Mapping from CAPS Software Architecture Modeling Language (SAML) to ThingML Language using Acceleo Code Generator

Ithar Saleh, Yara Shanaa, Rami Ilaiwi

Department of Networks and Information Security

An-Najah National University

Supervisor

Dr. Mohammad Sharaf

In partial fulfillment of the requirements for the degree of

Bachelor in Networks and Information Security

May, 2019

Abstract

The intent of this thesis was to provide a mechanism to map and convert between two different IOT modeling languages: CAPS-SAML and ThingML. This mapping and conversion was needed in order to have automatic generated codes from models. This required great knowledge of both languages and having a tool or a mechanism in between to do such a conversion which would be built on scientific and logical grounds . Our thesis presents our work which covers up all stages from the design of the SAML models using CAPS framework, passing by using the Acceleo code generator which does the model-to-text transformation from SAML to ThingML and ending by the final results of the ThingML code which will be used to generate different programming languages codes.

Contents

List of Figures

Chapter 1

Introduction

1.1 Motivations

The Internet of Things can be simplified and defined as a dynamic network which includes its own configuration and uses the standard communication protocols. Those things would have their requirements, specifications and attributes. IOT systems needs vary from reliability, security, availability and integrity but they also need a level of compliance, scalability and other important performance metrics.

Model Driven Engineering comes to meet those needs and requirements ever after it focuses on exploiting domain models. These models will target all the topics related to a specific problem related to this domain. By this, Model Driven Engineering (MDE) is meant to give an abstract knowledge about a particular domain and by taking into consideration all related activities. MDE aims to increase the level of productivity by using standardized models which will increase the level of compatibility between systems. In short, Software architecture aims to make fundamental structural choices which are costly to change after implementation.

1.2 Context of the Study

CAPS is a model driven engineering framework, used to describe the software architecture, hardware configuration and the physical space views for situational aware Cyper physical systems(CPS). It aims to include all things relate to Software in one view, as well as in the Hardware and the physical space views. This will support the concept of separation of concerns, since each view will be worked and focused on in an isolation of the others. This means the software developer won't need to know about the hardware requirements or about the space dimensions. CAPS also has two additional views in order to link the three modeling views together. These two views are denoted as Mapping Modeling Language (MAPML) and Deployment Modeling Language (DEPML).

What we need to focus on here is the Software Architecture view in CAPS. This view looks more in deep to the Software elements and its way of behaviour and structure. In CAPS, the Software Modeling Language (SAML) is composed of two main elements: Components and Connections. SAML describes how components and connections exchange messages through message ports. Each component can declare a set of modes, each one of these modes can contain a set of events, conditions and actions. All this represent the behaviour of the component. The component also has application data which are defined inside the component and manipulated using the actions.

On the other hand ThingML is a language and code generation framework, which has been used in the development of the IOT systems. ThingML language allows developers to customize the code generators for their specific needs. ThingML code generation has been used to generate code in C/C++, Java and JavaScript, and many more languages are under development these days.

ThingML has been used at first in the embedded systems like sensor networks in the oil and gas domains. Right now ThingML is being generally applied

to the IOT domain. Another advantage of ThingML is that it works well for heterogeneous platforms and even heterogeneous communication protocols.

1.3 Objectives and Contributions

Both CAPS and ThingML can be considered as domain specific modeling languages (DSML) when comparing to UML, but they can be used for a wide range of applications since they're not limited to a specific domain of business.

Since ThingML aims to model software components and automatically generate modules of code ready to be deployed and implemented, if we could find a way to convert from CAPS-SAML into ThingML, then we will be able to have multiple modules of codes for the same model designed using SAML. This would give us larger variety, scalability and better efficiency.

We have done this by using the Acceleo code generator. Acceleo is an open-source code generator from the Eclipse Foundation which uses the model driven engineering principles and approaches to build applications. The main aim of the Acceleo is to perform the model to text transformation.

1.4 Overview of the Thesis

This thesis clarifies the process of mapping and converting between two modeling languages, using the model-to-text transformation principles by applying the Acceleo code generator.

Chapter 2

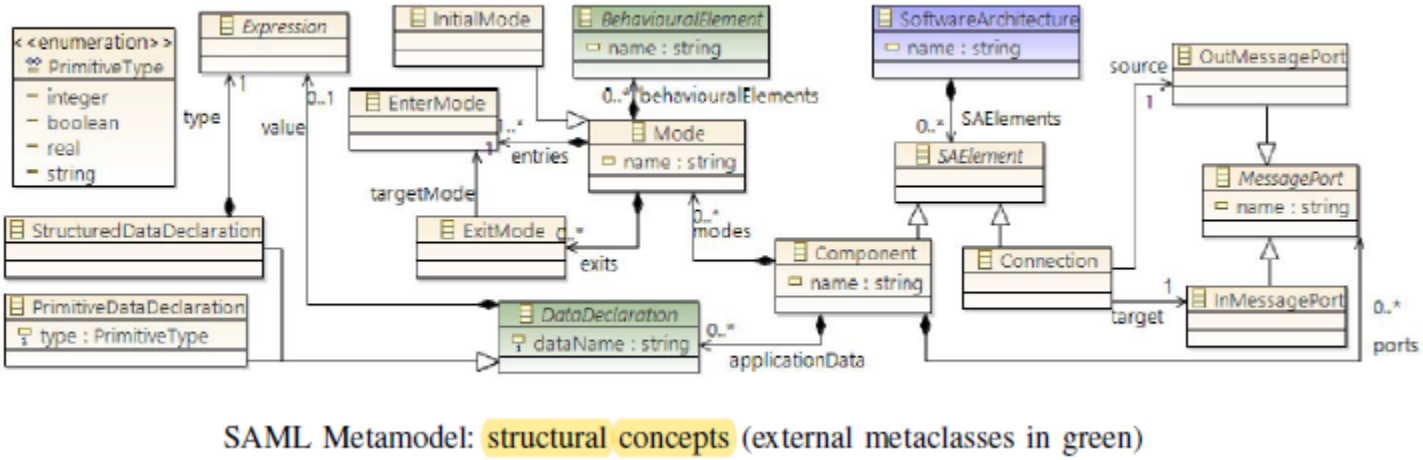
CAPS-SAML

Summary

CAPS was designed and implemented based on the distribution of the modeling information into 3 architectural views: the software structural and behavioural view (SAML), the hardware view (HWML) and the physical space view (SPML). This separation will enhance the principle of separation of concerns, since every stakeholder will focus on a specific field of concerns. CAPS also has a way to combine these three modeling views together, in order to tell those software elements defined in the software model have those specific hardware requirements which are defined in the hardware model. This is done by the MAPML which does the mapping between the software and the hardware models. In a same manner, the DEPML links the hardware model with the space view model, to specify the location of each hardware element. [1]

Our focus in this work will be on the software modeling language (SAML) which will be converted into ThingML. CAPS allows software architects to define the software architecture of the IOT system through using the SAML modeling language.

2.1 SAML Meta Model



SAML Metamodel: structural concepts (external metaclasses in green)

Figure 2.1: structural concepts SAML MetaModel

SAML basically is composed of components and connections between them. These components will exchange information using the message ports. A component can be defined as a unit of computation with internal state and well-defined interface [1]. The behavior of each component is determined by a set of events, actions and conditions. Local variables can also be defined inside the component scope which specifies the application data that will be deployed by the events, actions and conditions.

The Component can also contain several modes which will specify the state of this component, for example a mode can be energy saving mode or a sleeping mode. Component can have only one active mode at a time to determine the status of it. A mode has an action which is used for message passing from a mode to another, this is called ExitMode which will pass messages into an event called EnterMode. Every mode can contain several behavioral elements that all together will represent the control flow of the component.

2.1.1 SAML behavioral elements

The behavioral elements in the software modeling language can be classified as one of actions, events or conditions. An action in CAPS represents an atomic task that can be performed by the component [1]. This action can be achieved when an event is triggered or when a previous action in the control flow has been completed. An action can be for example send a message to a specif port, start or stop a timer.
Another type of the behavioral elements is events. An event is made in response to some internal tool in the component like Timer fired or an external motivation like a received message.
Events and actions are connected together using Links, which determine the control flow. These links can be used to determine the order in which actions will be executed and the actions which must be performed when an event is triggered.

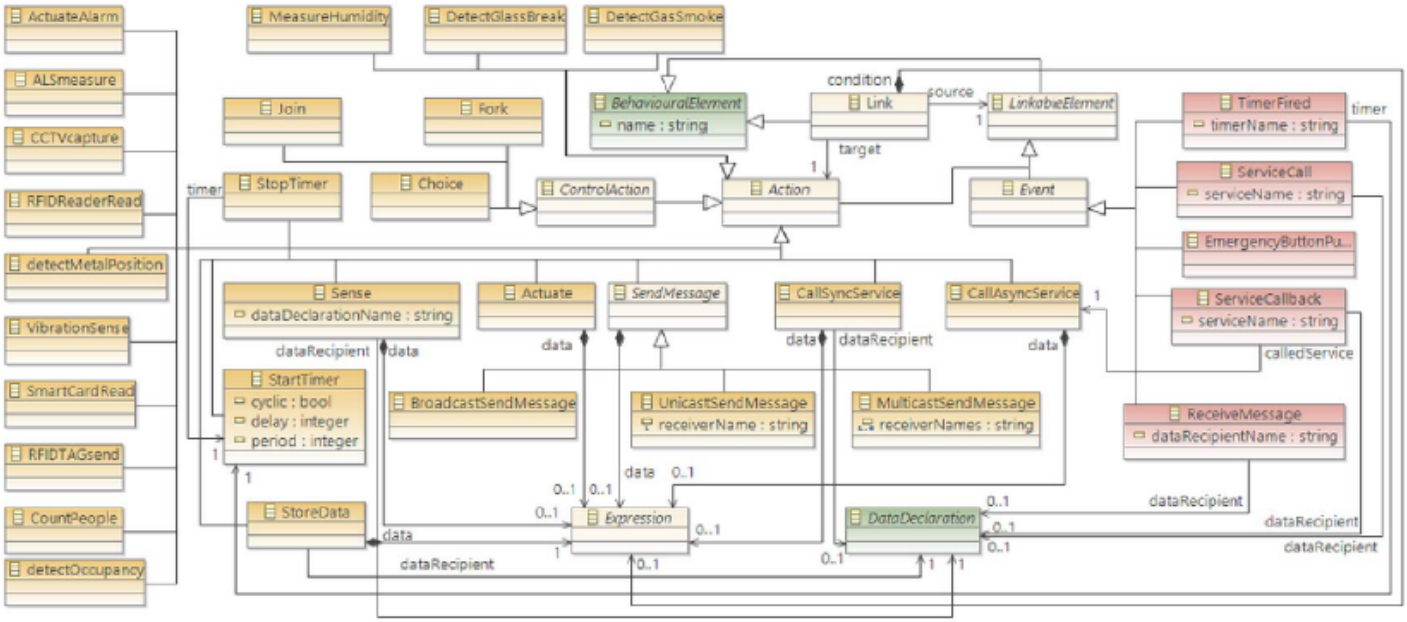


Figure 2.2: behavioral concepts SAML MetaModel

Chapter 3

ThingML Architecture

Summary

ThingML is considered to be a language and code generation framework for heterogeneous systems. This approach came to support the process of generating codes from models. ThingML includes a modeling language a tool designed to support code generation and a customizable multi-platform code generation framework [2]. ThingML has been used to develop systems ranging from research case studies to product development in industry projects. ThingML targets distributed IOT systems and gives a great advantage when applying in heterogeneous platforms.

ThingML was been developed based on the Model Driven Engineering (MDE) principles. While models in MDE can be really helpful in the requirements or resting phases, ThingML aims to bring MDE to the late design and the implementation phases of the software life cycle as well as to support the maintenance and evolution tasks [2].

3.1 ThingML DSL

ThingML language is composed of two key structures: Things which represent the software components and Configurations which describe their interconnections. A Thing can be defined as an implementation unit, a component or a process [2].

A Thing can assign properties, functions, messages and ports. It can also contain a set of state machines. The properties variables are defined locally inside the thing and can be accessed from within. Functions also can be treated as local functions inside the thing and can't be accessed or seen from the outside. Ports are the only public interface in the ThingML language. They are used to send or receive messages which are defined within a thing but can only be used for sending or receiving through the message ports.

The internal behavior of a Thing can be achieved using a combination of:

- Fundamental programming to clarify the procedures either by using the ThingML platform actions and expression language or by using the features of the target language. Libraries from the targeted languages can be wrapped and used. A mix of the two options can be implemented.
- Event-Condition-Action (ECA) which defines simple if in order to apply rules in response of the occurrence of events.
- Composite state machine, to react and coordinate events in a stateful fashion. This conforms to the UML charts.

3.1.1 ThingML Code generation Framework

ThingML code generation framework is composed of a family of compilers which are able to transform a ThingML model into a fully operational code in different languages [2]. By today, ThingML has 4 fully supported languages which are Java, C, JavaScript and more languages are under development.

Chapter 4

Acceleo Code generator

Summary

Acceleo is an open- source code generator developed by the Eclipse Foundation. It permits using model driven engineering approaches in order to get or build an application. It performs the model-to-text transformation. Acceleo was written in Java and deployed as a plugin in the Eclipse platform. It supports different platforms like Windows, Linux or MAC.

Acceleo provides a mechanism for generating codes from Eclipse Modeling Framework (EMF) based models. Acceleo allows to generate from any kind of MetaModel compatible with EMF. It also allows the customization of the generation for a user defined template and it can generate to any text language like C, Python, Java, etc.

4.1 Transformation from CAPS-SAML to ThingML

In order to run Acceleo, we need to specify the model in xmi format, MetaModel of the source model in ecore format. We also need to specify the location of the generated file and the generation file which will contain the syntax of the transformation from the model to the text.

```
launcher.runAcceleo("metamodels/SoftwareArch.ecore", "capsSAML", "models/test.xmi",
        "transformations/M2T/main.mtl", "gen/");
```

Figure 4.1: Running Acceleo

The mapping and converting between SAML and ThingML happened by the mapping between every element in SAML to a corresponding one in ThingML. By the analysis of both SAML and ThingML, the component in SAML meets the Thing in ThingML, since both of them declare a computational unit which includes a set of behavioral elements like actions, events and conditions. The connections between the components in SAML have been mapped to the connectors in the Configurations part in the ThingML language.

We started by using the xmi file of SAML model and by starting from the highest point in the model which is the Software architecture in SAML, then it goes through the SA-Element to reach the component which is mapped to Thing in the ThingML language. We then went gradually in the xmi code to complete the mapping.

The mode in CAPS-SAML was mapped to state in ThingML and the initial mode was mapped to the initial state. The behavioral elements in SAML like events, actions and conditions can be mapped directly to the corresponding events, actions and conditions in ThingML.

By this way, SAML would be mapped completely by the end of this project into ThingML and this would give us the ability of having multiple modules of

```
1  [comment encoding = UTF-8 /]
2  [module main('http://ualberta.ssrg.components')/]
3
4  [template public main(element : SoftwareArchitecture)]
5   [comment @main /]
```

Figure 4.2: Software Architecture in SAML

```
7  [file ('index.thingml', false, 'UTF-8')]
8  import "datatypes.thingml" from stl
```

Figure 4.3: Output File name in ThingML

different languages from one model. This work would decrease the amount of time, work and money exploited in order to compile the model into each one of these languages.

4.1.1 Structural Concepts of SAML and ThingML

The software modeling language (SAML) MetaModel in CAPS has two types of concepts: Structural and behavioral concepts. The structural concepts of SAML MetaModel declares that any software element would be a component or a connection. This has been mapped into Thing and connector in ThingML language.

We started the conversion by selecting the top element in SAML which is the Software Architecture since it's the main class in SAML so we can access all other classes in SAML. This is shown in Figure (4.2).

We defined the target file name we want to see the result of the conversion in and imported a special library in ThingML language for the data types definitions, as shown in Figure (4.3).

We could after that start converting from a component into Thing. The conversion was done as shown in Figure (4.4). Any component in SAML will

have some primitive data declarations and behavioral elements. In order to do such a conversion, primitive data declarations were mapped into the variables in ThingML language, since those data declarations will contain the values each behavioral element will produce or work on. The behavioral elements were mapped into functions which will abstract their functionality. In our mapping to Functions we excluded some behavioral elements which will be discussed later.

To achieve the connections between Components into ThingML, we mapped them into the configuration section in the ThingML language, this was done as shown in Figure (4.5). The configuration part will contain an instance of every component/thing implemented in the code and will implement a connector between every two connected components by using the source and target names of the connection in SAML.

The messages between components will contain a value of a behavioral element, this value will be a modification on some primitive data declaration value. Since these messages are parts of the components and would have a value, we have created a special kind of thing called fragment to include all primitive data declaration values which will be used in sending or receiving messages, so that every component has a message will just include this fragment to be able to use their values. This is shown in Figure (4.6)

Each Component in SAML has 0 or more message ports. These message ports might be InMessagePort or OutMessagePort. A connection will link between the OutMessagePort as a source to another InMessagePort as a target. These ports might receive 4 different types of messages, UnicastSendMessage, BroadcastSendMessage, MulticastSendMessage or a ReceiveMessage. This mapping and converting was done as shown in Figure (4.7).

```
22 [for (comp : Component | Components)][comment building Things /]
23 thing [comp.name/] includes Messages {
24
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
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 /]
40
```

Figure 4.4: converting a component into Thing

```
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
```

Figure 4.5: converting connections 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 }
```

Figure 4.6: converting PrimitiveDataDeclaration values in ThingML

```
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]
75
```

Figure 4.7: converting PrimitiveDataDeclaration values 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
```

Figure 4.8: Enter Mode in SAML to On Entry in ThingML

The final part of the mapping to introduce is Modes. A mode in SAML could be an initial mode or simply just a mode. The difference between them is that a component will start its execution from the initial mode and then move to another mode by following the behavioral links.

ThingML has a corresponding similar concept to Modes called States. ThingML has a statechart which includes one or more states that illustrates the execution of the Thing. Each statechart will specify the initial mode as initial state and other modes as states. Each state typically will have an enter mode and exit mode. The enter mode will be mapped into 'on entry' in ThingML which will start with the first behavioral element in the mode and then it leads to other behavioral elements in the same order they will be executed in. This has been done as shown in Figure (4.8).

Choice element was the most complicated one to convert, we needed to look for every possible choice could be taken. The first type of choices we selected to focus on is the messages. Messages could be unicast, multicast or broadcast. After determining the type of message, then we need to specify the transition movement. This has been done as shown in Figure (4.9). Transition will specify the event, condition and the action which will be taken.

The other option for Choice if not a message, is to be a normal behavioral element, this is shown in Figure (4.10).

The last behavioral element to cover is component messages in its 3 types. This has been covered here to provide the transition from one place to another in addition to the guards or the conditions. This has also been done as shown in Figure (4.11).

4.1.2 SAML and ThingML

By this, any SAML model can be mapped and converted into ThingML easily. The structure of SAML is summed up with a component, connections, modes, behavioral elements and primitive data declarations. ThingML is consisting of Thing, configuration, variables, ports and statechart.

```
[for (behaveChoice : BehaviouralElement | behaviouralElements)]
 [if (oclIsKindOf(Choice))]
  [for (linkChoice : Link | behaveChoice->
     filter(Choice).outgoing->asSequence())]
    [if (linkChoice.target.oclIsKindOf(SendMessage))]
      [if (linkChoice.target.oclIsKindOf(UnicastSendMessage))]
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/])
      [elseif (linkChoice.target.oclIsKindOf
               (MulticastSendMessage))]
       [for (receivers : String | linkChoice.target->
        filter(MulticastSendMessage).receiverNames->asSequence())]
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/])
       [/for]
      [elseif (linkChoice.target.oclIsKindOf
       (BroadcastSendMessage))]
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/])
      [/if]
    [/if]
  [/for]
 [/if]
```

Figure 4.9: Choice messages in ThingML

```
 [if (oclIsKindOf(Choice))]
  [for (linkChoice : Link | behaveChoice->
     filter(Choice).outgoing->asSequence())]
   [if (linkChoice.target.oclIsKindOf(SendMessage)._not()
        ._and(linkChoice.target
         .oclIsKindOf(ReceiveMessage)._not()))]
internal guard [linkChoice.condition/]
  action [linkChoice.target.name/]
([if (behaveChoice.oclIsKindOf(StartTimer))][behaveChoice->
  filter(StartTimer).period/][else][behaveChoice
   .eCrossReferences()->filter(PrimitiveDataDeclaration)
    .value/][/if])
   [/if]
  [/for]
 [/if]
[/for]
```

Figure 4.10: Choice behavioral elements in ThingML

```
  [for (messages : BehaviouralElement | behaviouralElements)]
   [if (messages.oclIsKindOf(UnicastSendMessage)
    ._and(messages->filter(UnicastSendMessage).incoming.source
     .oclIsKindOf(Choice).toString()->asSequence()->first()
      .equalsIgnoreCase('true')._not()))]
  transition -> [mode.name/] event e: [messages->
    filter(UnicastSendMessage).toMessagePorts.name/]?
   [messages->filter(UnicastSendMessage).receiverName/] action
   [messages->filter(UnicastSendMessage).toMessagePorts.name/]!
   [messages->filter(UnicastSendMessage).dataRecipient.dataName/]
  ([messages->filter(UnicastSendMessage).dataRecipient.dataName/])
   [elseif (messages.oclIsKindOf(MulticastSendMessage)
     ._and(messages->filter(MulticastSendMessage).incoming.source
      .oclIsKindOf(Choice).toString()->asSequence()->first()
       .equalsIgnoreCase('true')._not()))]
    [for (receivers : String | messages->
      filter(MulticastSendMessage).receiverNames->asSequence())]
  transition -> [mode.name/] event e: [messages->
    filter(MulticastSendMessage).toMessagePorts.name/]?[receivers/]
    action [messages->filter(MulticastSendMessage)
     .toMessagePorts.name/]!
   [messages->filter(MulticastSendMessage).dataRecipient.dataName/]
  ([messages->filter(MulticastSendMessage).dataRecipient.dataName/])
    [/for]
   [elseif (messages.oclIsKindOf(BroadcastSendMessage)
    ._and(messages->filter(BroadcastSendMessage).incoming.source
     .oclIsKindOf(Choice).toString()->asSequence()->first()
      .equalsIgnoreCase('true')._not()))]
  transition -> [mode.name/] guard [messages->
    filter(BroadcastSendMessage).dataRecipient.dataName/] action
   [messages->filter(BroadcastSendMessage).toMessagePorts.name/]!
   [messages->filter(BroadcastSendMessage).dataRecipient.dataName/]
  ([messages->filter(BroadcastSendMessage).dataRecipient.dataName/])
   [/if]
  [/for]
  [if (comp.modes->asSequence()->size()>1)]
   [for (exitM : ExitMode | exits)]
  transition -> [exitM.targetMode.eContainer(Mode).name/]
   guard [exitM.incoming.condition/]
   [/for]
  [/if]
 }
[/for][comment end Modes 2 /]
```

Figure 4.11: Messages in ThingML

Chapter 5

Experimental Results

5.1 CSA-CAPS: a special version of CAPS for Smart Agriculture

Climate Smart Agriculture stands for the employment of recent Information and Communication Technologies (ICT) into Agriculture, which would make a great difference in the production levels. Smart Agriculture aims to use modern management techniques in order to monitor, analyze and take decisions based on some predefined criteria. Rather than applying the same amount of fertilizers over an entire agricultural field, using a special framework designed for smart agriculture would be a great enhancement and modification towards a better future. Farming those days are consuming huge amounts of water, energy, fertilizers, pesticides, etc, CSA-CAPS would save a lot of money, time and resources by the pre-design and analysis before the real implementation of the system in the reality. This section presents Climate Smart Agriculture-CAPS (CSA-CAPS), a specialized modeling framework for smart agriculture. This framework targets farmers, government and any interested group.

5.1.1 CSA-CAPS TOOL

CSA-CAPS comes as a modified version from CAPS, adjusted to fit well with the Smart Agriculture field. This has been done by studying the Agriculture and Farming domain, analyzing the needs and requirements and monitor the outcomes. In order to make the Agriculture as smart as possible, we needed to automate all possible duties. We gathered all services, sensors and other facilities which could be helpful in our work.

All needed sensors and devices have been added to CAPS in order to make it fit with our field. Those sensors varies from sensing temperature, moisture, CO to sensing the amounts of minerals inside the soil itself. This modification was done only on the software architecture viewpoint, since the hardware components wont change from a sensor or another.

Farmers in their primitive ways of farming have been using random amounts of water, fertilizers and pesticides without any previous knowledge about the real needs. If we want to have smart Agriculture then we need more effective way of farming. SA-CAPS was found to improve the quality and style of Agriculture. It aims to develop the agricultural practices in order to adapt to and decrease the impacts of climate change and at the same time to increase the food production as possible.

Smart Agriculture aims to reduce water consumption by having better water retention inside the soil. It aims also to keep the nutrients available most of the time so an increased organic material accumulation will be gained.

5.1.2 Smart-Agriculture Case Studies

CSA-CAPS is a smart agriculture framework, so we have prepared 3 different case studies relevant to this domain as follows.

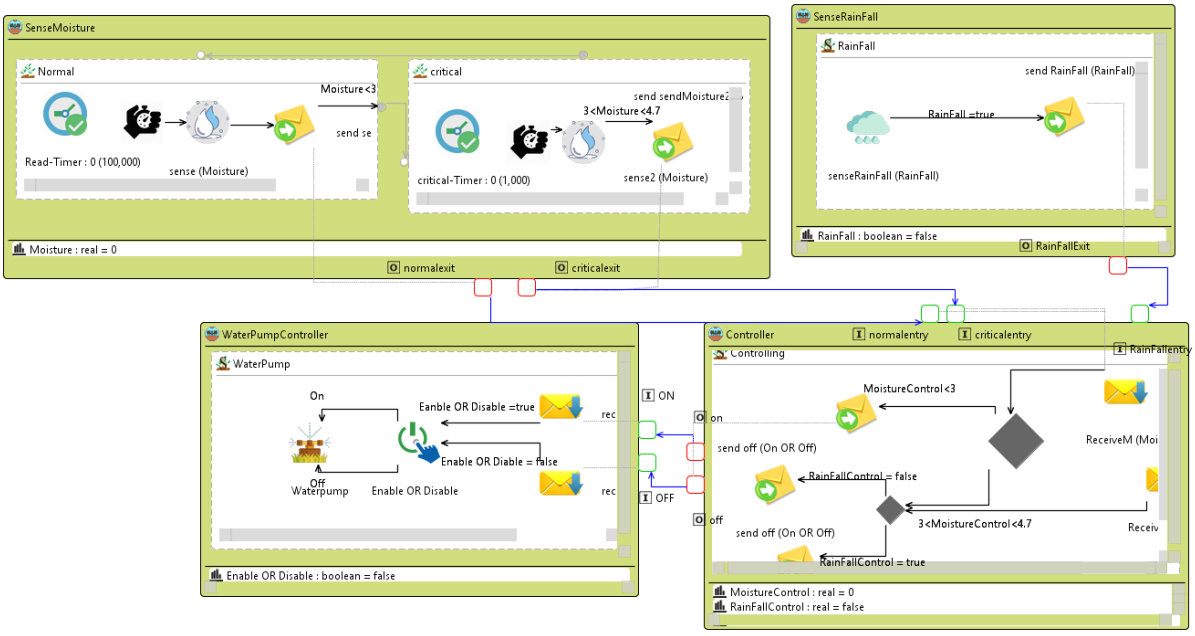


Figure 5.1: Humidity and Moisture Sensing.

- Humidity and Moisture Sensing

  Figure (5.1) shows the SAML model of the CSA-CAPS, simply representing a partial example of a Smart Agriculture case study. It is important to note that this figure is actually a screen shot of our real tool CSA-CAPS. As noted, Figure (5.1) is composed of four main components:

  1. The Sense-Moisture component is responsible for sensing the moisture value from the soil. It includes two modes: a. Normal: when the timer is on, the moisture sensor starts sensing the moisture value from the soil, then saves the value in moisture primitive variable. It then uses the unicast message to send the values to the controller component. The sensor keeps sensing every 100000 sec. b. Critical : every 1000 sec the sensor starts sensing the moisture value and saves it in primitive variable then uses the unicast message to send this value to the controller component if there is an abnormal value.

2. The Sense-Rainfall component is responsible for sensing the rainfall. It includes one mode: a. Rainfall: it contains an interrupt sensor which senses if there is a rainfall or not and keeps the value in a primitive variable. It uses a unicast message to send the result to the controller component.

3. The controller component is responsible for making decision to turn the water pump on or of. It includes one mode: a. Controlling: it receives the values of the moisture and rainfall in receive messages then stores the value in primitive variable to use it for making the decision depending on the condition. For example if the Moisture is more than 3, less than 4.7 and don't rain, the controller sends a message to the water pump to turn it on, but if Moisture is more than 3, less than 4.7 and Rainfall, the controller sends message to the water pump to turn it off. On the other hand, if the Moisture is less than 3 the controller sends message to the water pump to keep it off.

4. The water pump component is responsible for turning the pump on or off depending on the decision from the controller. It include one mode: a. Water Pump: this mode receives a message from the controller component then stores it in a primitive variable, the value stored in the primitive variable specify if the pump is turned on or not. This value will be send to an actuator, if the value equals true the actuator turn the pump on and if the value equals false the pump turns off.

- Gathering Results

  As shown in Figure (5.2), this case is about reading sensors and storing the readings in a server. This case study consists of seven components:

  The first six components (MoistureSensor, pHSensor, CalciumSensor, ChlorineSensor, HumiditySensor and PotassiumSensor) are the reading sensors

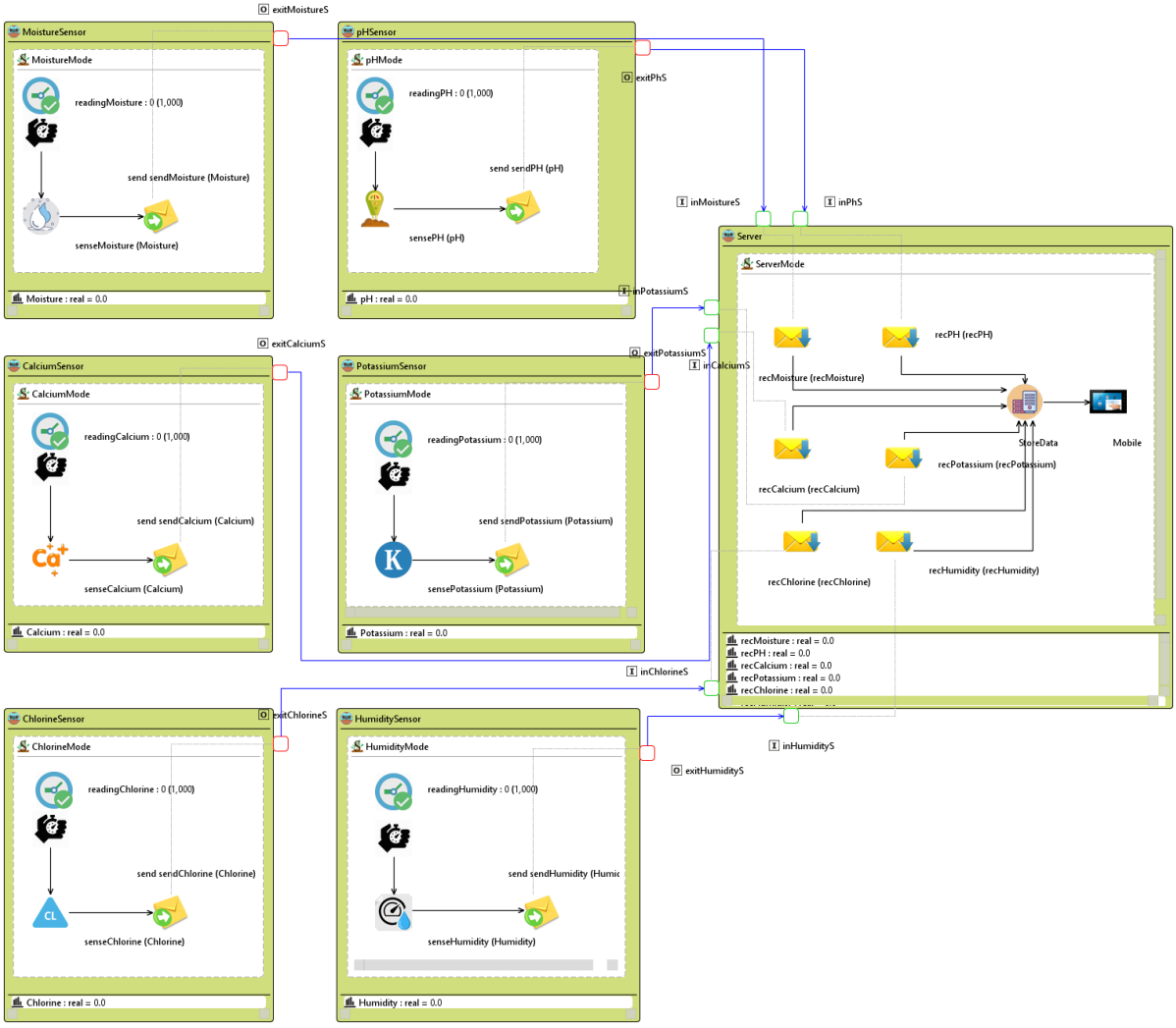


Figure 5.2: Gathering Results.

components and the seventh component is the Server component.

These sensors are for sensing the level of moisture, pH, calcium, chlorine, humidity and potassium levels in the soil. They start sensing every 1000 seconds, and send the sensed values to the server. Then the data can be accessed through a touchscreen mobile, this approach will help the farmer to detect the exact level of each measurement.

As shown in the figure, each sensor component contains an initial mode which consists of a start timer: the first behavioural element starts executing in the mode, and a sensor which is responsible for the sensing part. The sensed values are stored in primitive variables and these variables will go through the unicast send messages to be sent to the Server component.

The Server component is responsible for receiving the values using Receive Message and storing them in a server, which can be accessed using a touch screen.

- Detecting Fire

  The last case study shown in Figure (5.3) is about detecting Fire in Crops. Usually if a fire happened in the absence of Human beings, Crops will be damaged and disasters might happen. This model gives a case where a fire can be detected using CO, Temperature and Smoke sensors.

  This model is formed up of 6 components, as follows and shown in Figure (5.3):

  1. SenseCO: this component is formed of two modes, the first one will keep sensing as it's the Normal mode and the CO values are within acceptable range. This range would be less than 1200. The other critical mode will sense for 1000 seconds in order to see if the value is

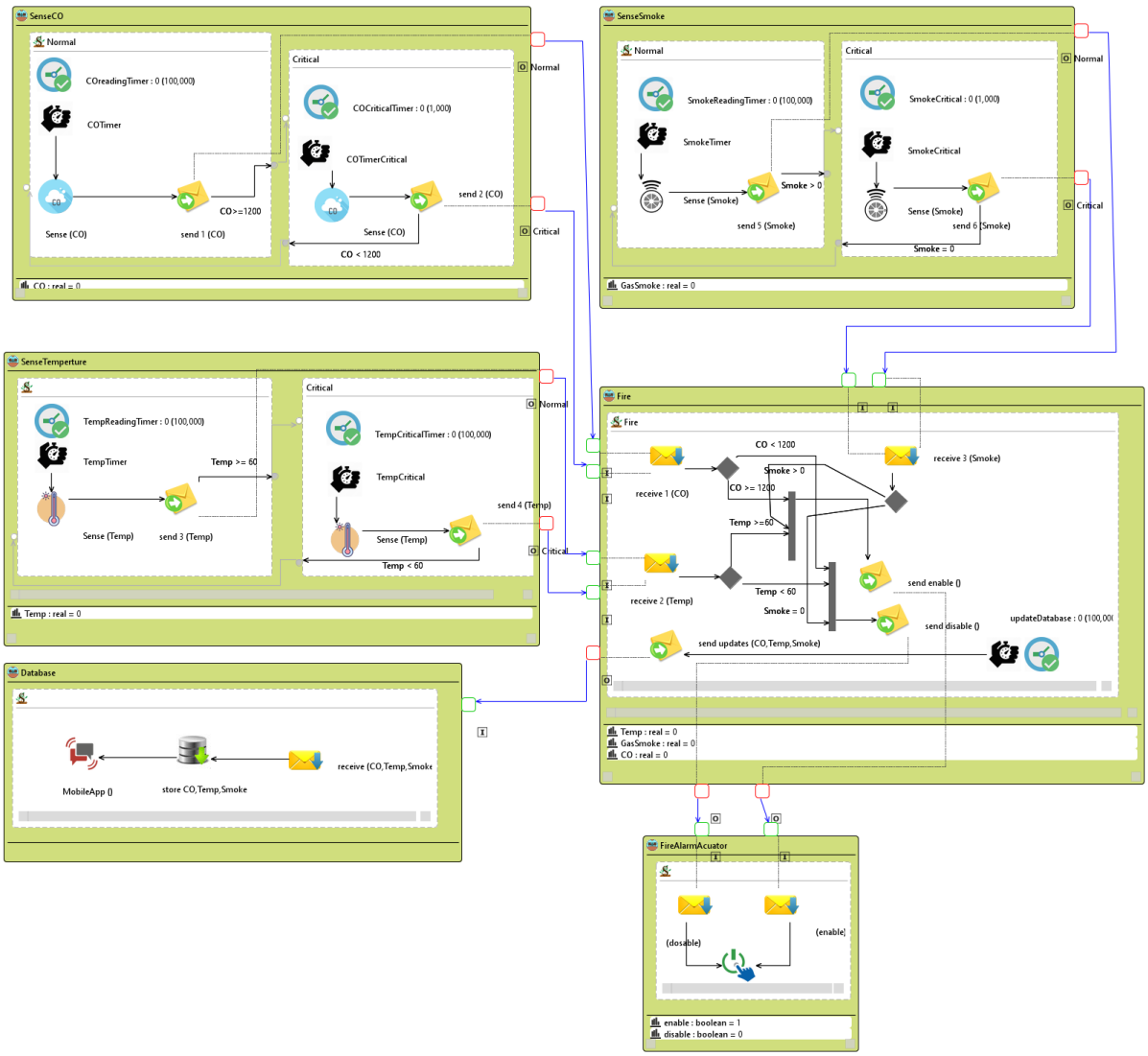


Figure 5.3: Detecting Fire.

still larger than 1200. In both cases the values will be sent to the Fire component.

2. SenseTemp: this component will sense the Temperature and send the value to the Fire component. Its inner scheme will work the same as in SenseCO component with the change of conditions. If the sensed temperature was higher than 60 degrees then it will treat it as critical situation.

3. SenseSmoke: this component will detect if there is smoke in the air and send the value to the Fire component. Its inner scheme will work the same as in SenseCO component with the change of conditions since any detected smoke would give an indication that there is Fire.

4. Fire: this component will receive CO, Temperature and smoke values, compare them to critical values in order to see if there is a fire or not. If the values show that there is a possible fire, a message will be sent to another component called FireAlarmActuator.

   This component also will send all the values of the sensors to the database in order to store them for later analysis.

5. FireAlarmActuator: this component would receive a message from Fire component informing it if there is a fire or not. This component will turn on the Fire alarm based on the sensors values.

6. DataBase: this component would receive a message from the Fire component including the sensed values in order to store them in the database for any further analysis. This data can also be accessible by a mobile application.

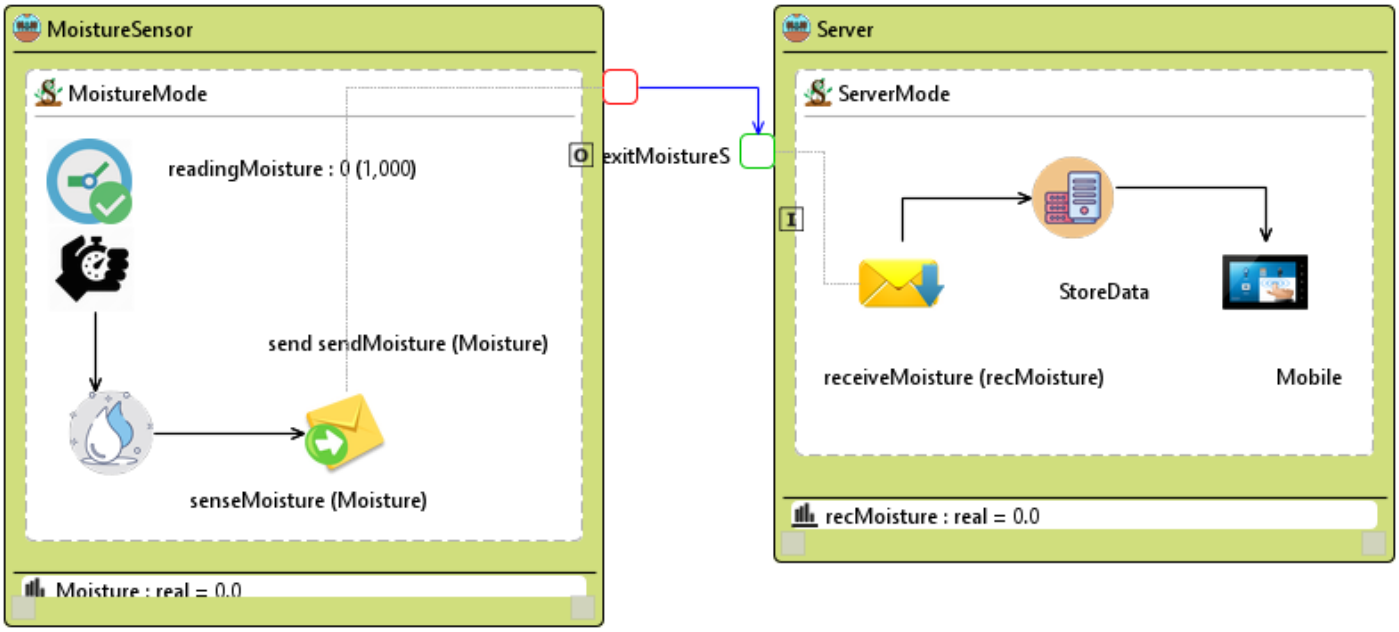


Figure 5.4: Test Model to convert

5.1.3 ThingML output using Acceleo

We have designed a small SAML model to convert it into ThingML so it could be shown here easily, since our models are very large and their conversion will be much more than can be carried in these pages.

The model is shown in Figure (5.4), it simply represents a Moisture Sensing System. This model is composed of two main components: 1. The Moisture Sensor component is responsible for sensing the moisture value from the soil. It includes one mode: Moisture Mode: when the timer is on, the moisture sensor starts sensing the moisture value from the soil, then it saves the value in a moisture primitive variable. It then uses the unicast message to send the values to the server component. The sensor starts sensing every 1000 sec. 2. Server: This component is responsible for storing the moisture values in a server after receiving them from the moisture component. These values can also be accessed using a

```
import "datatypes.thingml" from stl

thing fragment Messages {
    message Moisture (value : Float)
    message recMoisture (value : Float)
}

thing MoistureSensor includes Messages {

    // ***** Variables *****
    property Moisture : Float

    // ***** Behavioural Elements (Functions) *****
    function readingMoisture(val : Float) do
        // Do something
    end
    function senseMoisture(val : Float) do
        // Do something
    end

    // ***** Ports *****
    provided port exitMoistureS {
        sends Moisture
        receives recMoisture
    }

    // ***** States *****
    statechart init MoistureMode {
        state MoistureMode {
            on entry do
                readingMoisture(1000)
                senseMoisture(0.0)
            end

            transition -> MoistureMode event e: exitMoistureS?
                recMoisture action exitMoistureS!Moisture(Moisture)
        }
    }
}

thing Server includes Messages {

    // ***** Variables *****
    property recMoisture : Float

    // ***** Behavioural Elements (Functions) *****
    function StoreData() do
        // Do something
    end
    function Mobile() do
        // Do something
    end

    // ***** Ports *****
    required port inMoistureS {
        receives recMoisture
    }

    // ***** States *****
    statechart init ServerMode {
        state ServerMode {
            on entry do
                StoreData()
                Mobile()
            end

        }
    }
}

// ***** Configurations *****
configuration result {
    instance MoistureSensor: MoistureSensor
    instance Server: Server

    connector Server.inMoistureS => MoistureSensor.exitMoistureS
}
```

Figure 5.5: Converted model in ThingML

mobile application.

Figure (5.5) represents the mapping and converting result between the model shown in Figure(5.4) and the ThingML using our transformation tool (Acceleo code generator).

The first part represents the thing messages that includes all primitive variables which are stored in the unicast messages that will be used in the ports and transition part.

The second part represents:

1.The thing Moisture Sensor that includes two functions: the first function

reading moisture represents the timer in the CSA-CAPS model. The second function sense moisture that represents the moisture sensor in CSA-CAPS model. 2.The Moisture Sensor thing includes the provided port which is used to send the message to the other thing.

Each thing includes the 'statechart init' that has the name of the initial mode in each component. It points to the first state that will be executed in every thing. The statechart includes a state that represents the Mode in the SAML and it includes the on entry which is responsible for calling the functions sequentially. The statechart also includes the transitions which represent the links between the element in the same component in the CSA-CAPS.

The last part represents the thing server which follows the same structure of the moisture sensor thing.

5.1.4 Generating Languages from ThingML

ThingML code generation framework defines a framework of compilers able to transform a ThingML model into fully operational code in various languages.

We have stabilized four different languages: Java, Posix C, Arduino C and Node.JS in order to generate the output ThingML model into the different languages.

1. Java Language, in Figure (5.6).
2. Posix C Language, in Figure (5.7).
3. Arduino C Language, in Figure (5.8).
4. Node.JS Language, in Figure (5.9).

```java
import no.sintef.jasm.*;
public class Main {
//Things
    public static MoistureSensor MoistureSensor_MoistureSensor;
    public static Server Server_Server;

    public static void main(String args[]) {
//Things
        MoistureSensor_MoistureSensor = (MoistureSensor) new MoistureSensor();
        MoistureSensor_MoistureSensor.buildBehavior(null, null);
        MoistureSensor_MoistureSensor.init();
        Server_Server = (Server) new Server();
        Server_Server.buildBehavior(null, null);
        Server_Server.init();
//Connecting internal ports...
//Connectors
        MoistureSensor_MoistureSensor.getExitMoistureS_port()
            .addListener(Server_Server.getInMoistureS_port());
        MoistureSensor_MoistureSensor.initMoistureSensor_Moisture_var((float) 0.0f);
        Server_Server.initServer_recMoisture_var((float) 0.0f);
//Init instances (queues, etc)
//Network components for external connectors
        /* $NETWORK$ */
//External Connectors
        /* $EXT CONNECTORS$ */
        /* $START$ */
        MoistureSensor_MoistureSensor.start();
        Server_Server.start();
//Hook to stop instances following client/server dependencies (clients firsts)
        Runtime.getRuntime().addShutdownHook(new Thread() {
            public void run() {
                Server_Server.stop();
                MoistureSensor_MoistureSensor.stop();
                /* $STOP$ */
            }
        });
    }
}
```

Figure 5.6: Generated code in Java

```c
#include <unistd.h>
#include <stdlib.h>
#include <stdio.h>
#include <ctype.h>
#include <string.h>
#include <math.h>
#include <signal.h>
#include <pthread.h>

#include "runtime.h"


#define MAX_INSTANCES 2
#define FIFO_SIZE 32768

/*******************************
 * Instance IDs and lookup
 *******************************/

void * instances[MAX_INSTANCES];
uint16_t instances_count = 0;

void * instance_by_id(uint16_t id) {
  return instances[id];
}

uint16_t add_instance(void * instance_struct) {
  instances[instances_count] = instance_struct;
  return instances_count++;
}

/*****************************************
 * Simple byte FIFO implementation
 *****************************************/

byte fifo[FIFO_SIZE];
int fifo_head = 0;
int fifo_tail = 0;

// Returns the number of byte currently in the fifo
int fifo_byte_length() {
  if (fifo_tail >= fifo_head)
    return fifo_tail - fifo_head;
  return fifo_tail + FIFO_SIZE - fifo_head;
}

// Returns the number of bytes currently available in the fifo
int fifo_byte_available() {
  return FIFO_SIZE - 1 - fifo_byte_length();
}

// Returns true if the fifo is empty
int fifo_empty() {
  return fifo_head == fifo_tail;
}

// Return true if the fifo is full
int fifo_full() {
  return fifo_head == ((fifo_tail + 1) % FIFO_SIZE);
}

// Enqueue 1 byte in the fifo if there is space
// returns 1 for sucess and 0 if the fifo was full
int fifo_enqueue(byte b) {
  int new_tail = (fifo_tail + 1) % FIFO_SIZE;
  if (new_tail == fifo_head) return 0; // the fifo is full
  fifo[fifo_tail] = b;
  fifo tail = new tail;
  return 1;
}

// Enqueue 1 byte in the fifo without checking for available space
// The caller should have checked that there is enough space
int _fifo_enqueue(byte b) {
  fifo[fifo_tail] = b;
  fifo_tail = (fifo_tail + 1) % FIFO_SIZE;
  return 0; // Dummy added by steffend
}

// Dequeue 1 byte in the fifo.
// The caller should check that the fifo is not empty
byte fifo_dequeue() {
  if (!fifo_empty()) {
    byte result = fifo[fifo_head];
    fifo_head = (fifo_head + 1) % FIFO_SIZE;
    return result;
  }
  return 0;
}

/*****************************************
 * Synchronization for thread safe access
 *****************************************/

pthread_mutex_t fifo_mut;
pthread_cond_t fifo_cond;

void fifo_lock() {
  pthread_mutex_lock (&fifo_mut);
}
void fifo_unlock() {
  pthread_mutex_unlock (&fifo_mut);
}
void fifo_wait() {
  pthread_cond_wait (&fifo_cond, &fifo_mut);
}
void fifo_unlock_and_notify() {
  pthread_mutex_unlock (&fifo_mut);
  pthread_cond_signal (&fifo_cond);
}


/*****************************************
 * Initialization
 *****************************************/

void init_runtime() {
  pthread_mutex_init (&fifo_mut, NULL);
  pthread_cond_init (&fifo_cond, NULL);
}

```

Figure 5.7: Generated code in Posix C

```
/*****************************************************************
 * Implementation for type : Server
 *****************************************************************

// Declaration of prototypes:
//Prototypes: State Machine
void Server_OnExit(int state, struct Server_Instance *_instance);
//Prototypes: Message Sending
//Prototypes: Function
void f_Server_StoreData(struct Server_Instance *_instance);
void f_Server_Mobile(struct Server_Instance *_instance);
// Declaration of functions:
// Definition of function StoreData
void f_Server_StoreData(struct Server_Instance *_instance) {
}
// Definition of function Mobile
void f_Server_Mobile(struct Server_Instance *_instance) {
}

// Sessions functionss:

// On Entry Actions:
void Server_OnEntry(int state, struct Server_Instance *_instance) {
switch(state) {
case SERVER_STATE:{
_instance->Server_State = SERVER_NULL_SERVERMODE_STATE;
Server_OnEntry(_instance->Server_State, _instance);
break;
}
case SERVER_NULL_SERVERMODE_STATE:{
f_Server_StoreData(_instance);
f_Server_Mobile(_instance);
break;
}
default: break;
}
}

// On Exit Actions:
void Server_OnExit(int state, struct Server_Instance *_instance) {
switch(state) {
case SERVER_STATE:{
Server_OnExit(_instance->Server_State, _instance);
break;}
case SERVER_NULL_SERVERMODE_STATE:{
break;}
default: break;
}
}
```

Figure 5.8: Generated code in Arduino C

```
'use strict';

const Enum = require('./enums');
const Server = require('./Server');
const MoistureSensor = require('./MoistureSensor');
/*$REQUIRE_PLUGINS$*/


const inst_Server = new Server('Server', null);
inst_Server.initServer_recMoisture_var(0.0);
const inst_MoistureSensor = new MoistureSensor('MoistureSensor', null);
inst_MoistureSensor.initMoistureSensor_Moisture_var(0.0);
/*$PLUGINS$*/
/*Connecting internal ports...*/
/*Connecting ports...*/
/*$PLUGINS_CONNECTORS$*/
inst_MoistureSensor._init();
inst_Server._init();
/*$PLUGINS_END$*/

function terminate() {
    inst_Server._stop();
    inst_Server._delete();
    inst_MoistureSensor._stop();
    inst_MoistureSensor._delete();
};
/*terminate all things on SIGINT (e.g. CTRL+C)*/
if (process && process.on) {
    process.on('SIGINT', function() {
        terminate();
        /*$STOP_PLUGINS$*/
        setTimeout(() => {
            process.exit();
        }, 1000);
    });
}
```

Figure 5.9: Generated code in Node.JS

Chapter 6

Conclusions

In summary, CAPS provides a user-friendly geographical interface for modeling IOT systems and ThingML provides an automatic code generation framework from models. Therefore, we have used Acceleo in order to provide a mechanism to convert the SAML model into ThingML text model, so it can be compiled after that into several different languages.

References

[1] H. Muccini and M. Sharaf, "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. 4, 5, 6

[2] N. Harrand, F. Fleurey, B. Morin, and K. E. Husa, "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. 7, 8, 9

[3] E. Coronado, J. Villalobos, B. Bruno, and F. Mastrogiovanni, "Gesture-based Robot Control: Design Challenges and Evaluation with Humans," 05 2017.

[4] M. Sharaf, M. Abughazala, H. Muccini, and M. Abusair, "An architecture framework for modelling and simulation of situational-aware cyber-physical systems," in *European Conference on Software Architecture*, pp. 95–111, Springer, 2017.