Implementing the MDETools'18 challenge with ThingML*

Jakob Høgenes and Brice Morin

SINTEF Digital, Oslo, Norway
first.last@sintef.no

Abstract. This paper presents a ThingML implementation of the MDETools'18 challenge. ThingML is a textual modeling language that implements a sub-set of the UML (components and state-machines) and complements it with a first-class action language. ThingML also comes with a set of compilers targeting a large variety of platforms and programming languages. Using ThingML, we have been able to 1) fully model the MDETools'18 challenge, 2) automatically compile this specification it to two distinct programming languages, and 3) successfully execute the resulting programs and achieve reasonable results in the simulation.

Keywords: Model-Driven Engineering, Tool, Reactive Systems

1 Introduction

This paper presents a ThingML [1–3] implementation of the MDETools'18 challenge. ThingML is a textual modeling language that implements a sub-set of the UML (components and state-machines) and complements it with a first-class action language. The MDETools'18 challenge being to implement a control system for a follower rover interacting with a simulated leader rover, we believe state-machines communicating through asynchronous message passing is an adequate paradigm, and ThingML a relevant MDE tool supporting this paradigm. ThingML employs a textual syntax based on XText, fully integrated in the Eclipse IDE, providing an advanced editor with syntax highlighting, error reporting and code completion. Code for multiple target platforms can easily be generated using the standard Eclipse build interface. A key advantage of ThingML is its first-class action language, which allows modeling fine-grained behavior (*e.g.* actions in states and transitions) in a platform-independent way. Still, ThingML also provides a "kick-down" mechanism for mixing ThingML code with platform-specific code in order to leverage existing libraries and frameworks to avoid over-modeling by forcing designer to remodel those libraries. The complete implementation of the challenge is publicly available on GitHub[1].

* The research leading to these results has received funding from the European Commission's H2020 programme under grant agreement no 780351 (ENACT).

[1] https://github.com/SINTEF-9012/mdetools18-challenge-thingml

The remainder of this paper is organized as follows. Section 2 details how the challenge is modeled in ThingML. Section 3 presents how the platform-independent specifications from Section 2 can be specialized and compiled to different platforms. Section 4 concludes and discusses the strengths and weaknesses of ThingML when implementing this challenge.

2 Modeling the MDETools'18 Challenge with ThingML

The simulated environment of the posed challenge exposes two TCP ports for the external application (challenge solution) to 1) poll for the current state of the environment, and 2) send control signals to the follower rover. Not depicted in the figure, is a file containing parameters which describe both how the leader rover moves, the target distance for the follower rover, and the endpoints of the TCP ports.

The proposed solution to the problem, is an application consisting of: a) A **settings parser**, b) **Communication** with the simulator, c) **State estimation**, d) A **control system**, and e) A **timer**. Figure 1 shows how these parts are connected and interact with the simulated environment. A more detailed view into the actual components composing the application is given in Figure 3. The following sections will detail each part.

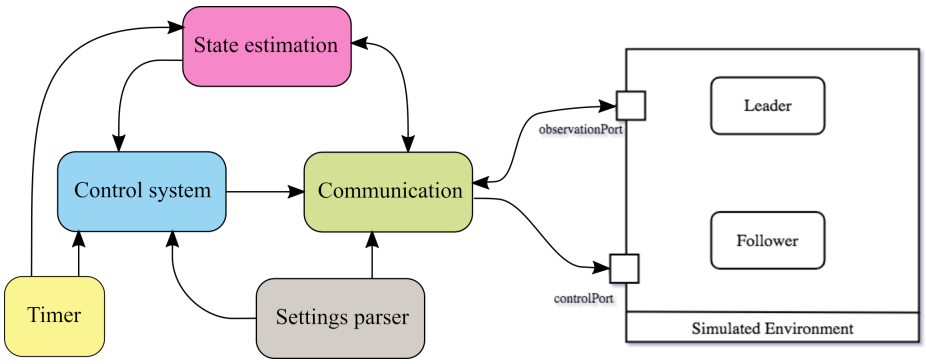

Fig. 1. Conceptual architecture of the proposed solution application. The direction of the arrows indicate the direction of flow of messages.

2.1 Settings parser

The settings parser is implemented with a single component. It reads the line-formatted file of parameters, parses them into suitable ThingML datatypes and when the whole file is read, sends a message out to the listening components containing the parsed parameters. In the proposed solution, the following parameters are used:

– `simulationIP`, `controlPort`, `observationPort`: for the communication part that needs to open TCP sockets to the simulator.
– `minDistance`, `maxDistance`: for the control system to calculate the target distance.
– `runTime`: for the control system to know when it is done and apply the brakes.

2.2 Communication

This part of the application is implemented with three components. There are two instances of a generic line-based TCP socket that 1) forwards ThingML messages containing a single string to the simulator as lines, and 2) reads lines coming from the simulator and sends them out as ThingML messages. The third component implements the communication protocol of the simulator, acting as the ThingML proxy to the simulator. It implements the parsing and formatting of strings and the ThingML messages shown in Listing 1.1, and uses two regions to handle the communication with both the leader and follower rovers concurrently, shown in Figure 2.

Listing 1.1. The communication API

```
thing fragment CommunicatorMsgs {
  message Ready()
  message SimulationStarted()

  message SetRoverLRPower(left : Integer, right : Integer)
  message SetBrake(brake : Integer)
  message GetRoverPosition()
  message GetRoverCompass()
  message RoverPosition(x : Double, y : Double)
  message RoverCompass(phi : Double)

  message LeaderStart()
  message GetLeaderPosition()
  message GetLeaderDistance()
  message LeaderPosition(x : Double, y : Double)
  message LeaderDistance(d : Double)
}
```

2.3 State estimation

The simulated environment only provides the position of the leader and follower, the heading of the follower, and the distance between the leader and follower rovers. To implement a decent control system, derivatives of these quantities are also needed. This part therefore contains two components that estimate the first and second order derivatives, using a finite difference scheme with 2^{nd} order accuracy, one for each of the rovers. At a fixed time interval each estimator requests the current position of its assigned rover. When the response arrives, the last three position values are used to estimate the derivatives.

A third component buffers the received state estimates from each of the rover estimators, and calculates values that are passed on to the control system:

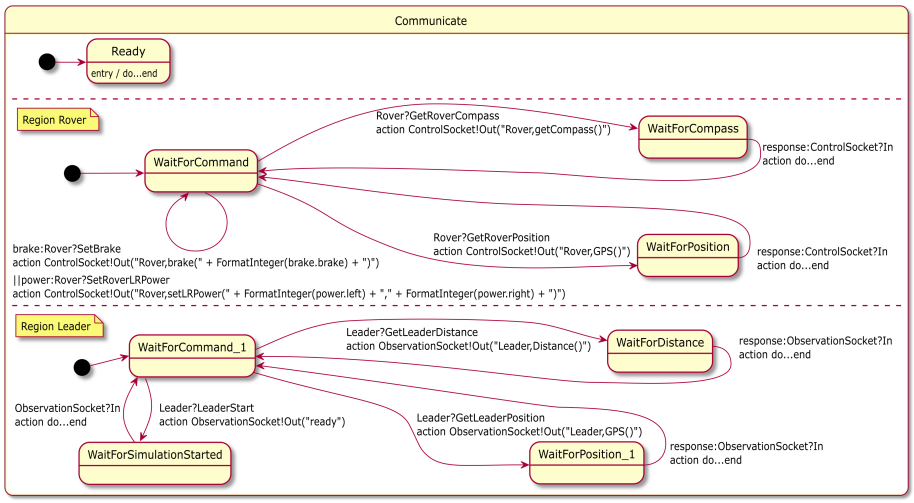

Fig. 2. Excerpt of the communicator thing statechart. Upon receiving the parsed settings from the settings parser, and opening observation and control sockets, the communicator enters the depicted composite state. The two parallel regions handle communications with the leader and follower rovers concurrently. As a command or request is received in the form of a ThingML message, it is formatted as a string and passed on to the appropriate socket. If a response is expected, an intermediate waiting state is entered until a string is received from the socket, which is then parsed and passed on as a ThingML message to listening components.

1. The angle from the follower to the leader, the heading.
2. The distance between the leader and follower.
3. The speed of the leader.
4. The speed of the rover, projected onto the speed of the leader. This is used to prevent the follower from going full in the wrong direction.

2.4 Control system

The control system consists of three components responsible for controlling different aspects of the system, and a fourth component for coordinating and composing the final inputs to the follower rover. The implemented control components are:

1. **Distance controller**. A P-controller that tries to keep the follower at a distance of (`minDistance` + `maxDistance`)/2 from the leader.
2. **Heading controller**. A P-controller that tries to point the follower directly towards the leader.
3. **Speed controller**. A PI-controller with anti-windup, that tries to match the projected speed of the follower with the speed of the leader.

These control systems receive state estimates from the estimation component, and buffer the last received values. At a configurable time interval the *control multiplexer* component sends a message to the first controller with $(0, 0)$ as the inputs for the left and right wheels of the follower. Each of the controllers add their own control input on top of the received input, and pass it on to the next controller, and finally back to the control multiplexer. When the control multiplexer gets the combined input from all the controllers back, it scales the values to make sure it is within the allowed range of $[-100, 100]$, and sends it to the communication component to be passed to the simulated environment.

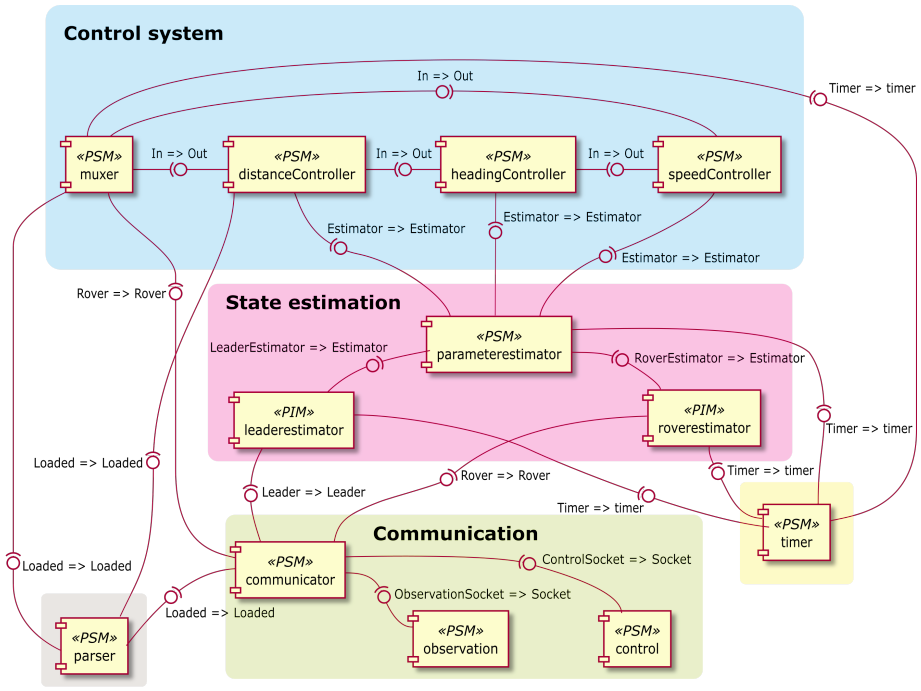

Fig. 3. Detailed component diagram generated by the PlantUML ThingML compiler. The colored boxes indicate the parts of the system shown in the conceptual architecture in Figure 1, and described in sections 2.1-2.5 . For each component, a corresponding state machine defines the component behavior. The lines depict connectors between ports of the components, which indicate the flow of messages in the system.

2.5 Timer

The timer is a single component that receives requests to start a countdown for a specific *ID*, and when the countdown has elapsed sends a notification to all listening components with the corresponding ID. Although the component is very simple, it is essential since the rest of system operates asynchronously without any notion of time.

3 Generating and Executing Code for Two Languages

ThingML comes with a rich platform-independent action language, and compilers targeting C (for microcontrollers and Linux), Java, JavaScript (Node.JS and Browser) and Go. This means that generating complete and executable code for multiple platforms, in many cases, comes at no extra cost to the developer. However, as very specialized functionality (like *ad hoc* communication protocols) are not included in the ThingML modeling language, some situations require platform specific models. There are three ways to accomplish this using ThingML:

1. Using plug-ins that come bundled with the ThingML compilers. These plugins are specialized for each target language, and generate code for exchanging ThingML messages across network protocols (such as MQTT and WebSockets) with other ThingML or legacy applications.
2. Using standard libraries that come bundled with ThingML. These libraries provide commonly used functions (*e.g.* mathematical functions or string formatting and parsing) as both platform independent abstract declarations, and platform specific concrete implementations.
3. Manually mix in platform specific code using the "kick-down" mechanism, or applying compiler specific options using annotations.

To demonstrate how platform independent and platform specific models are specified and mixed in ThingML, the proposed solution was implemented with models to generate code for both Java and Go. The final models consist of mainly platform independent components, while the following parts of the ThingML model is implemented in a platform specific way: a) Mathematical functions, b) String formatting and parsing, c) The timer component, d) Line-based messaging over TCP, and e) Reading the configuration file line-by-line.

The following sections first demonstrate how platform independent and specific models may be combined by showing the ThingML implementation of the configuration file parser, then compare the implementation of the proposed solution using the two languages.

3.1 Example of Platform Specific Models Using ThingML

The settings parser component is responsible for converting a specially formatted file of parameters to ThingML datatypes, and notifying listening components about their values. Reading files and parsing strings are not functionalities that is supported in the ThingML action language directly, so this component requires platform specific implementations using the mechanisms provided by ThingML.

Listing 1.2 shows the platform independent partial (indicated by *fragment*) implementation of the component. This model defines the relatively simple behavior of the component. Once started, the `ReadConfigFile` function is called, which again is supposed to call `HandleLine` for each line in the file, and after all lines are read, the parsed parameters are sent out over the `Loaded` port.

The model uses two ways of defining platform independent behavior without the platform specific implementation. The `ReadConfigFile` function is

declared as an abstract function, leaving all the details up to the final implementation of the component. The `HandleLine` function calls abstract functions declared in the platform independent `String` component from the ThingML Strings library, shown in Listing 1.4. To implement these functions, one may simply include the appropriate `*Strings` component from the library.

Listing 1.2. Platform independent partial implementation of the settings parser component. Some details have been left out to save space (...).

```
thing fragment SettingsParser includes ..., Strings {
  provided port Loaded { sends Settings }
    property ConfigPath : String = "Challenge problem/Settings/config.txt"
  property controlPort : Long = 0
  ...
  function HandleLine(line : String) do
    var option : String[2]
    SplitInto(line, "=", option)
    if (Equals(option[0], "controlPort")) do
      controlPort = ParseLong(option[1])
      return
    end
    ...
  end

  abstract function ReadConfigFile(path : String)

  statechart Loader init LoadConfiguration {
    state LoadConfiguration {
      on entry do
        ReadConfigFile(ConfigPath)
        Loaded!Settings(controlPort, ...)
      end
    }
  }
}
```

The final implementation of the settings parser component for Java is shown in Listing 1.3. It includes the `JavaStrings` component from the Strings library that implements the abstract functions for Java (wrapping methods from the Java String class in a straightforward manner), while the `ReadConfigFile` is manually implemented with Java-code using the "kick-down" mechanism. The `java_import` annotation instructs the ThingML Java compiler to include these additional imports at the top of the generated Java source files.

Listing 1.3. Platform specific implementation of the settings parser for Java.

```
thing JavaSettingsParser includes SettingsParser, JavaStrings
@java_import `
  import java.io.FileReader;
  import java.io.BufferedReader;
  import java.io.IOException;`
{
  function ReadConfigFile(path : String) do `
    try (BufferedReader br = new BufferedReader(new FileReader(`&path&`))) {
      for (String line = br.readLine(); line != null; line = br.readLine()) {
        ` HandleLine(`line` as String) `
      }
    } catch (IOException e) {
```

```
13        System.err.println("Couldn't read config file: "+`&path&`);
14        System.exit(1);
15      }
16    ` end
17  }
```

Listing 1.4. ThingML Strings library, showing both the platform independent declaration and the platform specific implementation for Java. Some details have been left out to save space (...).

```
1  thing fragment Strings {
2    abstract function ParseLong(s : String) : Long
3    ...
4  }
5
6  thing fragment JavaStrings {
7    function ParseLong(s : String) : Long do
8      return (`Integer.parseInt(`&s&`)`) as Long
9    end
10   ...
11 }
```

3.2 Evaluation of Controllers Generated Using Two Languages

To evaluate the performance of the proposed solution, and to ensure that the generated code for both Java and Go perform equally well and consistently, an experiment was conducted. The experiment consists of five *scenarios* corresponding to five randomly generated leader rover paths. For each scenario, both the generated Java and Go controllers were executed during five runs alongside the simulated environment. The resulting power input to the follower rover and the distance between the leader and follower rover were recorded over time, as well as the final percentage of time *in the zone*.

Table 1 shows the final percentage of time *in the zone* for all runs of the experiment. Overall, the performance of the two languages are similar for each scenario, and quite consistent across the runs within each scenario. For a couple of runs, the Go controller performed significantly worse than the other runs of the same scenario (highlighted with gray cells). The reason for this is still unclear, but since it does not seem to happen for the Java, it might be a manifestation of a bug in the ThingML Go compiler, which warrants further investigation.

Figure 4 provides a more detailed view of the execution of the controllers for both Java and Go. It shows the power input to the follower rover and the distance between leader and follower rover over time for the two controllers, for randomly selected runs in a randomly selected scenario. In general, the two controllers produce quite similar input, and perform equally well. The slight differences are likely caused by timing differences (since the setup is not a hard real-time system), and potential floating point number inaccuracies. The jaggedness of the power input is probably a result of the sampling of position of the rovers from the simulated environment, either that the sampling rate is too high, the sampling time being non-uniform, or the low-order finite differentiation scheme implemented. As the controllers performed satisfactory, this issue has not been further investigated.

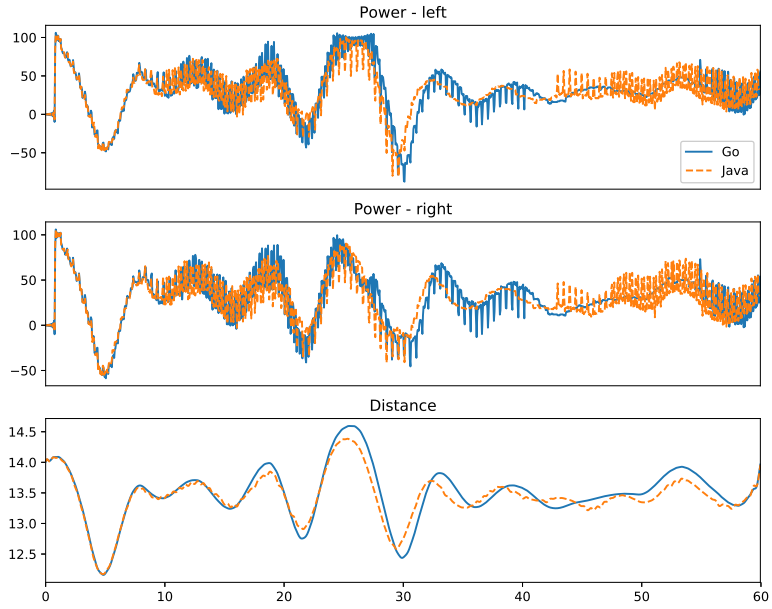

Fig. 4. Comparison of the power inputs for the Go and Java controllers, and the distance between leader and follower rover, for a a single execution.

Table 1. Resulting percentage of time *in the zone* for multiple runs of the simulation with both Go and Java controllers. Outliers are highlighted with gray cells.

Scenario:	Language:	Run 1:	Run 2:	Run 3:	Run 4:	Run 5:
Scenario 1	Go	100%	100%	100%	100%	100%
	Java	100%	100%	100%	100%	100%
Scenario 2	Go	69.2%	67.5%	70.0%	68.8%	68.8%
	Java	70.0%	70.0%	70.0%	70.4%	73.3%
Scenario 3	Go	97.5%	98.3%	29.6%	97.5%	97.5%
	Java	97.5%	97.5%	97.5%	97.0%	97.0%
Scenario 4	Go	100%	70.3%	100%	100%	97.9%
	Java	95.8%	95.0%	96.3%	100%	100%
Scenario 5	Go	100%	100%	100%	100%	100%
	Java	100%	100%	100%	100%	100%

4 Conclusion

In this work, we have proposed a solution to the MDETools'18 Challenge problem using the ThingML modeling language and accompanying tools. Leveraging the ThingML language and tools, we were able to generate executable code in two languages (Java and Go) that in our experiments produced similar and reasonably good results.

The resulting ThingML models for both languages consist of 767 lines of platform-independent code, and 71 (8%) and 88 (10%) lines of platform specific code for Java and Go respectively. In addition, 157 lines of code for new libraries has been written for the two languages that is to be contributed to the ThingML standard libraries.

This small amount of platform specific code, substantiates the claim that the ThingML modeling language is useful in practical applications, such as the posed challenge. Coupled with the wide range of platforms targeted by the ThingML compilers, it also means that generating code for suitable targets comes at little extra cost to the developer.

In terms of the control system implemented in the proposed solution of this work, the performance is far from optimal. However, as ThingML offers no specific support for control systems, the authors consider this as outside the main scope of this challenge, and leave further improvements for future work. Such improvements (such as limiting the rate of change of applied power to void wheel-spin) could with ease be implemented with platform independent models. Extending the proposed solution to other languages (*e.g.* C and JavaScript) is also reserved for future work.

During the work with solving the posed challenge, some bugs and shortcomings of the ThingML modeling language and tools was discovered, despite the already strong test suite in place for the compilers. Most have already been fixed and contributed to the official ThingML repository, while some (particularly related to the inconsistent behavior of the Go controller) requires further investigation, and will definitely be dealt with by the ThingML developers. As such, the posed challenge has both been a good exercise to assure the practicality of ThingML, as well as providing substantial input for the future development.

References

1. Franck Fleurey, Brice Morin, Arnor Solberg, and Olivier Barais. MDE to manage communications with and between resource-constrained systems. In *International Conference on Model Driven Engineering Languages and Systems*, pages 349–363. Springer, 2011.
2. Nicolas Harrand, Franck Fleurey, Brice Morin, and Knut Eilif 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*, pages 125–135. ACM, 2016.
3. Brice Morin, Nicolas Harrand, and Franck Fleurey. Model-based Software Engineering to tame the IoT Jungle. *IEEE Software*, 34(1):30–36, 2017.