Navigating the rover with xtUML

Keith Brown

One Fact Inc, Lafayette IN, USA
Keith.brown@onefact.net

Abstract. In this paper we describe the use of open source Model Driven Engineering (MDE) tools to design, test, and deploy an application for the MDETools'18 challenge problem. The solution application models data, control, and processing. Leader and follower positional data is stored in modeled class instances and used to navigate the follower. The control algorithm flow is sequenced with state machines. State and operational processing is specified with a platform-independent action language. We address the issue of interfacing the model to hand-written code. The model is simulated by being executed interpretively inside the BridgePoint model debugger before translation to target code. A model compiler translates the model to ANSI C with a built-in runtime architecture. The target executable runs and produces behavior identical to the interpretive simulation.

Keywords: executable modeling, xtUML, BridgePoint, open source tools, Shlaer-Mellor, Rover Challenge

1 Introduction

This paper highlights platform independent application modeling and multi-target execution. This exploration is accomplished within a specific context that depends upon a challenge problem, a modeling dialect and tooling. Our goal is to demonstrate a solution to the challenge problem using a production MDE tool and to highlight the strengths and weaknesses of the tool and associated modeling methodology.

The MDETools'18 Challenge Problem [16] provides an application *problem space* within which to explore executable modeling and MDE tools. A clear and well specified problem narrows the scope of the requirements for the language and tooling. A challenge problem establishes boundaries within which the variants of language and tooling can be observed with a measure of consistency. Specifically, the MDETools'18 Challenge Problem requires the creation of a modeled application to drive a rover that follows a leader rover at a safe distance.

Executable Translatable Unified Modeling Language (xtUML) [1] supplies a language and methodology in which an executable application model can be expressed. The Rover Challenge solution model [2] described in this paper is expressed in xtUML.

BridgePoint [9] provides a tool platform that supports editing, verifying and translating xtUML models.

The design of the application model solution is described first. Pieces of the model are illustrated as graphical and textual xtUML extracted from the BridgePoint tool. The design explains how the model integrates with the Rover Challenge simulator using hand-craft code to stitch the model to the socket layer. The testing is described using an interpretive model simulation platform. Deployment is explained next in the context of model translation using a model compiler targeted to ANSI C with a simplistic run-time executive. The strengths and weaknesses of the approach highlight the distinctive benefits and challenges faced by developers working on similar applications using xtUML and BridgePoint. A section proposing next steps outlines opportunity for future work. The paper draws to a conclusion with a summary of the findings.

2 Background

xtUML is a modeling dialect descended from the Shlaer-Mellor Method of Object-Oriented Analysis and Recursive Design. [3,4] Modelers use xtUML to construct views of the application data, control, and processing. Data is modeled as system component configurations and class diagrams. Control is modeled in state machines. Processing is expressed in action language.

BridgePoint is an Eclipse-based [5] tool for xtUML that supports graphical modeling, interpretive model execution, and model translation to target code. It is Open Source Software licensed under Apache License and Eclipse Public License. [7,8]

BridgePoint supports two action languages: Object Action Language (OAL) [17] and Model Action Specification Language (MASL) [18]. The solution model described here uses OAL for simplicity of introduction to new users because BridgePoint includes a C model compiler for OAL in the installation package. OAL model compilers exist to translate to C, C++, SystemC, and Java.

MASL is a textual syntax for another Shlaer-Mellor derived modeling language and is a recent addition to BridgePoint. The MASL model compiler translates to C++.

3 Design

The Rover may be viewed in simple terms as a “read and respond” application. Dynamic data is read periodically and used to calculate an appropriate response that navigates the follower in a reasonable way. It is a classic feedback control loop. Our application models the transfer and caching of the relevant leader and follower positional data. Additionally, our application models the Rover follower navigation algorithm with state machines and action language processing.

3.1 System Model

xtUML supports UML components and interfaces to define functional black boxes and the communication channels between them. A component diagram (Fig. 1) formalizes the provided and required interface connections between components in the application.

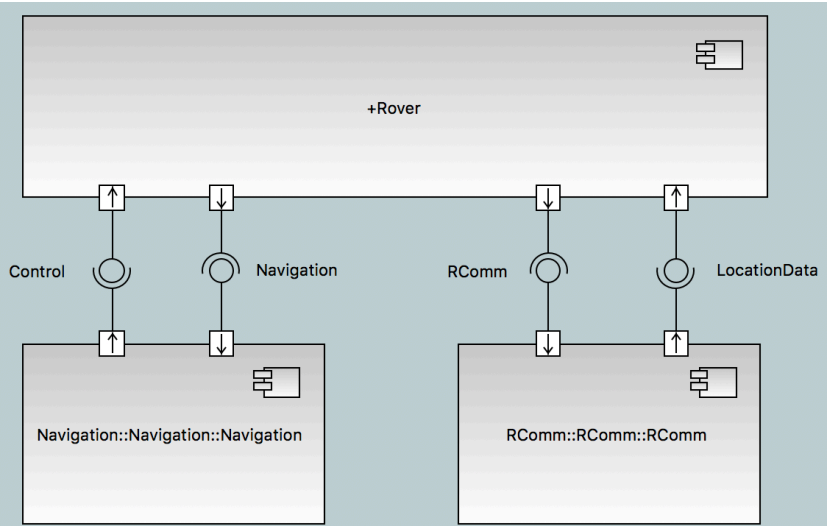

Fig. 1. Component diagram

The *Navigation* component contains the behavioral models that make up the heart of this application. The *RComm* component abstracts the socket communication layer. The *Rover* component facilitates the data passing between these two components; providing a means to do "impedance matching" between the application components to account for differences in the interfaces.

A guiding design decision is a choice to treat the *Navigation* and *RComm* components as if they were off-the-shelf modules, modeled independently by different developers and with flexibility around the definition of the interface messages provided and required by each. This not only simulates a likely real-world scenario, it supports a future enhancement to the model that requires this construct.

xtUML supports bidirectional interfaces, meaning a single interface can support both "to provider" and "from provider" messages. For the challenge application we have chosen to use unidirectional interfaces, where all messages are "to provider". This design choice also eases implementation of a future enhancement.

3.2 Data Model

Data is gathered periodically and cached in class attribute values. Since both the leader and follower provide positional information, we factor that data out into a superclass. In our application, there is need for only one instance each of leader and follower. Each of these instances has a related rover supertype instance as shown in Fig. 2.

A structured data type named `position` has x and z members of type real. For each rover, the application tracks the current position, previous position, and prior-to-previous position with attributes of the aforementioned structured type `position`. The distance value is important to the application control algorithm and is cached in the leader instance. The compass bearing of the follower is cached but not used in the simple navigation algorithm.

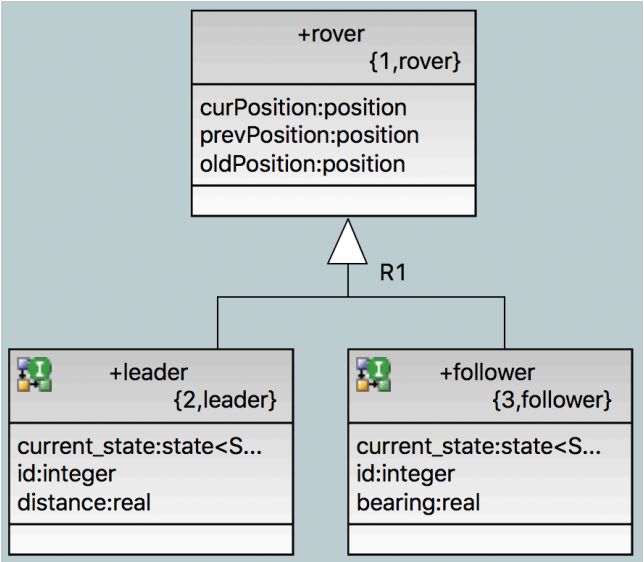

Fig. 2. Application data model

The application updates the cache of current information for each rover on a periodic basis. This polling loop is implemented as a state machine on the leader instance.

Note that our application does not model the behavior of the leader. It does model the information about both the leader and the follower by capturing and storing relevant position and distance data. This data is gathered via socket communication between the application and the simulation environment provided along with the challenge problem.

3.3 Control Model

The Rover application uses a state machine (Fig. 3) for primary navigation control of the follower. The control algorithm uses each rover's position information to make decisions about what the leader is doing and how the follower should respond.

Our application has made the design decision to disassociate the data polling state machine and the navigation state machine. Each state machine simply runs periodically and independently.

The state machine sends events to itself to transition from state to state based on the action to be taken. Each of the action states transition to a common finalization state. This state machine is driven by a periodic event to self. A constant value in the data model defines how often the periodic event fires and therefore defines how often we make a navigational decision and act on it.

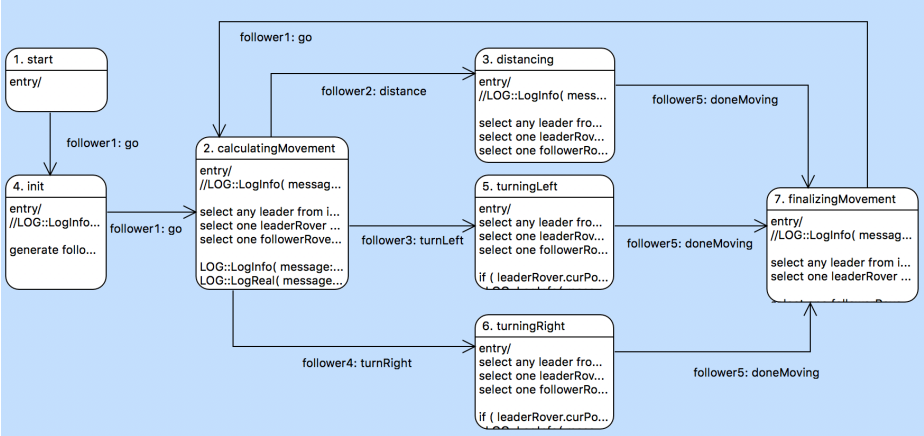

Fig. 3. Follower state machine

xtUML uses a "run to completion" rule for state actions. That is, xtUML specifies that a state action cannot be interrupted in the middle of processing. Events are queued as they arrive and processed in between states, with events to *self* handled before events coming in from outside the class.

3.4 Processing

The application activities (that is the functionality inside states, operations, and functions) were written in Object Action Language (OAL) rather than a specific target language like C, C++, or Java. OAL supports the xtUML method's philosophy of platform independent models (PIMs). The tool is able to run the PIM interpretively. The PIM may also be translated to a target language using a model compiler and then compiled and executed natively.

In the context of this application, an xtUML *domain function* is used as the entry point to activate the model to collect data and take action to drive the Rover. In our application, this function creates and relates the necessary class instances for the data model, queries the communication layer for the first data update, starts a recurring timer to fire the data polling loop, and signals the simulator we are ready to begin.

Inter-component interface messages were used the activate functionality and pass data between black box blocks known as Components. Within a component, class operations and the aforementioned state activities perform action.

The *calculatingMovement* state analyzes the track of the leader to decide if it is actually making a turn or just drifting slightly to one side or another. A constant value in the model defines the window of allowed drift. This state also compares the current position of each rover to determine if the follower is too far to one side or the other of the leader. A constant is also used to define this window. Based on these determinations, the follower continues straight and performs a distance check or turns left or right.

The *distancing* state checks to see how far apart are the leader and follower. As a safety step, if the leader and follower are way too close or way to far apart the follower

simply stops. An additional safety check stops the follower if we determine that a forward movement will cause the follower to pass the leader.

The *turningLeft* and *turningRight* states initiate a turn by applying uneven power to the wheels. A safety check is in place in these states as well. That is, if the leader is already to the left of the follower, we override the decision to turn the follower left. If the leader is already to the right of the follower we override the decision to turn right.

Once the movement action is performed, the control goes through a finalization step that updates the leader and rover track information in the data model. By performing the track update here, we avoid any potential issue with the data polling loop filling up the track only with current values.

3.5 Interfacing with Handwritten Code

Software applications often rely on pre-existing libraries or services. These pieces may be highly reusable functionality or tools to fulfill a basic need. The Rover application is no exception as it performs socket communication, informational output, and time-based event generation. The application must interface to this external and/or hand-written code. xtUML provides External Entities and realized Components to model functionality that is backed with non-modeled code. The Rover application uses both for different pieces of the application model.

External Entities (EEs) are best suited to simple call-return interfaces and have been a part of the method since the beginning. The xtUML Debugger and C model compiler include built-in functionality that implements the commonly used EEs for *Time* and *Logging*. Using these EEs, it is simple for a modeler to perform timed delays and output messages to the console.

Realized components are better suited for larger functional pieces of external behavior. In the Rover application, the socket-based communication layer is modeled as a realized component named *RComm*.

xtUML Debugger relies on a Java implementation of a realized component. The Rover application provides this hand-craft code in *RComm.java* that implements the messages of the provided and required interfaces of the component.

4 Test

xtUML models may be executed interpretively inside the BridgePoint xtUML Debugging perspective. This capability implements graphical debugging in the same way as the Java and C debuggers built into Eclipse. This provides a familiar experience to the user with views for the call stack, local variables and features such as breakpoints and action language stepping.

The interpretive runtime architecture interfaces with the aforementioned Java-based implementation of the communication layer. Without leaving the modeling tool we were able to execute the action language directly and have it communicate with the simulator to query positional information from the leader and drive the follower.

Our testing process began by running the leader on a path with no turns. In this way, the control algorithm processing simply had to analyze the current distance and make

changes to the follower speed to attempt to maintain a proper distance. This allowed us to discover that even when the rover is going straight, its positional x value drifts a little to the left or right and made it clear our application needed to distinguish between drift and a purposeful turn.

We then introduced randomness into the leader's path and used the simulator to observe how well the follower was able to track behind the leader. In this phase we added turning logic to the follower control. This application does not lend itself to setting breakpoints and stepping since we could not pause the leader inside the simulator. Instead we let the leader run its course and used console logging to see what data the follower was analyzing and what decisions it was making.

Once our testing and progressive refinement yielded a follower that behaved reasonably well, we added translation and native execution to the development process. This step did not preclude any further model development or interpretive test, it simply provided a milestone to measure our progress in the application development.

5 Deployment

xtUML models rely on a model compiler toolchain to translate the application model (both structure and action language) into a target language such as C. Our application uses two pieces of open source software during translation: pyrsl [6] and MC-3020 [15].

Together these tools load the model artifacts and process code generation rules to transform the application into C.

In the context of the Rover application, it is particularly beneficial that the C model compiler creates a lightweight and efficient runtime architecture to dispatch and receive the modeled events and application processing. This leaves the modeler free to focus on implementing and improving the Rover application itself.

Because the xtUML method defines strict rules regarding event dequeuing and state processing the application behaves the same when run interpretively in xtUML Debugging and when the translated and compiled application is run natively. Simply put, compiled execution yields the same behavior as simulation.

6 Tool Strengths and Weaknesses

6.1 Strengths

xtUML is suited to handling state-based operational behavior. The low-latency, periodic, decision making algorithm is handled cleanly with the xtUML approach to structural and behavioral modeling. As a method of explanation, the graphical data model and graphical state machine make it easy to show how the data cache is structured and how the control algorithm flows. As the control algorithm is expanded with additional state logic and transitions, these new paths and their connection into the existing flow remains clear.

The platform independent action language is simple and expressive as a means to execute the model interpretively and natively. The OAL activity editor supports model-

aware content assistance (Fig. 4). This is a useful feature of modern language editors, because it reduces bugs and improves efficiency.

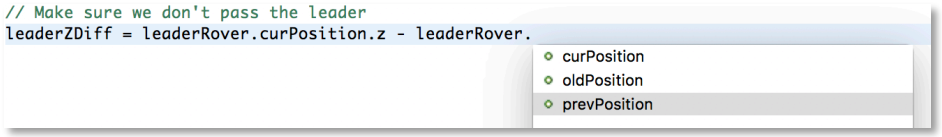

Fig. 4. Editing OAL using content assistance

One of the key advantages of using a platform independent model is the ability to re-target to new platforms with minimal impact and rework required. In this case, we created a version of the realized component communication layer in C (in *RComm.c*). The model required no changes in order for the translated C version of the application to interface with this implementation of the realized component. The C model compiler provided a head start by creating a structural template version of *RComm.c*. The author then had to implement the socket communication inside the various command messages.

The ability to execute the model interpretively simplified the debugging process and is more efficient than translating to and compiling target code to run. Interpretive model execution sped up testing changes by removing the need to build. The translate, compile, and launch steps were unnecessary. Changes could be made to the model and quickly tested.

The built-in execution architecture of the model compiler saves the user a significant amount of development work to create an application that is runnable and allows the modeler to spend time working on the application itself rather than infrastructure. The ability to have confidence that the native application behavior will mirror the simulation allows the user to test early and often.

6.2 Weaknesses

MDE tools have an inherent barrier to entry that typical programming in C or Java do not. Namely, the user must learn both the methodology and the tools. Before getting started a user must first determine "is modeling worth the cost?" Fortunately, the availability of free, open source MDE tools takes the initial monetary cost factor out of the equation. In this case, a history with both the method and tool provided a head start to the development of this application. Novice users would have to climb this learning curve when using any MDE tool instead of text-based, target language programming.

A specific BridgePoint weakness is the lack of built-in libraries. OAL has limited capabilities to perform mathematical operations. As the model is improved, a mathematics EE will be required to perform better data analysis. Similarly, we struggled with the C implementation of the socket communication layer. We had to go so far as using a packet sniffer (Wireshark [10]) to try to determine why the Java realized component communicated perfectly well with the simulator and the C realized component did not.

A modeled abstraction of socket communication would have allowed us to spend more time working on the control algorithm and less time debugging socket problems. Ironically, this is the exact benefit modeling should be providing to the user… moving beyond target language implementation details. Yet, the reality is that the model must interface with the outside world and we cannot always escape the sticky details of implementation languages.

A final weakness to note are tool bugs. BridgePoint has its own set of issues [11], and since BridgePoint is built on Eclipse it therefore inherits any bugs in the underlying Eclipse infrastructure. During the development of this application, two serious Eclipse issues [12,13] were encountered that caused application crashes. We were able to work around the problems, but they were frustrating and could easily dissuade a new user from using MDE because of the underlying tool issues.

7 Future Enhancements

7.1 Model Improvements

Few applications are ever completely finished, and the solution model presented here is no exception. Expanding the track log, especially of the leader, would be a useful enhancement. An option here is an instance list of track points with a relationship to the rover class for the first and last elements. The track point instance list is easily implemented in the data model with a reflexive association to the track point class.

With more data in hand, the control processing could perform improved analysis of the leader movement and make better decisions about the course of action to take. We envision the analysis determining how sharply the leader is turning, if the leader is speeding up or slowing down, and a whole new addition to the control algorithm to recover, rather than stop, if the follower gets lost.

7.2 Teaching Materials

Learning a modeling methodology requires time and effort, and example applications are a great help to new users. Easy access to these examples lowers the barrier to entry. We see the integration of the Rover example directly into the tool documentation as a positive step to take.

We also note that video-based instruction is a valuable tool some users prefer for instruction or direct guidance through a series of steps. The visual nature of this application is well-suited to video teaching. The xtUML.org website contains an online teaching course backed by videos hosted on YouTube [14]. The interchangeable realized component communication layer of the Rover application would make a nice addition to the course.

MASL support is a new addition to xtUML and therefore has a limited set of demonstrative examples. The Rover application presented in this paper uses OAL for processing. Converting this processing to MASL and integrating the MASL model compiler into the toolchain would yield another useful example.

8 Conclusion

The previous sections spotlight platform independent application modeling and multi-target execution. First, the context is established consisting of an application challenge, a methodology, and a modeling toolchain. The specific context is Rover Challenge.

The steps taken to develop the challenge application using the xtUML method and BridgePoint tool are described. Modeling the system, data, control and processing are illustrated, and the associated model artifacts explained. An explanation is given on how the models interface with existing modules of hand-written code. A review is given of the testing and deployment process experienced.

The strengths of the approach were realized in the development of the solution model. Interpretive execution of the model enabled early testing on an incomplete model. This early testing was especially helpful to work out the communication with the simulated environment. The promise of multi-target execution was experienced when the application was translated. The behavior observed in the interpreted simulation matched the behavior of the application running as compiled C code.

Many of the struggles associated with model-driven engineering were experienced and described in the concluding sections. The tools add another dimension of complexity when debugging lower level interactions such as socket communication. Weaknesses in the action language would require special library integration steps.

Now that the application is running, future work is anticipated to explore performance improvements and additional dialects and architectures for deployment.

References

1. xtUML Homepage, https://xtuml.org
2. Rover xtUML application, https://github.com/xtuml/models/tree/master/applications/rover
3. Stephen Mellor, Sally Shlaer - Modeling the World in Data, ISBN 978- 0136290230.
4. Stephen Mellor, Sally Shlaer - Object Lifecycles, ISBN 978- 0136299400.
5. Eclipse Homepage, http://www.eclipse.org
6. Pyrsl Homepage, https://github.com/xtuml/pyrsl
7. Eclipse Public License 2.0, https://www.eclipse.org/legal/epl-2.0/
8. Apache License 2.0, https://www.apache.org/licenses/LICENSE-2.0
9. BridgePoint Nightly Build Download Page, https://xtuml.org/download/
10. Wireshark Homepage, https://www.wireshark.org/
11. BridgePoint Issue Tracker, https://support.onefact.net
12. Eclipse bug 1089382, https://www.eclipse.org/forums/index.php/t/1089382/
13. Eclipse bug 479888, https://bugs.eclipse.org/bugs/show_bug.cgi?id=479888
14. xtUML YouTube channel, https://youtube.com/xtuml
15. BridgePoint Model Compilers, https://github.com/xtuml/mc
16. MDETools'18 Challenge Problem, https://mdetools.github.io/mdetools18/challengeproblem.html
17. Object Action Language, https://github.com/xtuml/bridgepoint/tree/master/src/org.xtuml.bp.doc/Reference/OAL
18. Model Action Specification Language, https://github.com/xtuml/bridgepoint/tree/master/src/org.xtuml.bp.doc/Reference/MASL