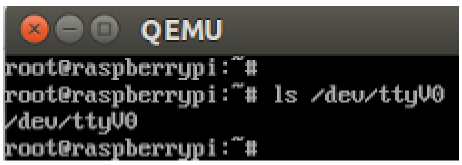

Fig. 14. Serial interface in Linux OS for the connection with the serial receiver simulated device.

As shown in Fig. 15, there are three different types of packages, a data package coming from sensor 1, sensor 2 and sensor 3. As configured for each sensor in the AADL model, there is a different *MAC* address for each sensor as shown in Fig. 9 and Table I, and which can be verified in Fig. 15 by observing the *MAC* value in the data packages received (inside the red squares).

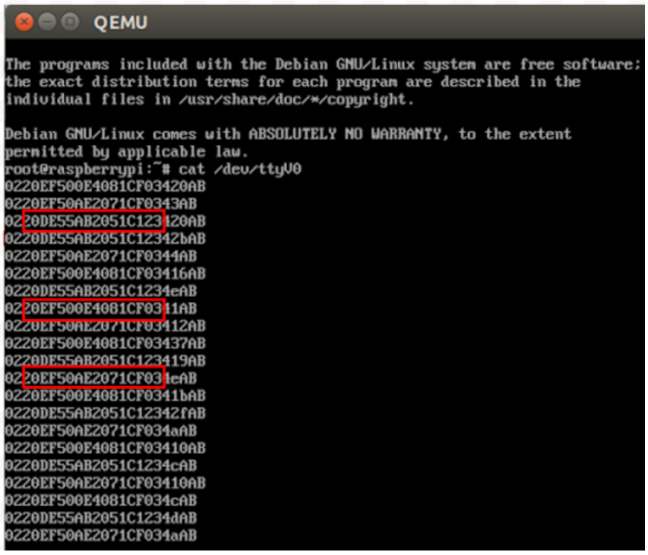

Fig. 15. Serial data being received through the serial interface.

TABLE I: MAC ADDRESS FOR SENSORS IN EXAMPLE

Sensor Number	MAC Address
Sensor 1	20DE55AB2051C123
Sensor 2	20EF50AE2071CF03
Sensor 3	20EF500E4081CF03

By obtaining data in the serial port, simulating a device connected to the emulated gateway, software or service requiring reading data from the specified port can do it for testing or other specific purposes. This simulation can be extended for X number of sensor devices and Y number of serial receivers connected to the emulated gateway (*QEMU* emulator), all by specifying the required characteristics of the system in the *AADL* model.

VI. CONCLUSION

The extensibility of *OSATE 2* allows developers to adapt extensions to it for different kind of purposes including simulation, analysis or other specific features that can be useful during the development of embedded hardware and software process. Also, the fact that the meta-model of *AADL* can be improved, this allows developers to extend the language to support other platforms, protocols, and components with the purpose of modeling another kind of systems and being able to simulate them. The work described in this paper could be modified for future developments, which is helpful for related works needing integration between *OSATE 2* and *QEMU* for simulation of systems or components modeled in *AADL*.

VII. FUTURE WORK

As future work, we propose to add support for the extension of other architectures and platforms that could be emulated. Also, we propose to extend the types of devices and protocols that can be simulated, with the purpose of expanding the usage of the *CSVD* extension developed for *OSATE 2*.

REFERENCES

[1] J. L. Delange, L. Pautet, A. Plantec, M. Kerboeuf, F. Singhoff, and F. Kordon, "Validate, simulate, and implement ARINC653 systems using the AADL," in *Proc. the ACM SIGAda Annual International Conference on Ada and Related Technologies*, New York, NY, USA, 31-44, 2009.

[2] M. Hamdane, A. Chaoui, and M. Strecker, "Toolchain based on MDE for the transformation of AADL models to timed automata models," *Journal of Software Engineering and Applications*, vol. 6, no. 3, pp. 147-155, 2013.

[3] S. Björnander, C. Seceleanu, K. Lundqvist, and P. Pettersson, "ABV - A verifier for the architecture analysis and design language (AADL)," in *Proc. the 2011 16th IEEE International Conference on Engineering of Complex Computer Systems*, Las Vegas, NV, 2011, pp. 355-360.

[4] H. Reza and E. S. Grant, "Toward extending AADL-OSATE toolset with color Petri Nets (CPNs)," in *Proc. the 2009 Sixth International Conference on Information Technology: New Generations*, Las Vegas, NV, 2009, pp. 1085-1088.

[5] F. Kordon, J. Hugues, A. Canals, and A. Dohet, "Embedded systems: Analysis and modeling with SysML, UML and AADL," 2013.

[6] P. H. Feiler and D. P. Gluch, "Model-based engineering with AADL: An introduction to the SAE architecture analysis and design language," Addison-Wesley 2012

[7] SEI. (2017). AADLv2 cheat Sheet: basics. [Online]. Available: https://wiki.sei.cmu.edu/aadl/images/a/ac/Aadlsheet_letter.pdf

[8] Wiki SEI. (2017). Osate 2. [Online]. Available: https://wiki.sei.cmu.edu/aadl/index.php/Osate_2

Fáber D. Giraldo is a system and computer engineer from the University of Quindío, Colombia (with a grant from the Ministry of Education of Colombia).

He has a Ms.Eng. degree with emphasis on informatics from EAFIT University, Colombia (with a grant from EAFIT University). He holds a Ph.D. in informatics (computer science) from the Universitat Politècnica de València, Spain (with a grant from the National Administrative Department of Science, Technology and Innovation of Colombia – COLCIENCIAS).

He is a full assistant professor in the Faculty of Engineering at the University of Quindío, He coordinates the master in engineering with emphasis on software engineering at the University of Quindío, and the center of studies and research of the Faculty of Engineering (CEIFI) of the University of Quindío. He is a researcher of the Information Systems and Industrial Control research group (SINFOCI) at the University of Quindío, and is recognized as associate researcher Type I by Colciencias. His research interests include software engineering, model-driven engineering, software quality, quality in model-driven engineering, software architecture, technical debt, enterprise architecture, enterprise modeling, HCI and innovation. He has supervised more than 100 undergraduate projects, and He has eight master works under his direction to date. He is also invited professor of some Universities of Colombia for master courses in Software Architecture and Software Process Improvement.

Mónica M. Villegas is an electronics engineer from the University of Quindío, Colombia. She is a candidate for the master engineering degree with emphasis on software engineering from the University of Quindío. In 2017 she began doctoral studies in informatics engineering at the Federico Santa Maria Technical University - UTFSM (with a grant from CONICYT Chile). Her research interests include software engineering, embedded systems, model-driven engineering, software architecture, and robotics.