An evaluation of Papyrus-RT for solving the leader-follower challenge problem

Karim Jahed

Queen's University, Kingston, ON
jahed@cs.queensu.ca

Abstract. We discuss and evaluate the use of Papyrus-RT modeling environment to solve the leader-follower challenge problem. We present a Papyrus-RT solution to the problem and highlight the tool's strengths and weaknesses in that context. Furthermore, we show how some recent third-party extensions to the tool can help in mitigating some of its shortcomings.

Keywords: UML-RT · Papyrus-RT · MDD.

1 Introduction

The goal of the challenge problem[1] is to design and develop a rover controller using a Model-Driven Development (MDD) approach. The rover, dubbed *follower*, runs in a simulated environment and is tasked with following the *leader*, while always keeping a safe distance. The *leader* is a computer-controlled rover that follows a random path in its environment.

As shown in Figure 1, the simulation environment exposes two TCP ports to allow interaction with third-party applications. The `observationPort` allows an application to query the environment for information such as the rovers' position, orientation, and distance. Likewise, the `controlPort` allows external applications to control the *follower* by, for instance, applying power to its wheels.

In this work, we use Papyrus for Realtime (Papyrus-RT) to design and implement the *follower* controller. We highlight the strengths and weaknesses of the tool in regard to different aspects of the challenge problem. While our solution shows that Papyrus-RT's support for modeling composite systems, along with its powerful code generator, is adequate for addressing the requirements of the challenge problem; it's reliance on C++ as an action language and lack of intrinsic support for TCP communication proved cumbersome while developing our solution. However, these shortcomings has not gone unnoticed, and there is an ongoing effort by the community to remedy some of these challenges. We will discuss some of the effort that is directly related to the challenge problem and can help in improving our solution.

[1] Available at https://mdetools.github.io/mdetools18/challengeproblem.html

The rest of this paper is organized as follows. Section 2 lay out some background related to Papyrus-RT and UML-RT. In Section 3 we describe our solution to the challenge problem. In Section 4 we highlight the strengths and challenges faced during the development of our solution, and evaluate the performance of our solution in realizing the simulation goal. We conclude this work in Section 5.

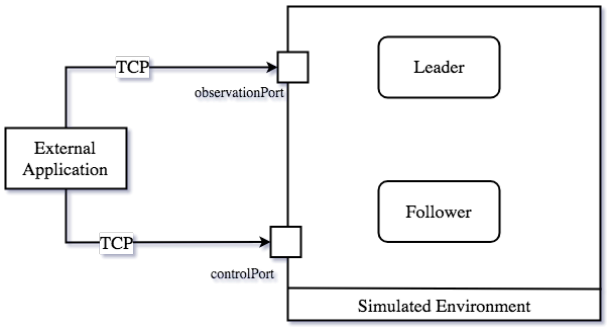

Fig. 1. Overview of the challenge problem (from [3])

2 Papyrus-RT

Papyrus-RT[2] is an open-source, industrial-grade modeling environment for the design and development of complex, real-time, embedded system. Papyrus-RT is based on the Papyrus modeling environment and has three main components: a modeling environment, a code generator, and a runtime system.

2.1 Modeling Environment

Papyrus-RT implements a graphical modeling environment that supports the structural and behavioural specification of software systems using the UML for Realtime (UML-RT) profile. UML-RT is a subset of the UML language tailored for realtime embedded systems.

UML-RT adopts a distributed message passing model based on the actor paradigm. Figure 2 shows a rough summary of the main concepts in UML-RT (see [5, 4] for a comprehensive discussion). *Capsules* are active, self-contained classes that encapsulate certain functionalities or represent a system component. A capsule can nest other capsules to describe a system's composition. Each capsule maintains an internal state and can receive messages through its *ports*. Each port is typed with a specific *protocol* that defines the set of messages that can be sent and/or received through that port. Unlike a network protocol, a

[2] Available at https://www.eclipse.org/papyrus-rt

protocol in UML-RT does not define 'how' messages are exchanged nor a specific message sequence. It simply define a set of input and output *signals*, as well as their formal parameters, that might be sent and received by any port typed with the protocol. A pair of ports typed with the same protocol can be linked using a connector which allows them to exchange the messages defined by the protocol. A port can be *conjugated* to reverse its role in the protocol. In a conjugated port, input and output signals are reversed, i.e., input signals becomes output signals and vice-versa. A signal sent by an un-conjugated port can only be received by a conjugated port using the same protocol.

A capsule's behaviour is specified using a statechart whose transitions are triggered exclusively by signals received from one of the capsule's ports. In order to send signals to other capsules, modify the capsule's internal attributes, and perform arbitrary calculations, snippets of action code (C++, in the case of Papyrus-RT) must be attached to transitions and/or states.

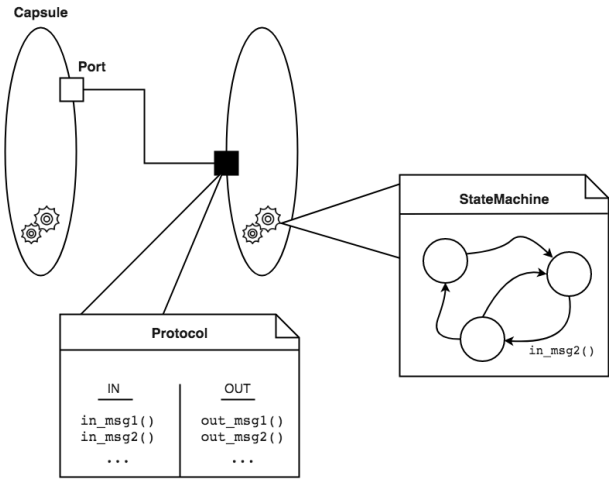

Fig. 2. The main concepts of UML-RT

2.2 Code Generator

Papyrus-RT implements a code generator capable of generating fully executable code from UML-RT model. The default code generator provided with Papyrus-RT generates C++ code packed into an Eclipse C/C++ Development Toolkit (CDT) project. Since Papyrus-RT is based on Eclipse, this allows the execution and debugging of the generated code right from within the modeling environment. While the default code generator is for C++, generators for other languages can be easily integrated via an extension point.

2.3 Runtime System

The Runtime System (RTS) is the glue that allows the model (generated code) to execute. The RTS is a set of libraries that provide implementations for the different UML-RT concepts such as capsules, ports, and protocols; as well as abstractions for OS-specific processes such as threading and semaphores. In addition, the RTS implements the mechanisms for message passing between capsules, and expose a set of services to facilitate modeling such as a timing and logging services.

3 Solution

In this section, we describe our solution[3][4] to the challenge problem using Papyrus-RT. We start by describing the structure of the system and its components followed by the behaviour of each of those component.

3.1 Composition

The main component of our system is the `Follower` capsule shown in Figure 3. The `Follower` implements a statechart to steer the *follower* rover in the direction of the *leader* while keeping a safe distance. The statechart's actions rely on information gathered from two sub-components: the `LeaderObserver` capsule and the `FollowerController` capsule. The `LeaderObserver` communicates with the `Follower` capsule through an internal port typed with the `ObservationProtocol` protocol. Similarly, the `FollowerController` capsule exchange messages with the `Follower` capsule via an internal port typed with the `ControlProtocol` protocol.

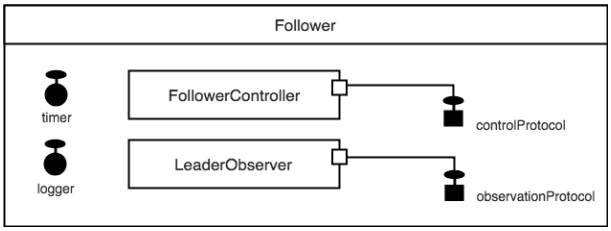

Fig. 3. Composition diagram for the `Follower` capsule

The main purpose of the `FollowerController` and `LeaderObserver` capsules is to hide the network communication detail from the `Follower` capsule.

[3] Available at https://github.com/kjahed/mdetools18-challenge

[4] While the challenge problem was devised in the same research group that the author is part of, the author did not participate in its design. Both, the challenge problem and the solution presented here, were developed completely independently.

Both of these capsules implements the necessary details to connect to the simulation environment over TCP, query it, and control it. Figure 4 shows the interface provided by both of these capsules through their port. Note that both of these capsules are nested within the main `Follower` capsule as sub-components. It is entirely possible to move these capsules to same hierarchy level as the Follower. However, out design choice is intentional, as both the `FollowerController` and `LeaderObserver` implements auxiliary services that are needed only by the `Follower`. This allows us to encapsulate the `Follower` and all of its dependency in one component that can be easily embedded into larger models.

In addition to the two internal ports used to communicate with the sub-capsules, the `Follower` capsule also has two additional internal ports that connect the capsule to services provided by the runtime system. The `timer` port allows the follower to set timers and schedule events, while the `logger` port is used to output messages to the console.

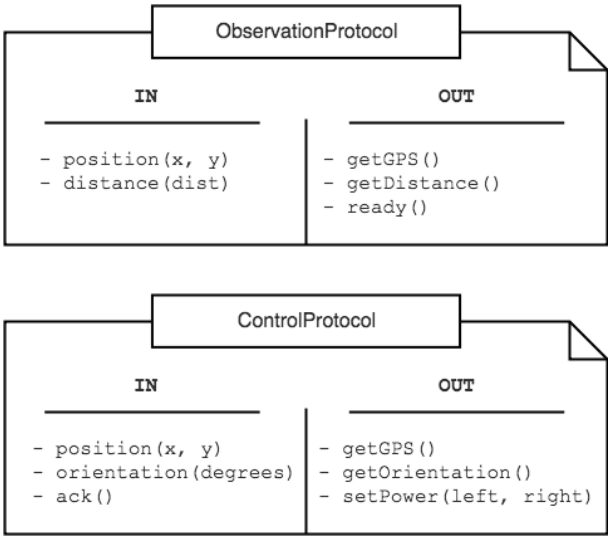

Fig. 4. The protocols used by the `Follower` to query/effect the simulation environment

3.2 Behaviour

Network communication. As mentioned earlier, the `FollowerController` and `LeaderObserver` capsules implements behaviour to facilitate communication with the simulation environment. Since the behaviour of both capsules is closely similar, we limit the discussion in this section to the `LeaderObserver` capsule.

The statechart for the `LeaderObserver` capsule is shown in Figure 5. The initial transition leading to the `WAITING` state executes a standard piece of C++

action code (not shown) to establish a TCP socket connection with the simulation environment. The statechart then transitions into the `WAITING` state and remains in that state until a request is received from the `Follower` capsule. A request is simply one of the output signals defined in the `ObservationProtocol` (Figure 4. These output signals are input signal to the `LeaderObserver` and used to trigger the transitions of its statechart. Whenever such signal is received, the `LeaderObserver` transitions into the appropriate state to process the `Follower`'s request. Upon entry to the new state, a message encoding the request, along with any parameters, is constructed and transmitted over the TCP socket to the simulation environment. The `LeaderObserver` remains in that state until a response message is received (from the simulation environment), and the appropriate response signal (an input signal defined in the `ObservationProtocol`) is sent back to the `Follower`. The `LeaderObserver` then returns to the `WAITING` and waits for the next signal.

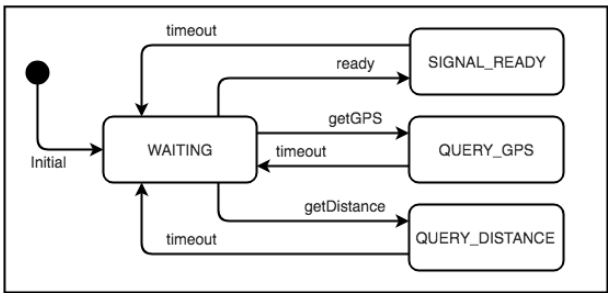

Fig. 5. Statechart for the `LeaderObserver` capsule

Follower. The `Follower`'s statechart (Figure 6) implements the main behaviour that steers the `follower` rover into the `leader`'s direction while keeping a predetermined safe distance. The behaviour is divided into two phases:

- In the first phase, the `Follower` waits until the *leader* rover starts moving. Upon entry to the `IDLE` state, the `Follower` sends a `getDistance()` signal to the `LeaderObserver` capsule. The `LeaderObserver` queries the simulation environment and responds with a `distance(dist)` signal to the `Follower`. This signal triggers the `Follower` to move into a choice state to check whether the received distance is greater than zero. In this case, the `Follower` transitions into the `START` state to begin tracking the *leader*. Otherwise, the `Follower` returns to the `IDLE` state and queries the distance again.
- The second phase starts when the `Follower` moves into the `START` state. The `START` state is a composite state containing itself a nested statechart.

Upon entry to the `START` state machine, the follower first collects all the necessary information from the simulation environment by querying the `LeaderObserver`

and `FollowerController` capsules. The information collected includes the current distance between the two rovers, their position in the environment, and the orientation of the *follower* in space.

Upon collection of all the information and exit from the `GET_MY_ORIENT` state, a piece of action code is executed to determine whether a rotation of the *follower* river is necessary to face the *leader*. The is done by taking the difference between the current orientation and the angle between their current positions. If the difference is larger than a certain threshold, the `Follower` moves into the `ROTATE` state. The `ROTATE` states rotates the *follower* by calculating the appropriate power to apply to each of its left and right wheels, and applying the power by sending a `setPower(leftPow, rightPow)` signal to the `FollowerController` capsule.

In case no rotation is necessary, the `Follower` moves into the `FOLLOW` state which applies the same amount of power to both wheels in order to move the rover in a straight line. To keep a safe distance from the *leader*, the power applied to the wheels is calculated proportionally to the current distance between the two rovers.

Finally, the `Follower` enters a `DELAY` state where a timer is set to repeat the entire process after a fixed amount of time.

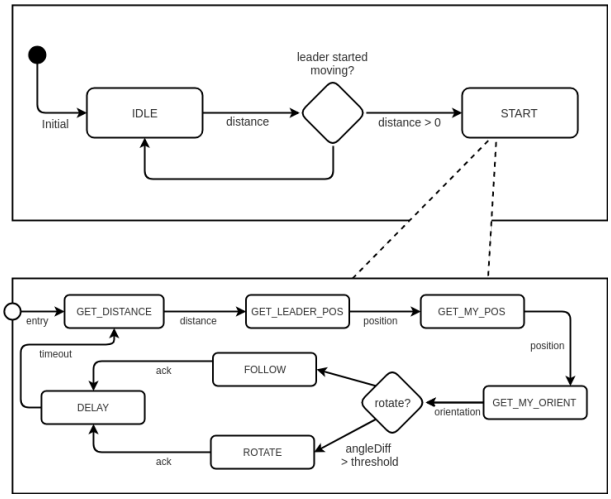

Fig. 6. Statechart for the `Follower` capsule

4 Evaluation

4.1 Solution evaluation

In order to evaluate the effectiveness of our control algorithm, we ran the simulation while varying the `straightPercent` variable of the simulation environment. This variables determines how straight the random path taken by the *leader* is. Obviously, the least straight the path is, the harder it is for the `follower` to stay within the required distance. Table 1 shows the results.

Table 1. Percent of time in safe zone with varying path 'straightness'

Straight %	20	40	60	80	100
Safezone %	66.48	72.49	76.25	88.58	97.16

4.2 Tool evaluation

UML-RT proved to be very suitable for modeling the challenge problem. The inherent actor-oriented nature of the language allowed us to isolate certain 'dirty' aspects of the solution, such as network communication, for the main computation. Message passing between the different components by means of well-defined protocols offers strong encapsulation of each component's internal state, and thus, simplifies the design.

Moreover, the powerful code generator allowed us to focus entirely on the models. At no point during the development process we resorted to editing or modifying the generated code. The automatic generation of an Eclipse CDT project allowed us to build and execute the generated code right from within the modeling environment, without having to switch workspaces.

The debugging facilities provided by the generated executable were very helpful in understanding the inner-working of the program. The Papyrus-RT Runtime System library, linked to the generated executable, allows us to activate several command-line flags to output debug messages at different levels of granularity. These debug messages enabled us to examine the full messaging sequence between the capsules.

The development process was not without challenges, however. The main culprit was the use of plain C++ as an action language. Unfortunately, Papyrus-RT does not yet offer any content-assist features when inputting the action code. Not even automatic indentation nor syntax highlighting. Essentially, the code generator takes the action code 'as-is' and places it in the appropriate location in the generated code. This means that syntax errors go unnoticed until the generated code is actually built. We also noticed that, in some very specific cases, the code generator might generate erroneous (uncompilable) source code. One example is when a protocol signal's name collides with a C++ reserved word such as `break`.

Calur [2] (Core Action Language for UML-RT) is an alternative action language for UML-RT that is currently under development. Calur attempts to remedy the aforementioned problems by providing a restricted language designed specifically for UML-RT, along with many 'smart' editing features integrated within Papyrus-RT. The language supports basic arithmetic operations, conditional statements, as well as UML-RT specific commands to, for example, send messages to other capsules and set timers. Its implementation within Papyrus-RT offers many traditional content-assist features such as syntax highlighting and auto-completion, in addition to the detection of certain UML-RT specific abnormalities in the action code and offering quick fixes. One example is the absence of an internal 'timing' or 'log' port in a capsule whose action code includes a log or timing statement.

A second challenge we experienced was the disconnection between the model and the running executable when it comes to debugging. While we could examine the sequence of messages exchanged between capsules at runtime, a manual effort was required to associate this information with model-level elements, and additional debugging statements where needed to examine the internal state of each capsule. MDebugger [1] attempts to remedy this problem by offering a model-level, platform-independent debugger for UML-RT models.

Finally, a third challenge we faced was the absence of intrinsic TCP communication support in Papyrus-RT. Essentially, all TCP related operations such as establishing connections with the simulation environment and sending and receiving messages were entirely specified in the action code using C++. This, of course, limits the expressiveness of the statechart. Moreover, the only way to trigger a statechart's transitions in Papyrus-RT is using a signal received from one of the capsule's ports. Therefore, in Figure 5 for example, the TCP response message received from the environment cannot be used to move back to the `WAITING` state. Since a trigger signal is necessary, whenever a state (e.g., QUERY_GPS) finishes processing a request, it sets a dummy timer in the action code to trigger the transition back to the `WAITING` state.

Our previous work on enabling Papyrus-RT for the development of Internet-of-Things applications remedies this problem by adding support for TCP communication to the Runtime System (RTS), and exposing a simplified interface to connect, send, and receive TCP messages using a special internal port, much like a timer port. This extension[5] enables capsules to communicate with external TCP devices by simply sending a UML-RT protocol signals to the RTS, and allows incoming TCP messages to trigger statechart transitions. Figure 7 shows how the `QUERY_GPS` state of the `LeaderObserver` capsule can be turned into a composite state to handle the TCP request using an internal port typed with a `TCPProtocol` defined by the RTS.

[5] Available at https://github.com/kjahed/papyrusrt-tcp

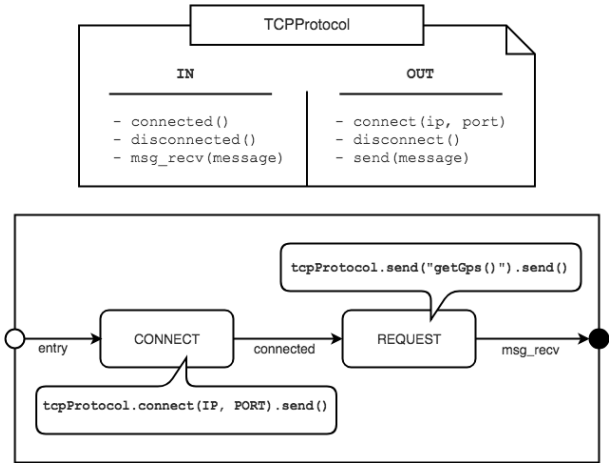

Fig. 7. Statechart for the `QUERY_GPS` state to handle TCP requests using an internal port typed with an RTS defined `TCPProtocol`

5 Conclusion

In this work, we presented a solution to the Leader-Follower challenge problem using the Papyrus-RT modeling environment. While the tool was in general well-suited to address the challenge, several weaknesses were noted. The most prominent being its reliance on C++ as an action language and lack of built-in TCP communication support. Several third-party extensions, some of which are still under active development, are attempting to curb some of these challenges.

References

1. Bagherzadeh, M., Hili, N., Seekatz, D., Dingel, J.: Mdebugger: a model-level debugger for UML-RT. In: Proceedings of the 40th International Conference on Software Engineering: Companion Proceeedings. pp. 97–100. ACM (2018)
2. Hili, N., Posse, E., Dingel, J.: Calur: an action language for UML-RT. In: 9th European Congress on Embedded Real Time Software and Systems (ERTS 2018) (2018)
3. MDETools'18: Challenge Problem. `https://mdetools.github.io/mdetools18/challengeproblem.html` (2018), [Online; accessed 10-July-2018]
4. Posse, E., Dingel, J.: An executable formal semantics for UML-RT. Software & Systems Modeling **15**(1), 179–217 (2016)
5. Selic, B.: Using uml for modeling complex real-time systems. In: Languages, compilers, and tools for embedded systems. pp. 250–260. Springer (1998)