Platform-specific Modeling for RIOT based IoT Systems

Burak Karaduman
International Computer Institute
Izmir, Turkey
bburakkaraduman@gmail.com

Moharram Challenger
University of Antwerp
Antwerp, Belgium
moharram.challenger@uantwerp.be

Raheleh Eslampanah
University of Antwerp
Antwerp, Belgium
raheleh.eslampanah@uantwerpen.be

Joachim Denil
University of Antwerp
Antwerp, Belgium
joachim.denil@uantwerpen.be

Hans Vangheluwe
University of Antwerp
Antwerp, Belgium
hans.vangheluwe@uantwerpen.be

ABSTRACT

The variety of smart devices and their communication models increase the development complexity of embedded software for IoT. Thus, the development of these systems becomes more error-prone, complex, and costly. To tackle this problem, in this study, a model-driven approach is proposed for the development of RIOT-OS based IoT systems. To this end, a meta-model is designed for RIOT-OS. Based on this meta-model, a Domain-specific Modeling Language (DSML) is developed to graphically represent the domain models. To gain more functionality for the language, domain rules are defined as constraints. Also, system codes are generated partially from the instance models. In this way, the development is supported by code synthesis and the number of bugs is reduced. Finally, a smart irrigation system and a smart lighting system are implemented to evaluate the proposed DSML. The results show that about 83.5% of the final code is generated automatically on average.

KEYWORDS

Domain-specific Modeling, Internet of Things, Embedded Software, Wireless Sensor Network, RIOT, Smart Irrigation System

ACM Reference Format:
Burak Karaduman, Moharram Challenger, Raheleh Eslampanah, Joachim Denil, and Hans Vangheluwe. 2020. Platform-specific Modeling for RIOT based IoT Systems. In *IEEE/ACM 42nd International Conference on Software Engineering Workshops (ICSEW'20), May 23–29, 2020, Seoul, Republic of Korea.* ACM, New York, NY, USA, 8 pages. https://doi.org/10.1145/3387940.3392194

1 INTRODUCTION

Internet of Things (IoT) is rapidly taking its place in different technologies and markets, such as home appliances, smart buildings, and Industry 4.0 applications. Generally, in these systems, devices communicate with each other and work in coordination to create a smart environment/space. The IoT systems consist of different components such as sensors, actuators, computation elements, and data (or log) managers. Creating IoT systems require hardware components, an operating system (or a firmware) to manage hardware resources, a communication protocol to establish a network. The resulting system is complex with different components requiring to be programmed to work collaboratively. This complexity makes the design and analysis of these systems time-consuming, costly, and cumbersome. This can be addressed with Model-Driven Engineering (MDE) techniques [15] which focus on models during the development to increase the level of abstraction and automatically synthesize the system artifacts. Using a model-centric development methodology, design models can also be used for the early analysis and validation of the system.

To this end, we introduce a Domain-specific Modeling Language (DSML) [18], called DSML4RIOT, for the design and development of RIOT based IoT systems. DSML4RIOT provides an abstraction for RIOT-OS [3] which focuses on programming low-power, memory-constrained, and wireless IoT devices. In this study, model-driven development is applied on RIOT's TCP, UDP, and CoAP protocols (backbone features to create a WSN or Wi-Fi based network) as well as multi-threading capability. Moreover, the GPIO feature of the RIOT operating system is modeled and code generation rules are written to achieve digital I/O operations as well as to implement IoT systems.

The RIOT operating system is selected as it has support for real-time tasks, multi-thread applications, small memory foot-print devices, vast hardware types, Wi-Fi protocol (IEEE 802.11) based IoT development boards (e.g. ESP32[1]), as well as Wireless Sensor Network (WSN) protocol (IEEE 802.15.4) based devices. This makes it a promising operating system for real-time IoT systems. It is a C-language based OS and considers modularity principles. RIOT has a full IPv6 network protocol stack as well as standards protocols such as 6LoWPAN, RPL, TCP, and UDP.

To evaluate the generation capability of the proposed DSML, two case studies are designed and implemented using DSML4RIOT. To this end, ESP32-WROOM (a low-cost 32-bit dual-core microcontroller) is used which has an integrated Wi-Fi and Bluetooth modules and targets a wide variety of applications ranging from low-power sensor networks to the demanding tasks.

This paper is organized as follows: Section 2 discusses the related work. The abstract syntax of the proposed DSML is presented in Section 3. In Section 4, the code generation rules are discussed.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
ICSEW'20, May 23–29,20, Seoul,Republic of Korea
© 2020 Association for Computing Machinery.
ACM ISBN 978-1-4503-7963-2/20/05…$15.00
https://doi.org/10.1145/3387940.3392194

[1]https://www.espressif.com/en/products/hardware/esp-wroom-32/overview

Section 5 elaborates the two case studies to evaluate the performance of the proposed modeling language. The paper is evaluated in Section 6 and concluded in Section 7.

2 RELATED WORK

This study contributes to the literature by providing a model-driven engineering method for developing RIOT-based IoT systems. To the best of our knowledge, currently, there is no study to address the modeling and development of WSNs based IoT systems using RIOT-OS. Providing a generative modeling language, as proposed in this study, can facilitate the efficient development of IoT systems based on WSN.

In the literature, there are some studies related to modeling and meta-modeling of IoT systems. Some of these studies address meta-models [20] and issues such as node connectivity, configuration [11, 19], security [4], service discovery [1], and runtime adaptability [12]. In addition, there are recent modeling approaches such as facet-oriented modeling [8], model federation [10] and template-based meta-modeling [7]. In the study of [5], a comprehensive survey of model view approaches is presented. In [16], a DSML is presented specific for ultra-thin IoT devices. The authors provide a DSML named IoT-PML with code generation support for sensor device drivers. They use RIOT operating system to implement a case study.

We had a series of studies with the ultimate goal of providing a platform-independent generative modeling language supporting both WSN and Wi-Fi based IoT systems independent of target tools. In this regard, we have developed a meta-model for Contiki-OS [9]. Later, domain-specific languages have been developed for TinyOS [17] and ContikiOS [21]. Moreover, to evaluate these DSMLs, case studies [14], [13], and [2] are implemented. The current study addresses domain-specific modeling for RIOT and the next step would be to level up the abstraction according to these DSMLs and develop a Platform Independent Modelling (PIM) framework to support generative development of IoT systems based on WSN and Wi-Fi systems.

3 THE SYNTAX OF THE LANGUAGE AND STATIC SEMANTICS

This section presents the main elements of the proposed DSML including the abstract syntax as a meta-model, static semantics as constraint checking rules, and code generation for the translational semantics of the language.

3.1 Abstraction syntax

The meta-model of RIOT operating system is introduced in this section. It represents the abstract syntax of the domain-specific language. Generally, a meta-model represents system elements, relations, and cardinally constraints of the elements and their relations. This meta-model is integrated with our previous study [21] which supports modeling ContikiOS. Considering the space limitations, the whole meta-model is not shown in this paper. However, the full version along with other components are available in an online bundle[2].

[2]http://bit.ly/2RHSf2t

Table 1: Number of elements in sub-parts of the meta-model

	EClass	EAttribute	Composition	EReferance
System View	4	3	0	4
IoT Log Manager	4	6	3	7
Gateway(Rpi 3)[3]	6	9	5	8
ESP8266	9	21	8	5
ContikiOS	24	60	20	26
RIOT-OS	8	24	7	6
Total	51	120	43	52

To give an idea of the complexity of the full meta-model Table 1 gives the number of meta-classes, attributes, compositions, and references for the meta-model (including RIOT-OS, ContikiOS, ESP, Gateway, Log Manager related elements) as well as for the whole meta-model. Elements for the System view are not considered in the total numbers, as it borrows its elements from the root elements of the other viewpoints (other Classes of the meta-model).

According to Table 1, the meta-model elements and relations of ContikiOS are much more than RIOT-OS. This is because Contiki has a lot of event-based features and specialized timers. Moreover, in the design of the ContikiOS' meta-model, some features are modeled as EClass to give diversity to the user. In RIOT-OS' meta-model, RIOT's features are modeled as attributes to provide more abstraction to the user. The same approach cannot be used in ContikiOS, because it is not possible to decide which and how many timers (or events) are going to be used by the user. In RIOT-OS, it is possible to decide where a timer should be. Since it is a real-time operating system, it does not have additional features for event handling. Accordingly, most of the features could be modeled as an attribute in one EClass. In Figure 1 part of the meta-model related to RIOT's is demonstrated.

The important elements of the meta-model are described below:

- Thread: It is used to define additional threads to the main thread. *Name* attribute is used for the function name of that thread. *Priority* value can be set to define the priority of the thread.
- Xtimer: It is a delay timer that can be used in threads. This feature is useful if a delay has to be used before the initialization of any task or to give running time for the other threads.
- PlatformR: This is the root of the elements of the RIOT viewpoint.
- NodeR: It represents any node (e.g ESP32) and it can be either server or client. If it is a server then, it can also run a *CoAP* server.
- CoapParams: Coap params is used for CoAP server. It defines a request as POST or GET. *FunctionName* parameter defines the URI of the request address.
- Socket: *Socket* is used for UDP and TCP protocols. With its attributes, it defines the usage of IPv4 or IPv6 as well as *Local* and *Remote* ports to establish a connection. Also, *IPAddress* parameter is the address to which it is connected and *Message* attribute represents the message to be sent.
- GPIO: It used to set a *GPIO* of the board (e.g ESP32) as active. *Number* attribute represents the GPIO number to be activated. Once it is activated, then input or output role can

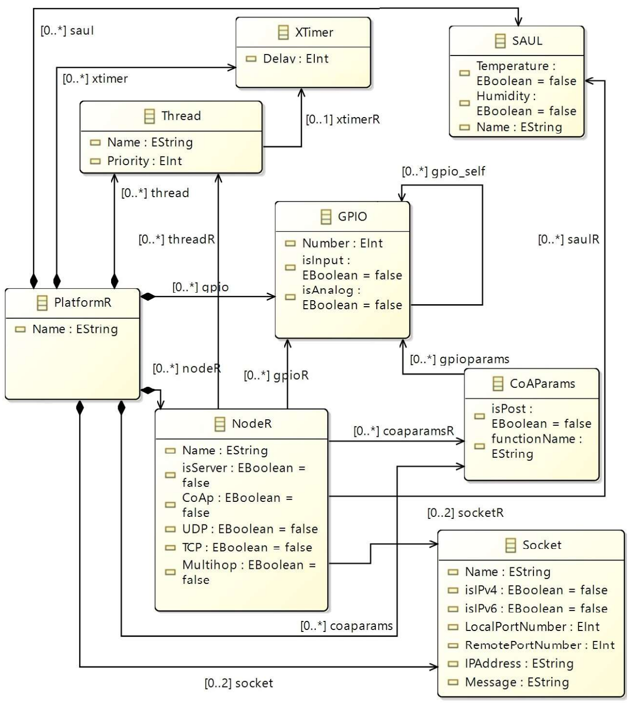


Figure 1: RIOT-OS Meta-model

be defined and also it can be set as analog. It is configured as digital by default.

- SAUL: Saul is a sensor/actuator abstraction layer for RIOT. In this way, any sensor can be activated and any actuator can be interacted without adding device parameters or extra configuration codes in the application.

3.2 Static Semantics

To apply the domain rules in the modeling environment and help the user to design more accurate models, semantic constraints are implemented in the framework using the Acceleo Query Language (AQL) [4] which is built-in in the Eclipse Sirius Framework. In fact, these rules apply the domain rules in the RIOT modeling environment.

The constraint checking languages are more expressive than the simple cardinality property checking rules implemented in the meta-model and they can check those properties as well. However, the constraint checking rules are generally used to implement the rules that are not possible to implement in the meta-model, such as the rules checking a property between the meta-elements that are not directly linked together. In other words, these rules are complementary to the meta-model cardinality rules. Some of the rules implemented in DSML4IOT are listed in the following list.

- Constraint rule 1: aql: (self.Number > 0)
- Constraint rule 2: aql: (self.UDP and self.TCP and self.socketR->size()>0)
- Constraint rule 3: aql:(self.LocalPortNumber != self. remotePortNumber)
- Constraint Rule 4: aql: (self.threadR->size() < 5)
- Constraint rule 5: aql: not (self.isIPv4 and self.isIPv6)

Constraint rule 1 checks whether the *Number* attribute of *GPIO* element is a negative value or not, and constraint rule 2 checks whether *TCP*, *UDP* or both of them are used in the model and whether *Socket* element is not used then it warns the user to include one. In constraint rule 3, local and remote port numbers are compared, to control not to have the same number, because, if these two ports have the same value, then messages cannot be sent. Constraint rule 4 is written to warn the user for using too much *threads* for these recourse constraint devices. If the user adds more than 5 threads, the user is warned. Because using a lot of threads may lead to poor performance and memory overflow problems. Constraint rule 5 is implemented to direct the user to use either IPv6 or IPv4, as enabling these two IP protocols at the same time is not considered a wise choice.

3.3 Graphical Editor

In this section, the graphical concrete syntax used in the editor for modeling target systems using DSML4RIOT is introduced. The graphical editor is developed using the Eclipse Sirius Framework[5].

The concrete syntax provides a mapping between graphical/textual notations and abstract syntax elements. In this study, some graphical notations are used to represent the elements in the editor. The modeling environment for RIOT operating system is added as a new viewpoint as well as an extension to the DSML introduced in [21]. In Figure 2, RIOT operating system is represented by a symbol *R*. In this way, the Platform-specific Modeling environment can be created using this element, represented by *R* symbol.

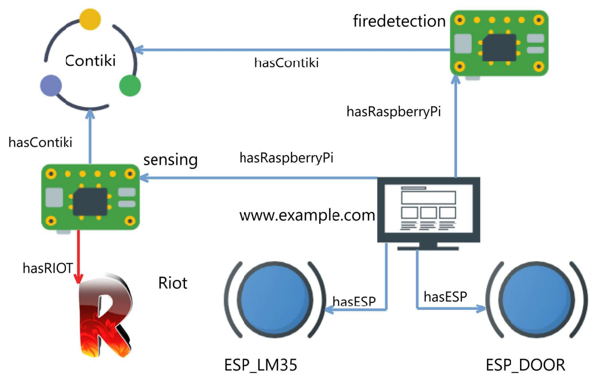


Figure 2: RIOT-OS as a new viewpoint in system viewpoint

In Figure 3, elements and relations of RIOT operating system in DSML4RIOT are shown. They are used to model a system and the designed model can be used to generate architectural codes for the target system. In Figure 5a and 5b, a model of smart irrigation case study is given. The models of the case studies are designed using

[4]https://www.eclipse.org/sirius/doc/specifier/general/Writing_Queries.html

[5]https://www.eclipse.org/sirius/

elements and relations contained by palette that is represented by Figure 3. The elements are represented by star sign and relations that connect these elements are represented by blue solid line.

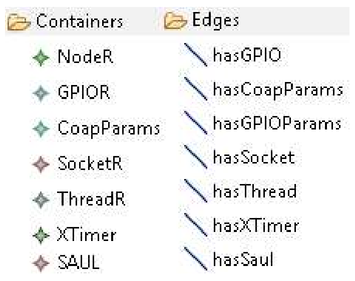


Figure 3: Elements and relations of RIOT-OS

4 CODE GENERATION: TRANSNATIONAL SEMANTICS OF DSML4RIOT

In this section, the code generation mechanism of DSML4RIOT is elaborated. Also, some excerpts are given and discussed from generation rules.

RIOT-OS' threads are defined using *Thread* statement and all the threads are run in parallel. During the development of DSML4RIOT models, the same concept is used. The code generation rules are written to create each defined thread with the same pattern. In RIOT-OS, a priority-based scheduling algorithm manages to schedule threads according to their priority level. In DSML4RIOT, thread priorities can be also set by the user. Each thread, including the main thread, is generated to run with the default priority. However, if the user desires to set a different priority level for a thread, then it must be done by adding thread call/yield functions (as delta codes) to handle the desired thread sequence.

Listing 1: Code excerpt for TCP, UDP, and IP rules

```
1   [ if (n.socketR −> at(1) . isIPv4 ) ] #include" net/ipv4/addr.h"
        // for udp\&tcp
2   [/ if ]
3   [ if (n.socketR −> at(1) . isIPv6 ) ] #include" net/ipv6/addr.h"
        // for udp\&tcp
4   [/ if ]
5   [ if(n.TCP) or (n.UDP)]#include"net/af.h"[/ if ] //udp\& tcp
6   [ if(n.UDP)]
7       #include "net/protnum.h" // for udp
8       #include "net/sock/udp.h" // for udp
9   [/ if ]
10  [ if (n.TCP)] #include "net/sock/tcp.h" [/ if ] // for tcp
```

In Listing 1, a code excerpt of the code generation rules for UDP and TCP libraries is shown. According to the user's model, necessary libraries are included in the generated code. For example, if the user designs a TCP based application, TCP library, and other related libraries that are shown in line 5 and line 10 are instructed. If the UDP is decided to be used, then, lines between 6 and 9 are instructed. As mentioned earlier, the user can use either IPv6 or IPv4. If the user selects both IPv6 and IPv4, a constraint rule, which is written to warn the user, pops-up to direct the user to select only one of them. If the user selects IPv4, then, lines 1 and 2 are instructed. Lines 3 and 4 are instructed when the user decides to use IPv6.

In Listing 2, an excerpt from code generation rules is given from *Socket* element. If the model has a socket element, the related rules are instructed to generate codes. According to the user selection, *sock_udp_ep_t local* variable is configured as either IPv4 or IPv6. Lines 1 and 3 are instructed to configure a socket variable for IPv6. The same configuration is made for IPv4 by instructing lines 2 and 4.

Listing 2: IPv4 and IPv6 based network configuration rules

```
1   [ if(socket. isIPv6 )]sock_udp_ep_t local  = SOCK_IPV6_EP_ANY;
        [/if]
2   [ if(socket. isIPv4 )]sock_udp_ep_t local  = SOCK_IPV4_EP_ANY;
        [/if]
3   [ if(socket. isIPv6 )]sock_udp_ep_t remote = {. family=AF_INET6};
        [/ if ]
4   [ if(socket. isIPv4 )]sock_udp_ep_t remote = {. family=AF_INET};
        [/ if ]
```

When the user selects TCP and/or UDP communication protocols, specific threads and thread functions are generated automatically with the same priority as the main thread (so each thread guarantees to be run). Therefore, additional threads are not required to be defined for UDP and TCP by the user. In this way, these automatically-generated threads ease the user's development work. Threads for UDP and TCP protocols are defined to be started when the device is powered on. The user-defined threads in RIOT operating system must be defined in the main thread. To run all threads equally, when the main thread is initialized *THREAD CREATE WOUT YIELD* parameter must be given as a parameter to each thread. Line 1 in Listing 3 shows the code generation rule for this special thread. In other words, If the user activates TCP protocol, then a special thread, a stack, and a thread function are generated. Also, the user can create a thread for any specific task. Lines 3-5 in Listing 3 show the code generation rules for a thread creation according to the user model.

Listing 3: User-defined thread generation rule

```
1   [ if (n.TCP)]char tcp_stack[ ' [THREAD_STACKSIZE_MAIN]'/];[/
        if]
2    ...
3   [for (thread:Thread|n.threadR)]
4   thread_create ([ thread .Name/]_stack, sizeof ([ thread .Name/]
        _stack), THREAD_PRIORITY_MAIN − [thread.Priority/],
        THREAD_CREATE_WOUT_YIELD, [thread.Name/], NULL, "[
        thread.Name/]");
5   [/for]
```

To create a thread, a stack, a thread function and a definition in the main function of this thread must be provided. In Listing 3, at line 4 *thread_create* function is used to create a thread with given parameters. If priority is not set, then the priority level is equal to the main thread. Moreover, *THREAD_CREATE_WOUT_YIELD* is given as a parameter to start a thread when the main function is initialized. *thread_name* represents the name of the function and also it is a pointer to thread's function. The for loop at line 3 and 5 generates the threads that are defined by the user in the model.

In Listing 4, lines 1-8 generate handler functions that are defined using *coapparams* element in the model. These functions are called when any request is made to that function with a URI. If any *coapparams* element has to use a *GPIO*, then, also *GPIO* configurations are generated in this rule. Moreover, lines 5-7 are instructed to toggle any *GPIO*. In this way, when any request is made, actuation can be made such as turn on/off a light, control a relay, or control a DC motor. Lines 10-14, define a CoAP function to be generated and accessed as either a POST or a GET request.

Listing 4: CoAP Code generation rule

```
1  [if (n.CoAp)]
2  [for(coap:CoAParams|n.coaparamsR)]
3      static ssize_t _[coap.functionName/]_handler(coap_pkt_t *
           pkt, uint8_t *buf, size_t len, void *context) {(void)
           context;
4      [for (gpio:GPIO| n.coaparamsR.gpioparams)]
5          extern gpio_t g[gpio.Number/];
6          gpio_toggle(g[gpio.Number/]);
7      [/for]}
8  [/for]
9  const coap_resource_t coap_resources['[]'/] = {
10     [for(coap:CoAParams|n.coaparamsR)]
11         {"/[coap.functionName/]",[if (coap.isPost)]COAP_GET
               [else]COAP_POST[/if],_[coap.functionName/]
               _handler,NULL},
12         {"/[coap.functionName/]/",[if (coap.isPost)]
               COAP_GET[else]COAP_POST[/if],_[coap.
               functionName/]_handler,NULL}[if(n.coaparamsR->
               size()<>i)],[/if]
13     [/for]
14 };
```

5 CASE STUDIES

To validate DSML4RIOT and evaluate its code generation capability, two case studies are designed and implemented using the proposed DSML. The architectural codes are generated using the DSML based on the design models and then the delta codes are added to have final code. UDP, IPv6, and GPIO features are used in these two case studies.

In the following sub-sections, these case studies are introduced and their development processes are elaborated. In general, most of the final code is generated using DSML4RIOT, which will be discussed in detail in the Evaluation Section.

5.1 Smart Irrigation System

This section includes the design and development of the smart irrigation system.

5.1.1 Case study design. The first case study is a smart irrigation system. It works using UDP protocol with two nodes including a server node and a client node. The client node uses a digital GPIO pin which has a moisture sensor connected. Moisture sensor's digital output pin is used and connected to the client node's GPIO pin. When moisture level drops to a certain level, it generates a digital HIGH voltage level, then, the client node sends a message to the server. If the received message is *Dry* then the server activates the GPIO pin to turn on the water motor/pump for irrigation of the plant. The water motor runs for 3 seconds then the ESP32's GPIO pin is set as low to stop the motor.

The moisture level of the soil is sampled by GPIO pin 15 of the client-side. Moreover, in the client-side GPIO pin 2 toggles the built-in LED of the ESP32 to inform the person in charge about the system running. If the moisture level is below a set point (this point is set by a variable resistor on the moisture sensor), in other words when the soil gets dry, moisture sensor sends a digital HIGH value to pin 15. Then, a string message, *Dry*, is sent to the server node. In parallel, the server always listens to the necessary port. If the message named *Dry* is received, then it activates the GPIO 5 to run the water motor for 3 seconds to irrigate the soil.

Figure 4 represents the schematic for the case studies (the setup of the case studies are shown in Appendix Section, Figure 7 for lighting system and Figure 8 for irrigation system). In Figure 4a, the irrigation system is depicted and in Figure 4b, the lighting system is depicted. Also, on the right side of irrigation system, the client node with a moisture sensor is shown. On the left side, the server node with a water motor is represented.

The design of this case study in DSML4RIOT is shown in Figure 5a for server-side and Figure 5b for client-side.

5.1.2 Generation and implementation using DSML4RIOT. Most of the code to create the smart irrigation system is generated by DSML4RIOT. The code generation is done using the models demonstrated in Figure 5.

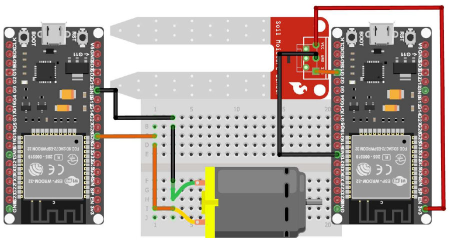
According to the Figures 5a and 5b, using three elements in the server-side and four elements in the client-side, majority of the code is generated automatically. As Figure 5a shows, *Server_Node* element has a *Socket* and a *GPIO* element. In this way, it establishes an UDP communication as a server and controls *GPIO* number 5. As Figure 5b shows, *Client_Node* element has a Socket and two *GPIO* elements. In this way, it establishes an UDP communication as a client and controls *GPIO* number 15 and number 2.

Listing 5 shows the delta codes that are added to the server-side. In lines 1 and 2 *puts* function is called to print the incoming messages to the screen. This also helps for testing purposes. In lines 3-7, the generated GPIO functions are shown to activate and deactivate GPIO 5. The xtimer_sleep function is called for three seconds delay. If the client sends *Dry* keyword to the server, then, the server sets *HIGH* GPIO number 5, waits for 3 seconds then sets the GPIO number 5 to *LOW*.

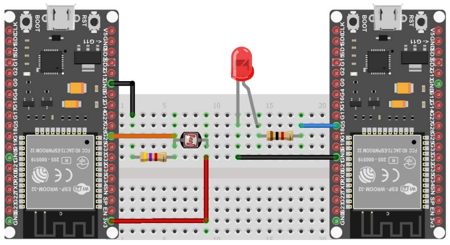
In Listing 5, the client-side's delta codes also are shown. In the client-side, line 9, *GPIO* number 15 is checked, whether the moisture sensor sends digital *LOW*. When *GPIO* number 15 equals to zero (i.e. digital *LOW*) then the client sends *Dry* keyword to the server. The *xtimer_sleep* function in line 10 is generated by DSML4RIOT, but its place is moved to another line in the generated code. In line 11, *GPIO* number 2 is toggled to inform about system activity. In line 12, *printf* function is called to print the state of the *GPIO 15*. In line 13 *xtimer_sleep* function is added to create a delay.

Listing 5: The delta codes for the server and client sides of the smart irrigation system

```
1  puts("A message is received ");
2  puts(message);
3  if(strcmp(message,"Dry")==0){
4      gpio_set(g5);
```

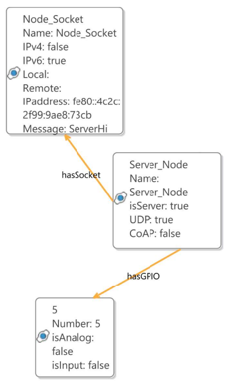


(a) Smart irrigation case study

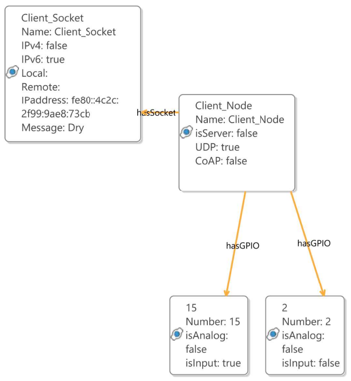


(b) Smart lighting case study

Figure 4: Schematics for the case studies



(a) Server-side model



(b) Client-side model

Figure 5: Models for the Irrigation case study

```
    xtimer_sleep (3) ;
    gpio_clear (g5);
}
 ...
if(gpio_read(g15)==0){}
xtimer_sleep (1) ;
gpio_toggle (g2);
printf ("%d \n",gpio_read(g15)) ;
xtimer_sleep (1) ;
```

5.2 Smart Lighting System

This section includes the design and development of smart lighting system.

5.2.1 Case study design. As the second case study, a light control system is designed and implemented. In the hardware part, an ESP32 development board is programmed to measure light intensity using a light-dependent resistor. A comparator circuit is built using a 4.7k resistor and a 5k light-dependent resistor to sample light intensity in a room. When light intensity drops below a certain level, the client sends a message to the server node to open the room light. To sample the environment light intensity in the client node, the server node must close the room light not to belie the sampling results in the client-side. Therefore, before sampling environmental light intensity, the client node must send a message to the server node to close the light. Next, the client node samples the environment light intensity and decides to open either the room light or to close it. In Figure 4b, the schematic of this case study is shown. On the left side of this figure, the client node with a comparator circuit is shown. On the right side, the server node with a light-emitting diode is represented.

Due to the similarity of the design models between two case studies, the model of the smart light system in DSML4RIOT is not included in the paper. However, the generated and delta code excerpts are discussed in the next section.

5.2.2 Generation and implementation using DSML4RIOT. Listing 6 represents the delta codes that are added to server-side of the smart lighting case study in lines 1-4. In lines 1-2, the incoming message and notification sentence are displayed for debugging purposes. Lines 3-4 show checks the server input. If it receives *o* character then it sets the *GPIO* number to *HIGH* to open the light. It closes the light when it receives *f* character. Listing 6 shows the delta codes on the client-side in lines 6-9. In line 6, *msg* variable is initialized. In line 7, one hour period is set for the sampling time. In lines 8-9, the client decides which message is sent to the server. In line 8, it samples the environment light intensity and checks the result of the light-dependent resistor from *GPIO* number 15.

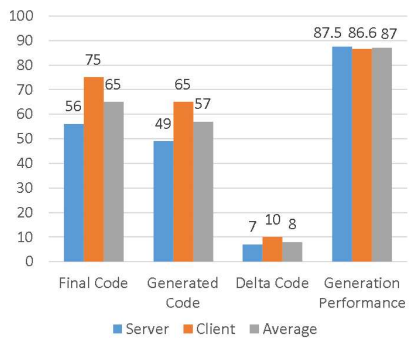
Listing 6: The delta code for the server and client sides of smart lighting system

```
1  puts("A message is  received  ");
2  puts(message);
3  if(strcmp(message,"o")==0){gpio_set(g2); //open  the  lights }
4  else  if(strcmp(message,"f")==0){gpio_clear(g2); // close  the
          lights }
5   ...
6  char msg=0;
7  xtimer_sleep(3600) ;  //  sample  period  is  1  hour.
8  if(gpio_read(g15)==0){msg="o";}
9  else{msg="f";}
```

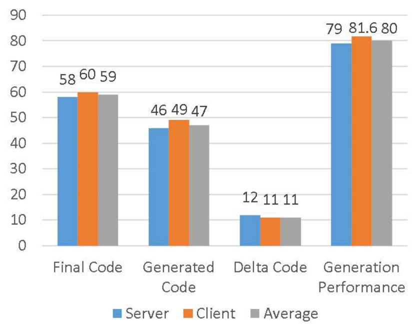
6 EVALUATION

Evaluation of our modeling platform is done by assessing its code generation capabilities [6]. This was done by comparing the lines of generated code and final code (the code after adding the user's delta code).

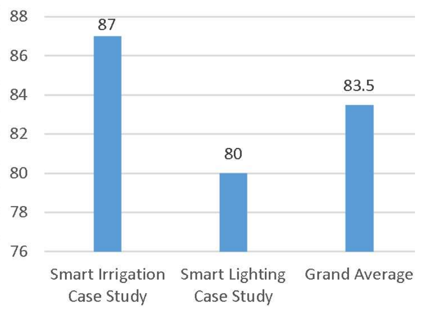
We are aware that the evaluation with a single case study introduces an external threat to the validity of the results, an inappropriate generalization of the results. We reduce this risk by evaluating



(a) Smart irrigation case study



(b) Smart lighting case study



(c) Averages

Figure 6: Generation performance of the DSML4RIOT

DSML4RIOT with multiple case studies. To this end, we focus on generation performance of two case studies discussed earlier.

Considering the evaluation of the code generation performance of the smart irrigation case study, moving a line of code is treated as a delta line of code. For this case study, 65 lines of client-side code are generated in total. Also, 10 lines of code are either added manually or replaced in the generated code. Additionally, the *strcmp* function also counts as extra delta code on the server-side. As a result, 65 lines of code are generated successfully over total of 75 lines of the final code. So, the code generation's performance is %86.6 for the client-side of this case study. At the server-side, 49 lines of code are generated automatically and 7 lines of code are either added manually or replaced. As a result, 49 lines of code are generated over 56 lines of the final code. So, the code generation performance is %87.5 for the server-side of the irrigation case study.

To ease the understating of these figures, they are depicted in a diagram. Figure 6a shows a comparison of number of the final code, generated lines of code, and delta lines of code for the smart irrigation system. It also shows the generation performance. For each of these items, the Blue bars (the left bars) represent the server-side, the orange bars (the middle bars) represent the client-side, and grey bars (the right bars) represent the average of the two sides.

More code is generated for the client-side compared to the server-side, so, the client-side requires more delta codes. However, the performance rate of the client-side and the server-side are close together. As a result, a high percentage of generation rate was achieved with the number of codes generated by DSML4RIOT, even if the client-side required more delta codes. In other words, the number of the generated codes overcame the number of the required delta codes.

Similar to the evaluation of the irrigation case study, most of the code for the smart lighting case study is generated by DSML4RIOT. However, compared to the smart irrigation case study, the smart lighting system has more delta code.

Considering the evaluation of code generation performance of DSML4RIOT in the development of the smart lighting system, moving a code line is treated as a delta line of code. Additionally, the *gpio_read* function also counts as extra delta code. In the client-side 49 lines of code are generated and 11 lines of code are either added manually or moved to another line. On the server-side, 46 lines of code are generated and 12 lines of code are added manually. Additionally, the *strcmp* function also counts as extra delta code. The generation performance of the DSML4RIOT for this case study is 79% on the server-side and 81.6% on the client-side.

These figures are demonstrated in a diagram to ease their understanding. Figure 6b shows a comparison of the number of generated lines of code with the number of delta lines of code for the smart lighting system. It also shows the generation performance in the diagram.

The final code bars show the lines of code required to build the smart lighting system. They are close together at the server-side and the client-side. Since the server-side has less generated lines of code and required more delta codes compared to the client-side the success rate of the server-side is low comparing to the client-side. The generated lines of code of the client-side are more than the server-side and it required fewer delta codes in this case study. The same result could not be observed in the smart irrigation system. This can be concluded as the requirements of a case study changes according to its target domain and given functionalities.

To compare the averages, Figure 6c shows the average code generation performance for the smart irrigation system (87%) and the smart lighting system (80%). The grand average of generation performance for these two case studies using DSML4RIOT is 83.5% of total artifacts.

7 CONCLUSION

In this study, a domain-specific modeling language is presented, called DSML4RIOT, to support the design and implementation of IoT systems based on RIOT operating system. DSML4RIOT addresses the increasing complexity and difficulty of developing these systems. Our study provides an abstraction to reduce this complexity and to ease this difficulty. A set of graphical notations, as well as some domain constraints, are subsequently used to develop a graphical editor for the DSML. The generation capability of the proposed DSML is evaluated by two case studies.

Our evaluation results show that the proposed approach for the development of IoT systems, using the provided DSML, can generate a significant part, about 83.5%, of the final code automatically. This can reduce the development complexity, time and cost as well as increasing the quality of the resulting system, as the generated code is architectural and with best practice (and without errors).

As future work, we plan to raise the level of abstraction for the modeling environment to support Platform-Independent Modelling by integrating the results of our previous studies, [21] and [17], and using the model to model transformations.

ACKNOWLEDGMENT

This study is partially funded by the Scientific Research Project No 17-UBE-002 at EGE University, Izmir, Turkey. Some of the ideas in this work were developed during Short Term Scientific Missions (STSM Nos. 41940 and 41967) by two of the authors, within the IC1404 Multi-Paradigm Modelling for Cyber-Physical Systems (MPM4CPS) COST Action.

REFERENCES

[1] Mehdi Adda and Rabeb Saad. 2014. A data sharing strategy and a DSL for service discovery, selection and consumption for the IoT. *Procedia Computer Science* 37 (2014), 92–100.

[2] Sadık Arslan, Moharram Challenger, and Orhan Dagdeviren. 2017. Wireless sensor network based fire detection system for libraries. In *2017 International Conference on Computer Science and Engineering (UBMK)*. IEEE, 271–276.

[3] Emmanuel Baccelli, Oliver Hahm, Mesut Gunes, Matthias Wahlisch, and Thomas C Schmidt. 2013. RIOT OS: Towards an OS for the Internet of Things. In *2013 IEEE conference on computer communications workshops (INFOCOM WKSHPS)*. IEEE, 79–80.

[4] Delphine Beaulaton, Najah Ben Said, Ioana Cristescu, Régis Fleurquin, Axel Legay, Jean Quilbeuf, and Salah Sadou. 2018. A language for analyzing security of IoT systems. In *2018 13th Annual Conference on System of Systems Engineering (SoSE)*. IEEE, 37–44.

[5] Hugo Bruneliere, Erik Burger, Jordi Cabot, and Manuel Wimmer. 2019. A feature-based survey of model view approaches. *Software & Systems Modeling* 18, 3 (2019), 1931–1952.

[6] Moharram Challenger, Geylani Kardas, and Bedir Tekinerdogan. 2016. A systematic approach to evaluating domain-specific modeling language environments for multi-agent systems. *Software Quality Journal* 24, 3 (2016), 755–795.

[7] Juan De Lara and Esther Guerra. 2010. Generic meta-modelling with concepts, templates and mixin layers. In *International Conference on Model Driven Engineering Languages and Systems*. 16–30.

[8] Juan de Lara, Esther Guerra, Jörg Kienzle, and Yanis Hattab. 2018. Facet-oriented modelling: open objects for model-driven engineering. In *Proceedings of the 11th ACM SIGPLAN International Conference on Software Language Engineering*. ACM, 147–159.

[9] Caglar Durmaz, Moharram Challenger, Orhan Dagdeviren, and Geylani Kardas. 2017. Modelling Contiki-Based IoT Systems. In *6th Symposium on Languages, Applications and Technologies (SLATE 2017) (OpenAccess Series in Informatics (OASIcs), Vol. 56)*. Dagstuhl, Germany, 5:1–5:13. https://doi.org/10.4230/OASIcs.SLATE.2017.5

[10] Fahad R Golra, Antoine Beugnard, Fabien Dagnat, Sylvain Guerin, and Christophe Guychard. 2016. Addressing modularity for heterogeneous multi-model systems using model federation. In *Companion Proceedings of the 15th International Conference on Modularity*. ACM, 206–211.

[11] Nicolas Harrand, Franck Fleurey, Brice Morin, and Knut Eilif Husa. 2016. Thingml: a language and code generation framework for heterogeneous targets. In *Proceedings of the ACM/IEEE 19th International Conference on Model Driven Engineering Languages and Systems*. ACM, 125–135.

[12] Mahmoud Hussein, Shuai Li, and Ansgar Radermacher. 2017. Model-driven Development of Adaptive IoT Systems.. In *MODELS (Satellite Events)*. 17–23.

[13] Burak Karaduman, Tansu Aşıcı, Moharram Challenger, and Raheleh Eslampanah. 2018. A Cloud and Contiki based Fire Detection System using Multi-Hop Wireless Sensor Networks. In *Proceedings of the Fourth Intl. Conf. on Engineering & MIS 2018*. ACM, 66.

[14] Burak Karaduman, Moharram Challenger, and Raheleh Eslampanah. 2018. ContikiOS based library fire detection system. In *2018 5th Intl. Conf. on Electrical and Electronic Engineering (ICEEE)*. IEEE, 247–251.

[15] Geylani Kardas, Zekai Demirezen, and Moharram Challenger. 2010. Towards a DSML for semantic web enabled multi-agent systems. In *Proceedings of the International Workshop on Formalization of Modeling Languages*. 1–5.

[16] Arthur Kühlwein, Anton Paule, Leon Hielscher, Wolfgang Rosenstiel, and Oliver Bringmann. 2019. Firmware Synthesis for Ultra-Thin IoT Devices Based on Model Integration. In *2019 ACM/IEEE 22nd International Conference on Model Driven Engineering Languages and Systems Companion (MODELS-C)*. IEEE, 339–346.

[17] Hussein M Marah, Raheleh Eslampanah, and Moharram Challenger. 2018. DSML4TinyOS: Code Generation for Wireless Devices. In *International Workshop on Model-Driven Engineering for the Internet-of-Things.*

[18] Elaheh Azadi Marand, Elham Azadi Marand, and Moharram Challenger. 2015. DSML4CP: a domain-specific modeling language for concurrent programming. *Computer Languages, Systems & Structures* 44 (2015), 319–341.

[19] Behailu Negash, Tomi Westerlund, Amir M Rahmani, Pasi Liljeberg, and Hannu Tenhunen. 2017. DoS-IL: A domain specific Internet of Things language for resource constrained devices. *Procedia Computer Science* 109 (2017), 416–423.

[20] Baris Tekin Tezel, Moharram Challenger, and Geylani Kardas. 2016. A metamodel for Jason BDI agents. In *5th Symposium on Languages, Applications and Technologies (SLATE'16)*. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik.

[21] Tansu Zafer Asici, Burak Karaduman, Raheleh Eslampanah, Moharram Challenger, Joachim Denil, and Hans Vangheluwe. Accepted. Applying Model Driven Engineering Techniques to the Development of Contiki-based IoT Systems. In *1st International Workshop on Software Engineering Research & Practices for the Internet of Things (SERP4IoT 2019)*.

APPENDIX: SETUP OF THE CASE STUDIES

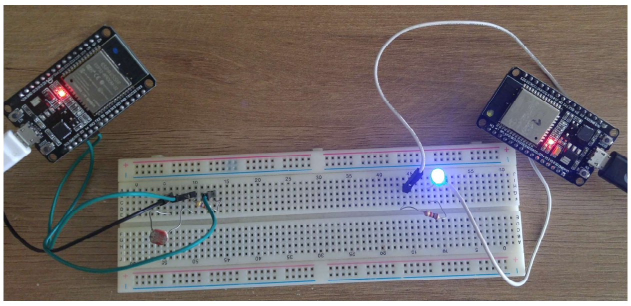


Figure 7: Setup for the smart lighting system

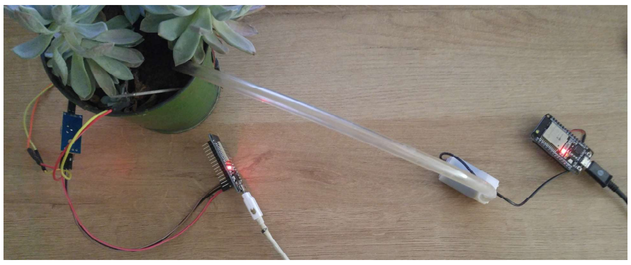


Figure 8: Setup for the smart irrigation system