PapyrusRT: Modelling and Code Generation

Ernesto Posse
eposse@zeligsoft.com

Zeligsoft

Abstract. In this talk we introduce PapyrusRT, an open-source, industrial-strength model-driven development environment for real-time and embedded systems, implementing UML-RT [2,3], a UML-based language. PapyrusRT is implemented on top of Papyrus, an Eclipse modelling tool for UML, SysML, and EMF models. We describe the motivations for this project and in particular for the need of an open-source environment. We provide a brief summary of the UML-RT language and give a brief description of the tool itself. Then we give an overview of the code generation process and its architecture, with emphasis on its extensibility.

1 Introduction

Developing software for real-time and embedded systems (RTES) poses many challenges and model-driven engineering (MDE) methods have been proposed as a way to address them. UML [1] has become the *de facto lingua franca* of the software modelling world, with many tools, both commercial and non-commercial supporting parts of the language. Nevertheless, UML is a large and complex language and mastering it is itself a difficult task. For this reason, more specialized modelling formalisms and languages have been proposed, which are better adapted to the needs of RTES. One such language is UML-RT.

UML-RT is a language that is based on UML (it is defined as a UML profile) and simplifies it in order to tame software complexity, better capture high-level system architecture, focus on the concurrent structure of a system, and improve the analyzability and predictability of a system's behaviour. This is achieved mainly by restricting the language to two kinds of diagrams: composite structure and state machine diagrams. These diagrams have additional restrictions over general-purpose UML. In addition to these syntactic restrictions, UML-RT has a more precise execution semantics, designed with the needs of soft-real-time systems in mind.

Although based in UML, UML-RT predates it (and influenced the definition of the UML 2 standard). UML-RT has its roots at the Telos project at Bell Northern Research (part of Nortel) in 1987. In 1992, this project led to spin-off company called ObjectTime which released its namesake development environment implementing the core language. In 1994, an influential book on the language and the methodology was published [2], and the language became known as ROOM. In 1998, the name UML-RT was coined for the UML profile

describing the ROOM language [3]. In 2000, Rational Software acquired ObjectTime and turned the tool into Rational RoseRT. In 2002, Rational was acquired by IBM and in 2006 they migrated RoseRT to the Eclipse IDE where the tool was rebranded as IBM Rational Software Architect Real Time Edition, or RSA-RTE for short.

In all these incarnations, UML-RT has been successfully applied to large-scale industrial projects. However, all these implementations have been proprietary. This has presented a challenge to its users. As with all proprietary software, users are bound to the vendor for support, updates and customization. In the context of RTES, users rarely want a one-size-fits-all solution. They typically want tools that are better suited to their specific needs. This entails customizations and/or extensions to the tools which are difficult or impossible to make. This requires a willingness on the vendor's part to adapt a generic tool to each user's specific needs.

It is out of these considerations that the need for an open-source development environment for UML-RT arose. Under the open-source framework, users don't depend on the vendor to adapt the tool. This is where PapyrusRT comes in. PapyrusRT is a new open-source implementation of a full UML-RT development environment, including a graphical modelling environment, a code generator and a runtime system. This full set makes UML-RT models executable.

PapyrusRT is implemented on top of Papyrus, a well-known UML modelling environment on Eclipse. Papyrus was chosen as the basis because it is open-source, it has a rich user-interface, it already supports the latest OMG UML standard (2.5.1) and is part of the rich Eclipse ecosystem, from which a wide range of components, tools and resources can be leveraged.

In this presentation we give a brief tour of the language, the tool and the code generation process.

2 UML-RT

2.1 Capsules, ports, parts and state machines

The central concept in UML-RT is the *capsule*. A capsule is, as the name suggests, an encapsulated entity. It is a *class* in the object-oriented sense, more precisely an *active* class, meaning a class whose instances have autonomous behaviour, specified by a *hierarchical state machine*. Furthermore, capsules have a well-defined *interface* consisting of a set or *ports*, and capsule instances can communicate with their environment (other capsules) exclusively through these ports. Each port is typed by a *protocol* which defines the kinds of *messages* or *signals* allowed. Capsules may also define internal structure consisting of *parts* (sub-capsules), which are properties (in the UML sense) typed by some other capsule. Parts are connected by linking their ports with *connectors*. A connector can link only two ports, but ports and parts can be replicated to obtain different connection patterns. Figure 1 on page 3 shows a structure diagram depicting a capsule called "Top" with two parts called "pinger" and "ponger", typed by

some capsules named “Pinger” and “Ponger” respectively, each with a port and a connector linking them.

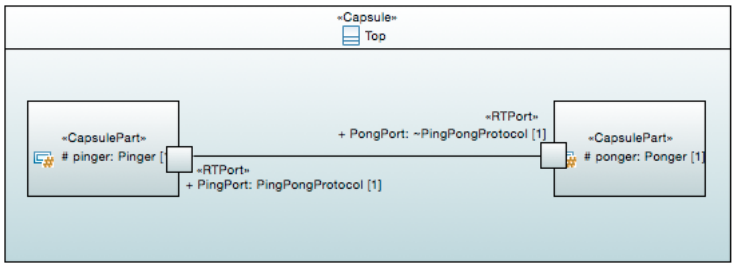

Fig. 1. A UML-RT composite structure diagram for a capsule.

A port in the outer boundary of a capsule can be connected to an internal part, in which case it is called a *relay* port, as it simply relays messages to and from the internal part. A boundary port not connected to an internal part is called an *external* port, and messages arriving there are handled by the capsule's state machine. Similarly, it is through external ports that the capsule's state machine can send messages outside. Non-boundary ports are called *internal*, and are used by the capsule's state machine to communicate with the capsule's internal parts.

Protocols define three kinds of messages: input, output and input/output. Messages can be parametrized. This allows messages to carry data as a payload. Ports can have two kinds of role: *base* or *conjugated*. A conjugated port is one where the protocol messages' direction is flipped: an input message in a base port is an output message in a conjugated port, and vice-versa. This means that messages from/to a base port at one end of a connector correspond to messages to/from a conjugated port at the other end of the connector, unless one of the ports is a relay port.

The behaviour of a capsule is described by a hierarchical state machine. Figure 2 on page 4 shows a state machine diagram. These state machines are like UML state machines with some restrictions: there are no AND-states (orthogonal regions), that is, each composite state has exactly one region, so at any point the state machine is in exactly one state or pseudo-state. Therefore, there are no fork or join pseudo-states. The only pseudo-states allowed are initial, deep-history, choice, junction, entry, and exit. Transitions cannot cross state boundaries. Transitions can form a chain, but to enter or exit composite states they must go through entry and exit points explicitly. States may have entry and exit actions, but they do not have “do” actions. Shallow history is not supported and neither are final states. Transitions arriving at the boundary of a composite state are deemed to arrive at its deep-history pseudo-state.

Capsule parts can have one of three *roles*: *fixed*, *optional* or *plug-in*. Fixed capsule parts are parts whose instance(s) are owned by the capsule that contains

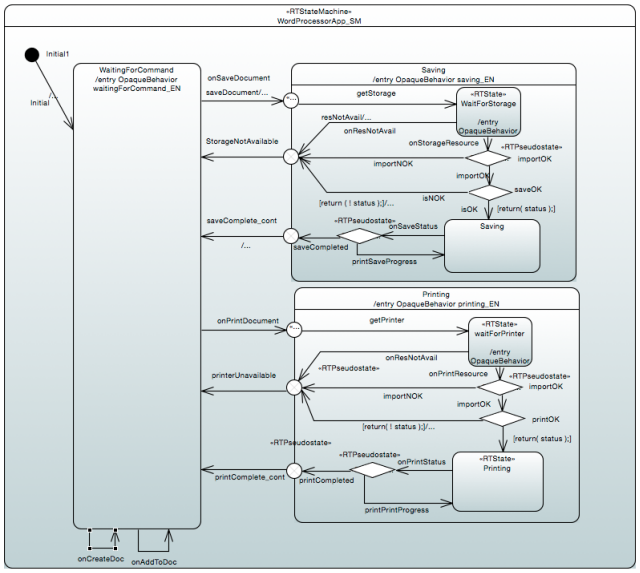

Fig. 2. A (hierarchical) state-machine diagram for a capsule.

the part, and are created and destroyed at the same time as the containing capsule is created and destroyed. Optional capsule parts are parts whose instance(s) are also owned by the containing capsule, but are created and destroyed dynamically, by some action in the capsule's state machine. Plug-in capsule parts are parts whose instances are not owned by the containing capsule. Rather, they are created elsewhere, and are "imported" and "deported" by some action in the capsule's state machine. Since they are not necessarily owned by the capsule importing them, plug-in capsule instances can be shared between different capsules.

Another important feature is that of *service ports*. Ports can be marked as *service provision points* (SPPs) or *service access points* (SAPs). These represents ports that provide (resp. access) some service to other capsules, usually in a different architectural layer. Unlike normal ports, service ports are connected dynamically. This is, the connection between an SAP and an SPP is performed at runtime by some action in the relevant capsule's state machine.

2.2 Execution semantics

As mentioned above, capsules have autonomous behaviour and therefore it is natural to think of them as threads, with each capsule instance executing concurrently with other capsule instances. However, UML-RT makes a distinction between capsules and threads. The concept of a capsule as an entity with autonomous behaviour is a modelling concept, whereas the concept of thread is a deployment concept. Each capsule is assigned to a thread, and more than one

capsule may be assigned to the same thread. The execution of a UML-RT model is carried out by a *runtime system* (or RTS) which consists of one or more *controllers* and a *runtime services library*. Each controller runs in its own thread and maintains and executes the behaviour of the collection of capsules assigned to its thread. Hence, when we assign a capsule to a thread, we are assigning it to a controller. Assigning a capsule to a particular thread/controller can occur at runtime in the case of optional and plug-in capsule parts.

A controller has, in addition to the collection of capsules assigned to it, a message (priority) queue. Messages in the queue can come from capsules in other controllers or in the same controller. The controller runs a main loop, extracting the message with the highest priority from the queue (or the first one, if all have the same priority), and directs the message to the target capsule, or more precisely, passes the message to the target capsule's state machine.

The execution semantics of state machines mandate a *run-to-completion* semantics (RTC), that is, an incoming message is fully processed before the next message is processed. This means that the state machine is always in a stable state when a message arrives, and it follows a full transition chain, possibly going through several pseudo-states, and executing the corresponding transition actions as well as exit and entry actions encountered along the way, until it reaches a stable state, at which point the RTC step is finished and the state machine is ready to process the next incoming message.

In addition to handling messages directed to its capsules, the controller is in charge of managing the capsules' lifetimes for capsules that are created, destroyed, imported, or deported dynamically.

The runtime services library provides, as the name suggests, a set of common services, in particular services related to logging, timing, and dynamic capsule operations in addition to the messaging operations already discussed.

3 The tool

Figure 3 on page 6 shows a screenshot of the tool. The central view has the main canvas or model editor with a palette of elements on the right. The bottom view includes a Properties view for the properties of the currently selected element. This includes standard UML properties, UML-RT-specific properties, applied stereotypes, advanced properties, etc. The bottom also has views for validation, error reporting, etc. The top-left shows the project explorer to add and manipulate projects. The middle-left shows the model explorer which presents the abstract syntax view of the model and imported libraries. The bottom-left shows an outline view.

Code generation is triggered by selecting the root element in the model explorer and choosing either "UMLRT Code Generator" or "UMLRT Code Generator (regenerate)". The first performs incremental generation (or full generation if the model has not been previously generated). The second regenerates the whole model.

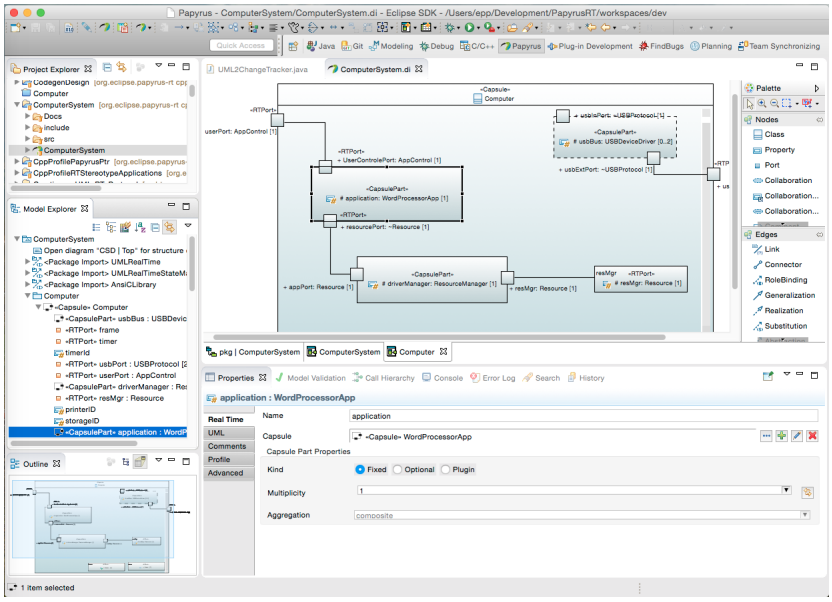

Fig. 3. PapyrusRT screenshot.

In an application there must be a "top" capsule, which corresponds to the "main" capsule. By default it is a capsule called "Top", but the user may select any other capsule as the top-capsule by right-clicking on it in the model explorer and selecting "Set as Top" or "Generate as Top".

If code generation succeeds, there will be a CDT project created (or updated) in the workspace, named after the input project with the "_CDT_project" suffix. This project includes the necessary Makefiles to compile the generated C++ sources and link it to the RTS, producing an executable application. The Makefiles use a variable called "UMLRTS_ROOT" for the location of this library. The code generator attempts to assign it the correct value according to the installation, but in circumstances where this location cannot be determined, the user may still specify it manually as an environment variable.

In its current version (0.7.1), the generated code compiles C++03 with GCC 4.6.3, targeting Linux, with some Windows support.

4 Code generation

PapyrusRT generates executable C++ code from the model. This includes both structural and behavioural elements of the model. The code generator can run either as a standalone application, or within the Eclipse environment. The input is a UML model with the UML-RT profile applied, and possibly other profiles, such as the RTCppProperties profile used to customize the generated C++ code.

The output is an Eclipse CDT project, including all generated source files and Makefiles required for building (compiling and linking).

The generator supports *incremental generation*, that is, if code has already been generated, then a run of the code generator will only generate those elements that have changed, and their dependent elements. For example, if a protocol changes, the code generator will regenerate the protocol, and all capsules with a port typed by the protocol.

Model validation is performed by a separate operation. Nevertheless, the code generator does perform some limited validation and sanity checks as it proceeds. When errors in the model are encountered, these are reported to the user. Nevertheless, there are certain types of errors which cannot be easily detected during code generation. These are errors, such as a model element missing a stereotype, that would require the code-generator to know the modeler's intentions.

4.1 Model transformation

The code-generation process is structured as a model transformation. More precisely, it is a sequence of model-to-model transformations with a final model-to-text transformation. This is depicted in Figure 4 on page 7.

Fig. 4. Model transformation.

In the first phase, the UML model is translated into xtUMLrt, a simplified intermediate representation which contains all the required UML-RT constructs. The xtUMLrt meta-model is intended to simplify UML, in order to simplify the generator itself, while allowing customization, isolating the generator from the toolset, and providing a common language that allows its eventual extension to support xtUML, and, potentially, other approaches.

Once the model has been translated into xtUMLrt, elements other than state machines are translated into a simplified C++ meta-model, from which the final phase of generating the actual source C++ files and CDT project is done. The C++ meta-model isolates the generator from issues such as formatting, body/header file generation, file regeneration avoidance, CDT project and makefile generation, etc. For example, if we need to generate a class in C++, then the class must be declared in a header file and defined in the corresponding implementation file. Using the C++ meta-model, we can simply create a "`CppClass`" object (where `CppClass` is the meta-class used to represent C++ classes in the meta-model) without the need for separate rules to generate the header and the implementation and keep them coordinated. Instead, the last phase takes that `CppClass` object and writes the required declarations in the header and definitions in the implementation file.

State Machines are also translated to the C++ meta-model but go through several sub-stages that expand the inheritance hierarchy and flatten the state machine.

During the translation from xtUMLrt to the C++ meta-model, the generator collects, for each element to be generated, the set of dependent elements. For each kind of element to be generated, a specialized generator class implements the specific translation. This is, there are specific generators for basic classes, capsules, state machines, protocols and a special generator for deployment that builds the deployed structure.

4.2 Architecture

The overall architecture if the generator is depicted in Figure 5 on page 8.

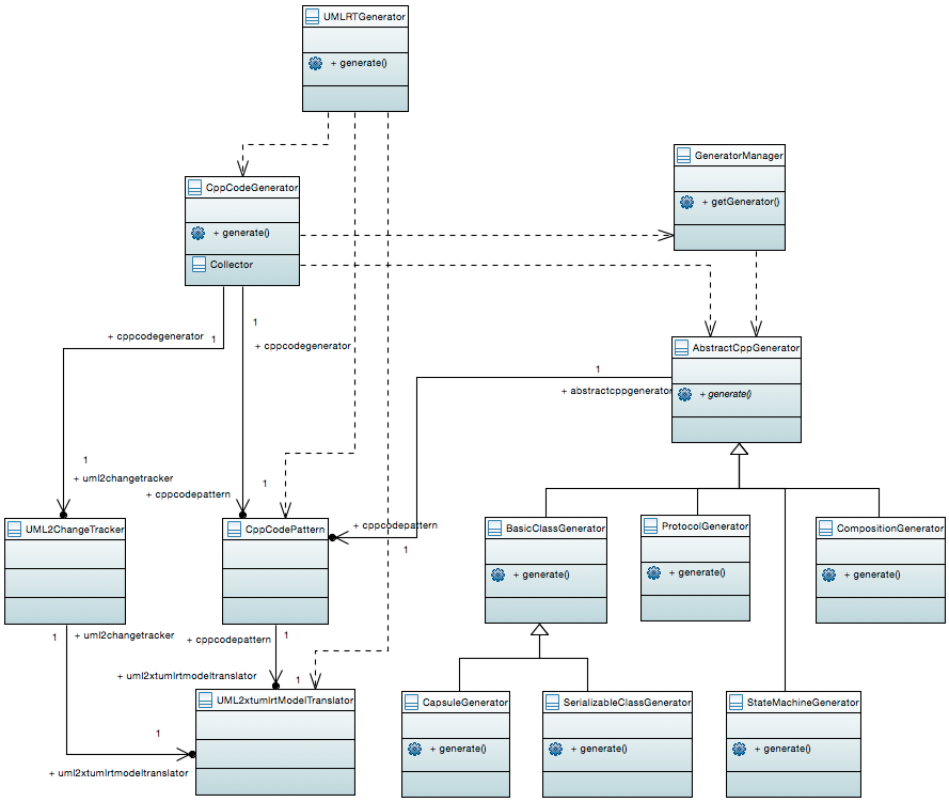

Fig. 5. Code generator architecture.

The `UMLRTGenerator` class on top is the "director" that executes the transformations described in Figure 4 on page 7. It contains references to:

- `UML2xtumlrtTranslator`: the translator from UML to xtUMLrt.
- `CppCodeGenerator`: the core generator from xtUMLrt to the C++ meta-model.
- `CppCodePattern`: the "C++ code pattern", a class that provides a facade to the C++ meta-model with multiple factory methods, and which caches generated elements that can be shared by multiple parts of the generator. It also contains the main model-to-text method.

The `CppCodeGenerator` also has a reference to the `CppCodePattern`, as well as references to:

- `GeneratorManager`: the "generator manager", which keeps track of the individual element-specific generators.
- `Collector`: the "collector", which creates element-specific generators for each element and their dependent elements.
- `UML2ChangeTracker`: the "change tracker", which keeps track of which elements have already been generated and which have changed.

`CppCodeGenerator` first invokes the collector to build a list of applicable generators. The collector obtains element-specific instances from the generator manager which provides instances from the built-in generator classes, or from generators provided as extensions. Generators for elements which have already been generated but have not changed are pruned from the list. All element-specific generators are subclasses of `AbstractCppGenerator`.

Once all the generators for elements to be generated are collected, and the list pruned according to which elements have changed, the code generator invokes the `generate` method for each element-specific generator instance. This will result in the specific generator leaving the resulting C++ model elements in the `CppCodePattern` instance. Finally the code generator will instruct the `CppCodePattern` instance to perform the model-to-text transformation by invoking its `write` method.

The code generator can be extended with the standard Eclipse extension mechanism: the `org.eclipse.papyrusrt.codegen.cpp` plugin contains an extension point called `generator` which has two parameters: a `type` and a `class`, which must be provided by a user extension. The `type` parameter is one of the following: ClassGenerator, EnumGenerator, CapsuleGenerator, StateMachineGenerator, EmptyStateMachineGenerator, ProtocolGenerator, StructuralGenerator, or ArtifactGenerator. The `class` parameter is a class implementing the `AbstractCppGenerator.Factory` interface which has a `create` method returning an instance of a subclass of `AbstractCppGenerator` for a given model element. This is, a user-provided generator must subclass `AbstractCppGenerator`, and must provide a factory method that creates its instances for a given model element. The `AbstractCppGenerator` class defines an abstract `generate` method that must be implemented by its concrete subclasses. For example, a custom `StateMachineGenerator` must inherit from `AbstractCppGenerator`, and provide its factory method which will receive as input a state-machine instance. The generator manager will invoke this factory method during the collection process described above, overriding built-in generators.

5 Final remarks

Open-source software presents both challenges and opportunities for software developers in general, and for the MDE community in particular. While an OSS project may not necessarily have the same resources as a commercially-backed product, the transparency and ability of the community to contribute to it, may provide an edge leading to greater adoption. Both industry and academia benefit from such endeavor. Industrial users gain unrestricted access and that allows them to develop their own custom and domain-specific variants without incurring on the costs of developing a fully fledged product from scratch. Academics can develop their ideas and proofs of concept on an industrial-strength platform where they may reach a larger audience. One of the major obstacles for the adoption of MDE is the lack of access to mature and robust tools. While PapyrusRT is still in its early stages of development, its open-source nature provides a natural ground to grow into an environment that yields some of the core promises of MDE. It is with this philosophy that PapyrusRT was conceived.

Acknowledgements

This project is a collaboration between Zeligsoft (2009) Ltd., CEA LIST, Malina Software, and Ericsson. Bran Selic (Malina) has provided the UML-RT profile, its documentation, as well as substantial consulting in clarifying the semantics of the language. Peter Cigéhn from Tieto has provided invaluable input regarding requirements, as well as extensive testing. Andreas Henriksson from Ericsson has also provided requirements as well as contributing the RTCppProperties profile for C++ generation. IncQuery Labs contributed part of the xtUMLrt intermediate meta-model. At Zeligsoft, the project is led by Simon Redding, with Charles Rivet as Product Manager and Ernesto Posse as Software Developer working on code generation. Andrew Eidsness has been the principal software developer working on code generation and the runtime system (RTS). The RTS has been implemented mostly by Barry Maher. Other Zeligsoft contributors include Young-Soo Roh, Tim McGuire, Toby McClean and Stephanie Chafe. The group at CEA LIST, headed by Sébastien Gérard with Rémi Schnekenburger as project lead, also including Ansgar Radermacher, Camille Letavernier, Önder Gürcan and Céline Janssens from All4tec has worked on the tooling, validation import and CDT integration.

References

1. Object Management Group. UML Superstructure Specification v2.5. http://www.omg.org/spec/UML/2.5/, September 2012.
2. B. Selic, G. Gullekson, and P. T. Ward. *Real-Time Object Oriented Modeling*. Wiley & Sons, 1994.
3. B. Selic and J. Rumbaugh. Using UML for modeling complex real-time systems. Whitepaper, Rational Software Corp., 1998.