Modeling and Code Generation Framework for IoT

Mohammad Sharaf[1(✉)], Mai Abusair[1], Rami Eleiwi[2], Yara Shana'a[2], Ithar Saleh[2], and Henry Muccini[3]

[1] Computer Science Department, An-Najah National University, Nablus, Palestine
massharaf@yahoo.com, mai.abusair@gmail.com
[2] Networks and Security Department, An-Najah National University, Nablus, Palestine
rami.ilaiwi1997@gmail.com, yaraadnan177@gmail.com, net.itharsaleh@gmail.com
[3] DISIM Department, University of L'Aquila, L'Aquila, Italy
henry.muccini@univaq.it

Abstract. In the Internet of Things (IoT) every physical device has an embedded technology that interacts with internal and external states. The heterogeneity of devices and networks complicates the mission of implementing and integrating the objects in IoT systems. In this paper, we present our model driven code generation framework, called CAPSml. The framework enables IoT designers and architects who are using CAPS environment to transform CAPS software model into ThingML model. CAPS is an architecture-driven modeling framework for the development of IoT Systems. ThingML includes modeling language and framework designed for IoT systems to support code generation for multi-platform targets.

1 Introduction

Nowadays most systems are relying in their development and evolution on reusing and customizing open-source components, services and frameworks. Model-Driven Engineering (MDE) has been widely used in system development. MDE can enable analysis process, promote communications between system stakeholders, simplify design process and facilitate software production [1].

IoT technologies aim at integrating objects into a communicating environment. A significant challenge in IoT system development is to produce a code that reflects concerns of IoT system specification and design. Accordingly, many issues in IoT systems life cycle are targeted by researchers. The CAPS has been realized to model and analyze IoT architectures [2]. ThingML framework adopted the idea of facilitating code generation for IoT systems [3]. Our approach aims at covering a full chain of modeling and analyzing using CAPS, and then implementing using the power of ThingML code generation.

This paper proposes CAPSml code generation framework built of top of CAPS modeling framework [4,5]. The framework follows MDE approach to

© Springer Nature Switzerland AG 2019
P. Fonseca i Casas et al. (Eds.): SAM 2019, LNCS 11753, pp. 99–115, 2019.
https://doi.org/10.1007/978-3-030-30690-8_6

transform SAML-CAPS model into ThingML model. SAML model represents the software architecture structural and behavioral view in CAPS framework [6]. ThingML model represents software components and configurations that describe their interconnection in ThingML framework [3]. The transformation from SAML to ThingML is performed using CAPSml code generation framework. The aim of CAPSml is to allow IoT developers to mitigate their worries of learning programming languages that implement their IoT systems. In addition, the paper suggests a methodology for modeling, transforming and generating code for IoT systems. It aims to facilitate IoT systems development.

This paper is organized as follows: Sect. 2 presents a brief description for CAPS. Section 3 presents a brief description for ThingML. Then, Sect. 4 introduces the CAPSml code generation framework. Afterward, Sect. 5 provides the modeling and code generation methodology. Section 6 shows a case study example. Finally, Sect. 7 concludes the work.

2 The CAPS Background

CAPS is a modeling framework that was formerly initiated at The University of L'Aquila [4,6]. It has a tool for Architecting Cyber-Physical Systems (CPS) [2,7,8]. The terms CPS and IoT are used interchangeably. They both refer to the integration of digital capabilities, including systems of physical devices and network connectivity [9].

CAPS offers a rich modeling framework that performs a separation of concerns among different modeling views. It is designed and implemented taking into account three architectural views: the software architecture structural and behavioral view (SAML), see Figs. 1 and 2, respectively, the hardware view (HWML), and the physical space view (SPML) [6]. Moreover, CAPS tool provides a graphical user interface for modeling the three views. Accordingly, CAPS provides abstractions for low-level details of the different views, enhances reuse, and supports the ability to model the time and space. Moreover, CAPS allows stakeholders to perform analysis for architectural design decisions at earlier stages of the CPS development life cycle.

In this paper, we focus on SAML view for the sake of performing code generation. SAML modeling allows designers to define a software architecture that is basically constructed of a collection of software Components and Connections: (i) The Component: It is a unit of computation with internal state and defined interface. Each Component can contain several modes that specify its state. The behavior in each Component's mode is denoted by a set of events, actions and conditions. The Components can exchange data by passing messages through message ports. (ii) The Connection: It defines the communication between Components. It sets the source and target Components for the communication channel between two message ports of two different Components. For more information about SAML, refer to the full work in [6].

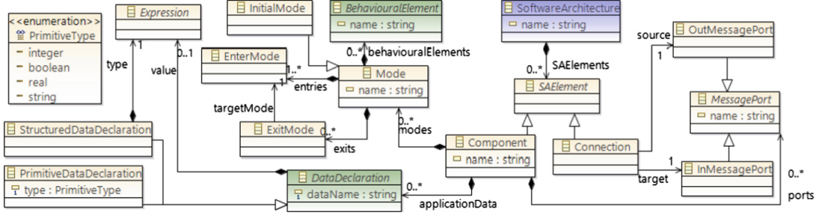

Fig. 1. SAML metamodel: structural concepts [6]

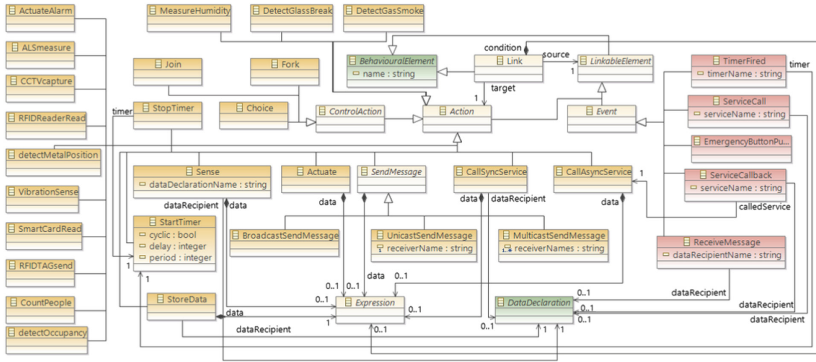

Fig. 2. SAML metamodel: behavioral concepts [6]

3 The ThingML Background

ThingML includes a modeling language combines software modeling constructs for designing and implementing IoT systems [3]. It has an open-source tool designed for supporting code generation and a highly customizable multi-platform code generation framework. ThingML tool targets heterogeneous platforms and has a set of compilers able to transform a ThingML model into fully operational code, in various languages (e.g. C, Java, Javascript), ready to build and run.

The ThingML language is based on two fundamental structures [3], see Fig. 3: (i) Thing: It represents software component. It is an implementation unit, also referred to as process or component. A Thing can assign properties, functions, messages and ports. Moreover, it can contain a set of state machines conforming to the UML state charts. The properties represent the variables that are defined locally inside the Thing. The functions can be treated as local functions inside the Thing and can not be accessed from the outside world. The ports are the only public interface in the ThingML language and they are used to send and receive messages that are defined within a Thing. Further, the internal behavior of a Thing is demonstrated using orchestration of composite states that are expressed using Event-Condition-Action (ECA) fashion. (ii) Configuration: It

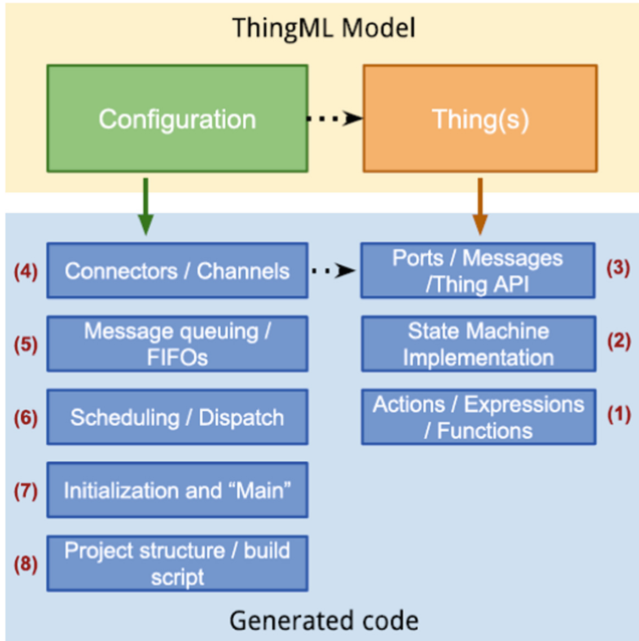

Fig. 3. ThingML model [3]

describes the Things interconnection. It has a set of instances of the pre-defined Things, and a set of connectors between instances ports.

ThingML was developed based on MDE principles. It is used to develop IoT systems ranging from research case studies to product development in industry projects. For more information about ThingML, refer to [3].

4 CAPSml Code Generation Framework

In this section, we will introduce our CAPSml framework that aims to transform SAML model in CAPS into ThingML language. The transformation process in CAPSml starts from the ecore and xmi files of SAML. Then, through several model to text transformations, performed in Acceleo[1], the contents of SAML model is mapped to contents in ThingML model. Finally, a complete ThingML language is generated automatically and is able to be imported directly in ThingML framework.

The fundamental part in the code generation framework is the process of mapping SAML to ThingML. The mapping benefits from the similarities in concepts between the SAML and ThingML models. Basically, the Component in SAML meets the Thing in ThingML; both of them declare a computational unit

[1] https://www.eclipse.org/acceleo/.

which includes a set of behavioral elements like actions, events and conditions. Further, the Connection between the Components in SAML is mapped to the connector in the Configurations part of the ThingML language.

The CAPSml code generation framework passes through four phases to perform model transformation. The preparation, component conversion and connection conversion phases, Sects. 4.1, 4.2 and 4.3, respectively, aim to build the Acceleo code file. The fourth phase, Sect. 4.4, aims to launch the CAPSml framework to be ready for converting CAPS-SAML model xmi file into ThingML file. These phases are described in the following sections.

4.1 Preparation Phase

In this phase, we set the URI of the SAML meta model, see Line 2 Fig. 4. Then, we set the starting point of the conversion at the top class in SAML, which is the SoftwareArchitecture class shown in Line 4 Fig. 4. From this main class we can move gradually to all the software elements indicated in SAML meta model. Further, we set the target name of the file that will have the conversion results (CAPS.thingml), see Line 7 Fig. 4. Finally, we import a special library in ThingML language that will be needed for the data types definitions, see Line 8 Fig. 4.

```
1 [comment encoding = UTF-8 /]
2 [module main('http://ualberta.ssrg.components')/]
3
4 [template public main(element : SoftwareArchitecture)]
5  [comment @main /]
6
7 [file ('CAPS.thingml', false, 'UTF-8')]
8 import "datatypes.thingml" from stl
9
```

Fig. 4. Code generation preparation

4.2 Component Conversion Phase

In this phase, we map the Component with its elements in SAML to the Thing and their correspondences in ThingML. The conversion starts from mapping the Component into Thing, see Fig. 6. Basically, any Component in SAML has primitive data declarations, ports and modes. These are transformed as follows:

1. Primitive data declarations: They are variables defined and used locally during the processes performed inside the Component. Every data declaration is mapped into property in ThingML language, see Fig. 7. Each property represents a local variable to be used inside the Thing. It is important to mention that the real data type in CAPS is converted into float data type in ThingML

```
10 thing fragment Messages {
11  [for (messComp : Component | Components)]
12   [for (dataDec : PrimitiveDataDeclaration | applicationData->
        filter(PrimitiveDataDeclaration)->asSequence())]
13    [if (dataDec.type.toString().equalsIgnoreCase('real'))]
14  message [dataDec.dataName/] (value : Float)
15    [else]
16  message [dataDec.dataName/] (value : [dataDec.type.toString()
                                          .toUpperFirst()/])
17    [/if]
18   [/for]
19  [/for]
20 }
```

Fig. 5. Messages variables transformation

```
22 [for (comp : Component | Components)][comment building Things /]
23 thing [comp.name/] includes Messages {
24
```

Fig. 6. Component transformation

```
25  // ***** Variables *****
26  [for (var : PrimitiveDataDeclaration | applicationData->
       filter(PrimitiveDataDeclaration)->asSequence())]
27  property [var.dataName/] : [if (var.type.toString()
          .equalsIgnoreCase('real'))]Float[else]
           [var.type.toString().toUpperFirst()/][/if]
28  [/for]
29
```

Fig. 7. Data declarations variables transformation

that does not define real data type. In addition, variables that are used in exchanging messages between Components in SAML are mapped to messages in ThingML. We created a special kind for the Thing, named fragment, used to define all the variables to be used in components messages exchanging. Thus, every Component has a message to be sent or received will include the fragment to be able to use the messages values. Transformation of messages variables is shown in Fig. 5.

2. Ports: They are used as an interface for the Component. Each Component in SAML has zero or more message ports. These message ports might be InMessagePort or OutMessagePort. A Connection links the OutMessagePort as a source to InMessagePort as a target. These ports might receive four different types of messages; UnicastSendMessage, BroadcastSendMessage, MulticastSendMessage or a ReceiveMessage. The InMessagePort in SAML is mapped to 'required port' in ThingML, and the OutMessagePort is mapped to 'provided port' in ThingML. The type of message to be sent through the ports in ThingML can be determined using 'sends' and 'receives' elements. In case of broadcast message, we determine only the data to be sent from the 'provided port' in 'sends' and it will reach all connected ports. Otherwise, in unicast

```
41  // ***** Ports *****
42  [for (modePorts : Mode | modes)]
43   [for (it : BehaviouralElement | behaviouralElements)]
44    [if (oclIsKindOf(UnicastSendMessage))]
45     [for (mess : MessagePort | it->
         filter(UnicastSendMessage).toMessagePorts)]
46  provided port [mess.name/] {
47   sends [it->filter(UnicastSendMessage).dataRecipient.dataName/]
48   receives [it->filter(UnicastSendMessage).receiverName/]
49  }
50     [/for]
51    [elseif (oclIsKindOf(BroadcastSendMessage))]
52     [for (mess : MessagePort | it->
         filter(BroadcastSendMessage).toMessagePorts)]
53  provided port [mess.name/] {
54   sends [it->filter(BroadcastSendMessage).dataRecipient.dataName/]
55  }
56     [/for]
57    [elseif (oclIsKindOf(MulticastSendMessage))]
58     [for (mess : MessagePort | it->
         filter(MulticastSendMessage).toMessagePorts)]
59  provided port [mess.name/] {
60   sends [it->filter(MulticastSendMessage).dataRecipient.dataName/]
61      [for (receivers : String | it->
          filter(MulticastSendMessage).receiverNames->asSequence())]
62   receives [receivers/]
63      [/for]
64  }
65     [/for]
66    [elseif (oclIsKindOf(ReceiveMessage))]
67     [for (mess : MessagePort | it->
         filter(ReceiveMessage).fromMessagePorts)]
68  required port [mess.name/] {
69   receives [it->filter(ReceiveMessage).dataRecipient.dataName/]
70  }
71     [/for]
72    [/if]
73   [/for]
74  [/for]
```

Fig. 8. Ports transformation

message, we determine the data to be sent in ‘sends’ and the receiver of the data in ‘receives’. Moreover, in multicast message, we determine the data to be sent in ‘sends’ and the group of selected receivers (filtered out from SAML MultiCastMessage) in ‘receives’. Finally, if the port receives a message, then we set the port name in the ‘required port’ and we determine the data to be received in ‘receives’. Figure 8 shows the ports transformations that take in consideration the different message types.

3. Modes: It represents the behavioral part of the Component. The Component can have several modes in which the initial mode is determined and the orchestration of the entrance and exit for the rest of modes is specified. ThingML has a corresponding similar concept to modes called states. ThingML has a statechart that includes one or more states which illustrate the behavioral execution of the Thing. Each statechart indicates the initial mode in CAPS-SAML as initial state (statechart init) and other modes as states. Typically,

each state has an entry source that can be determined through 'on entry' in ThingML. Mode transformation to state is shown in Fig. 10. Every mode has behavioral elements that describe the concept of the mode's execution. The behavioral elements can mainly be events, conditions and actions, and can also be the links that specify the source and target behavioral elements. Every link between behavioral elements must consider the event that causes the transition from a behavioral element to another, the condition to be checked in a choice and the action to be taken. The action might be nested choice, send message or behavioral functionality. Every concept in these behavioral elements has almost its correspondence in ThingML. Accordingly, every event in CAPS-SAML is mapped to event in ThingML, condition is mapped to guard, link is mapped to transition, and action in CAPS-SAML can be translated to action in ThingML (in case the action is send message or choice), see Fig. 11, or otherwise to function in ThingML. The function can be responsible of several actions like sensing data, store data, actuating, etc. See Fig. 12 to see behavioral element in the mode turned into function. See Fig. 9 that summarizes the mapping between Component and Thing elements.

4.3 Connection Conversion Phase

In this phase, we map the Connection concept in SAML to the Configuration concept in ThingML. The Connection specifies the communication link between Components ports in SAML. Thus, in the configuration part in ThingML, we specify for every Component in SAML an instance in ThingML. The Connection between the target and source ports in SAML is mapped to Connector in ThingML with the required port name => provided port name. See Fig. 13 for Connection transformation. By the end of this phase, we will have "CAPSml.mtl" file that contains the Acceleo code required for ThingML code generation.

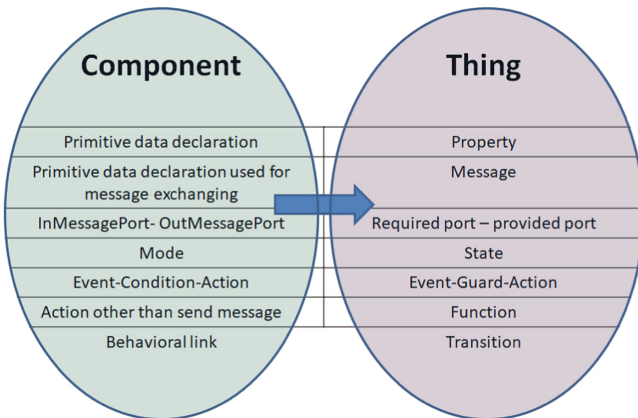

Fig. 9. Mapping between Component elements in SAML and Thing elements in ThingML

```
76  // ***** States *****
77  [for (mode : Mode | modes)][comment display states /]
78  [if (mode.oclIsTypeOf(InitialMode))]statechart init [mode.name/] {[/if]
79   state [mode.name/] {
80    on entry do
81    [for (onEntry : BehaviouralElement | behaviouralElements)]
82     [if (onEntry.eClass().eSuperTypes->last().name.equalsIgnoreCase('Action'))]
83      [if (onEntry->filter(Action).incoming.source.oclIsKindOf(Choice)
       ->asSequence()->first().toString().equalsIgnoreCase('true')._not())]
84     [onEntry.name/]([if (onEntry.oclIsKindOf(StartTimer))]
                         [onEntry->filter(StartTimer).period/]
                        [else][onEntry.eCrossReferences()->
                          filter(PrimitiveDataDeclaration).value/][/if])
85      [/if]
86     [/if]
87    [/for]
88    end
```

Fig. 10. Modes transformation

```
90   [for (behaveChoice : BehaviouralElement | behaviouralElements)]
91    [if (oclIsKindOf(Choice))]
92     [for (linkChoice : Link | behaveChoice->
        filter(Choice).outgoing->asSequence())]
93       [if (linkChoice.target.oclIsKindOf(SendMessage))]
94         [if (linkChoice.target.oclIsKindOf(UnicastSendMessage))]
95   transition -> [mode.name/] event e: [linkChoice.target->
       filter(UnicastSendMessage).toMessagePorts.name/]?[linkChoice.target->
       filter(UnicastSendMessage).receiverName/] guard [linkChoice.condition/]
       action [linkChoice.target->filter(UnicastSendMessage).toMessagePorts.name/]!
       [linkChoice.target->filter(UnicastSendMessage).dataRecipient.dataName/]
       ([linkChoice.target->filter(UnicastSendMessage).dataRecipient.dataName/])
96         [elseif (linkChoice.target.oclIsKindOf(MulticastSendMessage))]
97          [for (receivers : String | linkChoice.target->
              filter(MulticastSendMessage).receiverNames->asSequence())]
98   transition -> [mode.name/] event e: [linkChoice.target->
     filter(MulticastSendMessage).toMessagePorts.name/]?[receivers/] action
     [linkChoice.target->filter(MulticastSendMessage).toMessagePorts.name/]!
     [linkChoice.target->filter(MulticastSendMessage).dataRecipient.dataName/]([linkChoice
      .target->filter(MulticastSendMessage).dataRecipient.dataName/])
99          [/for]
100        [elseif (linkChoice.target.oclIsKindOf(BroadcastSendMessage))]
101  transition -> [mode.name/] guard [linkChoice.target->
     filter(BroadcastSendMessage).dataRecipient.dataName/] action
     [linkChoice.target->filter(BroadcastSendMessage).toMessagePorts.name/]!
     [linkChoice.target->filter(BroadcastSendMessage).dataRecipient.dataName/]
     ([linkChoice.target->filter(BroadcastSendMessage).dataRecipient.dataName/])
102        [/if]
103      [/if]
104    [/for]
105   [/if]
106   [if (oclIsKindOf(Choice))]
107    [for (linkChoice : Link | behaveChoice->filter(Choice).outgoing->asSequence())]
108     [if (linkChoice.target.oclIsKindOf(SendMessage)._not()._and(linkChoice.target
           .oclIsKindOf(ReceiveMessage)._not()))]
109  internal guard [linkChoice.condition/] action [linkChoice.target.name/]
      ([if (behaveChoice.oclIsKindOf(StartTimer))][behaveChoice->
           filter(StartTimer).period/][else]
            [behaveChoice.eCrossReferences()->
              filter(PrimitiveDataDeclaration).value/][/if])
110     [/if]
111    [/for]
112   [/if]
113  [/for]
```

Fig. 11. Choice and messages transformation

4.4 Launching Code Generator Phase

In this phase, we run the Acceleo file that resulted from the first three phases. Therefore, we build a Java launcher project that import the MTCLauncher

```
30  // ***** Behavioural Elements (Functions) *****
31  [for (modeFunc : Mode | modes)]
   [comment display Behavioural Elements in each mode as functions /]
32   [for (behaveFunc : BehaviouralElement | behaviouralElements)]
33    [if (oclIsKindOf(Link)._not()
        ._and(oclIsKindOf(UnicastSendMessage)._not())
        ._and(oclIsKindOf(MulticastSendMessage)._not())
        ._and(oclIsKindOf(BroadcastSendMessage)._not())
        ._and(oclIsKindOf(ReceiveMessage)._not())
        ._and(oclIsKindOf(Choice)._not())
        ._and(oclIsKindOf(TimerFired)._not()))]
34  function [behaveFunc.name/]([if (behaveFunc.oclIsKindOf(StartTimer))]
         val : Float[elseif (behaveFunc.eCrossReferences()->
         filter(PrimitiveDataDeclaration).type->notEmpty())]
         [if (behaveFunc.eCrossReferences()->
          filter(PrimitiveDataDeclaration).type->first()
           .toString().equalsIgnoreCase('real')) ]val : Float
         [else]val : [behaveFunc.eCrossReferences()->
           filter(PrimitiveDataDeclaration).type.toString()
            .toUpperFirst()/][/if][/if]) do
35   // Do something
36  end
37    [/if][comment end if testing /]
38   [/for][comment end Behavioural Elements /]
39  [/for][comment end Modes 1 /]
```

Fig. 12. Behavioral elements transformation

```
135 [/for][comment end Component /]
136 // ***** Configurations *****
137 configuration result {
138 [for (compConf : Component | Components)]
139  instance [compConf.name/]: [compConf.name/]
140 [/for]
141
142 [for (conn : Connection | element.SAElements->filter(Connection))]
143  connector
      [conn.target.eContainer(Component).name/].[conn.target.name/]
      => [conn.source.eContainer(Component).name/].[conn.source.name/]
144 [/for]
145 }
146 [/file]
147
148 [/template]
149
```

Fig. 13. Connection transformation

library [10]. The MTCLauncher is a library developed by a researcher, called Victor Guana, at the University of Alberta [10]. The library allows running ATL model-to-model transformations, and Acceleo model-to-text transformations in an isolated fashion and can be executed in a command line outside Eclipse. It helps in avoiding errors in running Acceleo code in Java environment. In the launcher project, the SAML metamodel ecore file is included under metamodel folder, the CAPSml.mtl file is included under the AcceleoTransformations folder. To start the conversion for any SAML model, we need to open the launcher project and include the SAML model xmi file under models folder in the launcher project. Then, we run the project to get the ThingML output

file (CAPS.thingml) created under the gen folder. The thingml file will contain the ThingML language transformed from SAML model.

The CAPSml framework is able to produce a complete thingml file. It acts as a link between CAPS-SAML and ThingML frameworks. Thus, it enables IoT designers, who are using SAML-CAPS to model and analyze their IoT systems, to benefit from the power of ThingML in code generation. The generated thingml file can be imported in ThingML framework to allow designers to select among several compilers the one that targets their desirable platform.

5 Modeling and Code Generation Methodology

The modeling and code generation methodology can be used during IoT systems life cycle. It benefits from CAPS, CAPSml, and ThingML frameworks. It encompasses three phases illustrated in Fig. 14:

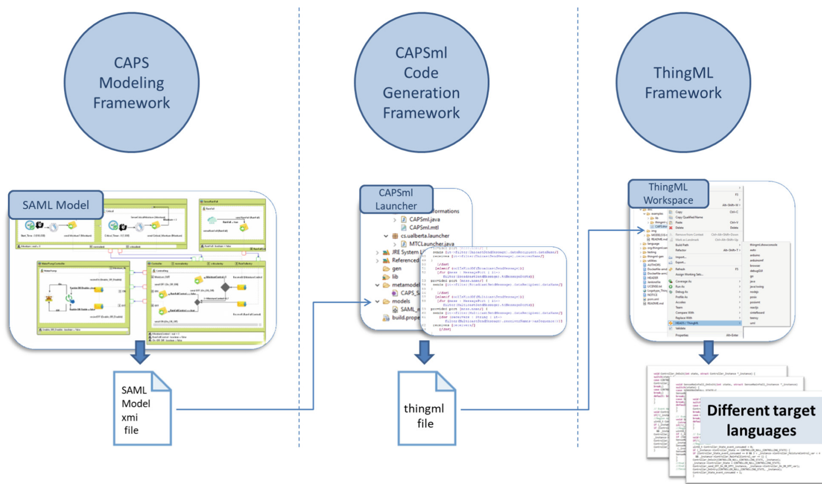

Fig. 14. Modeling and code generation methodology

1. Modeling using CAPS framework: In this phase, the designers architect their IoT systems. They will benefit from the power of CAPS in modeling and analyzing IoT systems [11]. Moreover, they will benefit from the graphical user interface supported by CAPS framework for modeling. The necessary output from this modeling phase is the SAML model xmi file.
2. Running CAPSml framework: In this phase, the designers run CAPSml framework to transform CAPS-SAML model into ThingML model. The transformation process automatically starts from the xmi and ecore files of SAML. Then, the contents of SAML model is mapped to contents in ThingML model. Finally, a complete ThingML language is generated automatically in a thingml file.

3. Running ThingML framework: In this phase, the designers use the thingml file that resulted from phase two for running the ThingML framework. ThingML framework allows the designers to select among several compilers the one that targets their desirable platform.

Following this methodology helps developers to model, analyze and produce IoT systems. It mitigates the developers problems in learning ThingML language and thus the programming languages that can be generated using ThingML framework. In the following section, we show the phases of our methodology on smart irrigation case study example.

6 Smart Irrigation Case Study

The agriculture is one of the most vital resources of nation's economy and food's production. There are many concerns related to traditional methods of agriculture. For example, the excessive wastage of water during irrigation, wastage of money, dependency on human resources, etc. IoT can provide smart solutions for such problems and help in developing agriculture sector in countries.

In this paper, we focus on smart irrigation services. A SAML model is built using CAPS, presented in Sect. 6.1. By using our CAPSml code generator, the SAML model will be transformed into ThingML language, presented in Sect. 6.2. Consequently, using ThingML framework, the ThingML language is used to generate different target languages, presented in Sect. 6.2.

6.1 Describing a Scenario Using CAPS

We introduce a simple scenario describes the monitoring of soil moisture and climate condition in order to change the work of the water pump in a filed [12]. This scenario aims at preventing the wastage of water resources [13].

SAML model of the scenario is shown in Fig. 15. It is important to note that this Figure is a screenshot of modeling using the graphical user interface supported by CAPS tool. The SAML model is composed of four components:

1. The SenseMoisture component: It is responsible for sensing the soil moisture value. It includes two modes:

- Normal mode: In this mode, the moisture sensor senses the moisture value from the soil every 100 s. Then, it saves the value in Moisture primitive variable. After all, it uses the unicast message to send the values to the Controller component. If the moisture value is more than 3, the SenseMoisture component enters the Critical mode.
- Critical mode: In this mode, the sensor senses the moisture value every one second. It saves the value in Moisture primitive variable. Then, it uses the unicast message to send the value to the Controller component. If the moisture value is less than 3, the SenseMoisture component enters the Normal mode.

2. The SenseRainfall component: It is responsible for sensing the rainfall. It includes one mode that is RainFall.

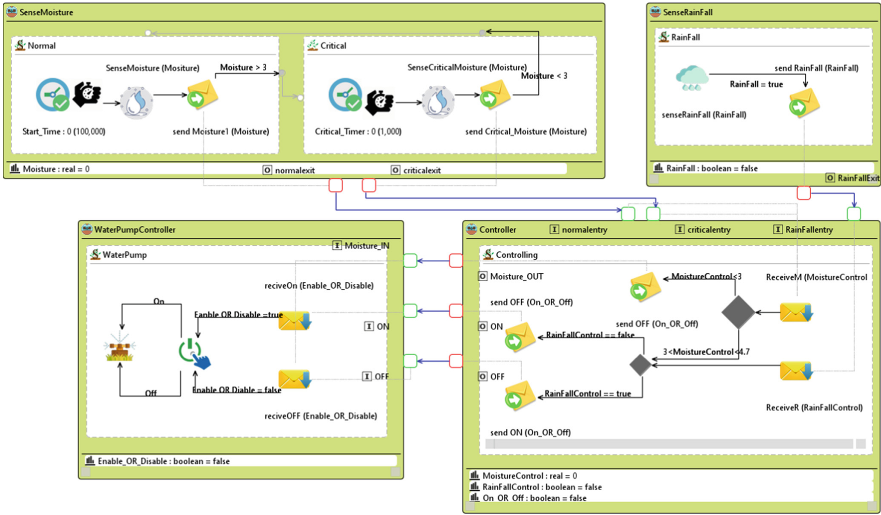

Fig. 15. Software architecture of simple scenario in smart irrigation case study

– Rainfall Mode: In this mode, there is an interrupt sensor that senses if there is a rainfall or not. The value taken from this sensor is kept in a primitive variable. It uses a unicast message to send the value to the Controller component.

3. The Controller component: It is responsible for making decisions to turn the water pump on or off. It includes one mode that is Controlling.

– Controlling mode: In this mode, the values received from the Moisture and Rainfall messages are stored in primitive variables. These values are used for making decisions depending on the current condition. If the Moisture is more than 3 and less than 4.7 and the weather does not rain, the Controller sends a message to the water pump to turn it on. While, if the Moisture is more than 3 and less than 4.7 and the weather is Rainfall, the Controller sends a message to the water pump to turn it off. Moreover, if the Moisture is less than 3 the Controller sends a message to the water pump to keep it off.

4. The WaterPumpController component: It is responsible for turning the pump on or off depending on the decision from the Controller. It includes one mode that is WaterPump.

– WaterPump mode: It receives a message from the Controller component and stores it in a primitive variable. The value stored in the primitive variable specifies if the pump is turned on or off. This value is sent to an actuator. If the sent value is true, the actuator turns the pump on. If the sent value is false, the actuator turns the pump off.

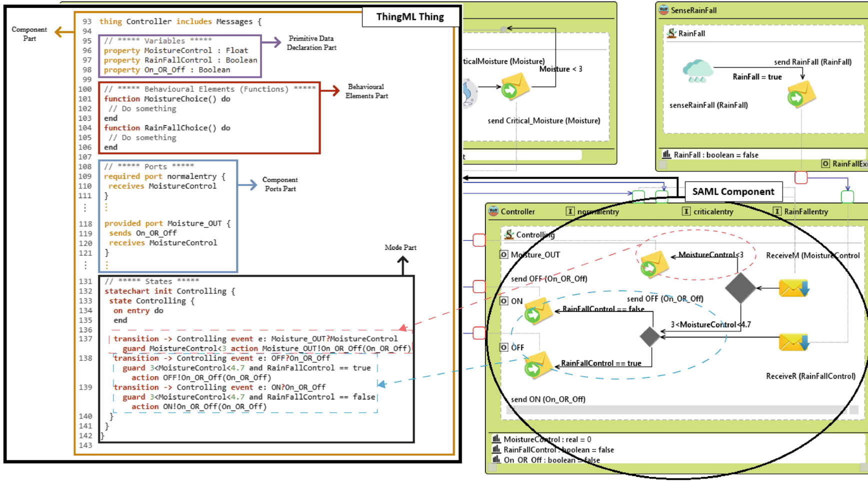

Fig. 16. Example of the Controller component in SAML converted into Controller Thing in ThingML language

6.2 Code Generation Using CAPSml

In this section, we describe the results of running SAML model on the CAPSml framework. Then, we show the code generated using ThingML code generation framework.

```
180 // ***** Configurations *****
181 configuration result {
182     instance SenseMoisture: SenseMoisture
183     instance SenseRainFall: SenseRainFall
184     instance Controller: Controller
185     instance WaterPumpController: WaterPumpController
186
187     connector Controller.RainFallentry => SenseRainFall.RainFallExit
188     connector Controller.normalentry => SenseMoisture.criticalexit
189     connector Controller.criticalentry => SenseMoisture.normalexit
190     connector WaterPumpController.ON => Controller.ON
191     connector WaterPumpController.OFF => Controller.OFF
192     connector WaterPumpController.Moisture_IN => Controller.Moisture_OUT
193 }
```

Fig. 17. Part of the generated ThingML Configuration

Before running CAPSml framework, we need to specify the model, described in Fig. 15, in xmi format and the meta model of SAML in ecore format. Then, to run CAPSml framework launcher project, we need to find the xmi file for the model generated using CAPS-SAML. After running CAPSml launcher, we will automatically have a thingml file that has a complete ThingML language in the gen folder of the launcher project. Figure 16 shows part of converting

```
3⊖ thing fragment Messages {
4      message Moisture (value : Float)
5      message RainFall (value : Boolean)
6      message MoistureControl (value : Float)
7      message RainFallControl (value : Boolean)
8      message On_OR_Off (value : Boolean)
9      message Enable_OR_Disable (value : Boolean)
10 }
```

Fig. 18. Part of the generated Thing fragment

```
49 // On Exit Actions:
50 void Controller_OnExit(int state, struct Controller_Instance *_instance) {
51 switch(state) {
52 case CONTROLLER_STATE:{
53 Controller_OnExit(_instance->Controller_State, _instance);
54 break;}
55 case CONTROLLER_NULL_CONTROLLING_STATE:{
56 break;}
57 default: break;
58 }
59 }
60
61 // Event Handlers for incoming messages:
62 void Controller_handle_OFF_On_OR_Off(struct Controller_Instance *_instance, bool value) {
63 if(!(_instance->active)) return;
64 //Region null
65 uint8_t Controller_State_event_consumed = 0;
66 if (_instance->Controller_State == CONTROLLER_NULL_CONTROLLING_STATE) {
67 if (Controller_State_event_consumed == 0 && 3 < _instance->Controller_MoistureControl_var < 4.7
     && _instance->Controller_RainFallControl_var == 1) {
68 Controller_OnExit(CONTROLLER_NULL_CONTROLLING_STATE, _instance);
69 _instance->Controller_State = CONTROLLER_NULL_CONTROLLING_STATE;
70 Controller_send_OFF_On_OR_Off(_instance, _instance->Controller_On_OR_Off_var);
71 Controller_OnEntry(CONTROLLER_NULL_CONTROLLING_STATE, _instance);
72 Controller_State_event_consumed = 1;
73 }
74 }
```

Fig. 19. Part of C++ code generated for Controller Thing/Component

Controller component in SAML into Controller Thing in ThingML language. Figure 17 shows part of the generated Configuration for the Things in ThingML language. Figure 18 shows part of the generated Thing fragment that contains the messages to be exchanged between components.

Consequently, the ThingML language, that resulted from running CAPSml, was used to run ThingML code generator framework for producing several target languages. We experimented the code generation using different compilers supported by ThingML framework. The results show successful transformations to different targeted languages and platforms. For the sake of space, we present a snippet for the results of only running Posix compiler in the ThingML framework. Posix Generates C/C++ code for Linux or other Posix runtime environments. Figure 19 shows part of the C++ code generated for the Controller Component/Thing.

7 Conclusions

In this paper, we presented CAPSml, a code generation framework built on top of CAPS modeling framework and targets ThingML framework. CAPS framework offers a graphical user interface that facilitates the production of IoT systems architecture. While, ThingML offers a code generation framework that brings MDE to the late design and implementation stages.

CAPSml transforms CAPS model into ThingML language. Thus, CAPS users can generate code for their models using ThingML framework. Moreover, CAPS users do not need to learn ThingML modeling language. To show the utilization of our tool, we ran an example on smart irrigation case study and clarified how our code generation approach can take place in IoT systems life cycle.

References

1. Ciccozzi, F., Spalazzese, R.: MDE4IoT: supporting the internet of things with model-driven engineering. Intelligent Distributed Computing X. SCI, vol. 678, pp. 67–76. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-48829-5_7
2. Muccini, H., Sharaf, M.: Caps: a tool for architecting situational-aware cyber-physical systems. In: 2017 IEEE International Conference on Software Architecture Workshops (ICSAW), pp. 286–289. IEEE (2017)
3. Harrand, N., Fleurey, F., Morin, B., Husa, K.E.: 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, pp. 125–135. ACM (2016)
4. Sharaf, M., Abughazala, M., Muccini, H., Abusair, M.: An architecture framework for modelling and simulation of situational-aware cyber-physical systems. In: Lopes, A., de Lemos, R. (eds.) ECSA 2017. LNCS, vol. 10475, pp. 95–111. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-65831-5_7
5. Sharaf, M., Muccini, H., Abughazala, M.: ArIA: arduino code generation based on the caps. In: Proceedings of the 12th European Conference on Software Architecture: Companion Proceedings, p. 4. ACM (2018)
6. Muccini, H., Sharaf, M.: Caps: architecture description of situational aware cyber physical systems. In: 2017 IEEE International Conference on Software Architecture (ICSA), pp. 211–220. IEEE (2017)
7. Sharaf, M., Abughazala, M., Muccini, H., Abusair, M.: CAPSim: simulation and code generation based on the CAPS. In: Proceedings of the 11th European Conference on Software Architecture: Companion Proceedings, pp. 56–60. ACM (2017)
8. Sharaf, M., Abughazala, M., Muccini, H., Abusair, M.: Simulating architectures of situational-aware cyber-physical space. In: Proceedings of the 11th European Conference on Software Architecture: Companion Proceedings, pp. 66–67. ACM (2017)
9. Yan, Z., Zhang, P., Vasilakos, A.V.: A survey on trust management for internet of things. J. Netw. Comput. Appl. **42**, 120–134 (2014)
10. Guana, V.: Running Acceleo and ATL Transformations Programmatically. University of Alberta (2016). http://victorguana.blogspot.com/2016/05/running-acceleo-and-atl-transformations.html

11. Sharaf, M., Abughazala, M., Muccini, H.: Arduino realization of CAPS IoT architecture descriptions. In: Proceedings of the 12th European Conference on Software Architecture: Companion Proceedings, p. 6. ACM (2018)
12. Sharaf, M., Abusair, M., Eleiwi, R., Yara, S., Ithar, S., Muccini, H.: Architecture description language for climate smart agriculture systems. In: Proceedings of the 13th European Conference on Software Architecture: Companion Proceedings. ACM (2019)
13. Gondchawar, N., Kawitkar, R.: IOT based smart agriculture. Int. J. Adv. Res. Comput. Commun. Eng. **5**(6), 838–842 (2016)