# Rialto Profile in the SMW Toolkit

Dag Björklund, Johan Lilius and Ivan Porres
TUCS Turku Centre for Computer Science
Department of Computer Science, Åbo Akademi University
Lemminkäisenkatu 14, FIN-20520 Turku, Finland
e-mail:name.surname@abo.fi

***Abstract*— We present an extension to the System Modeling Workbench to transform UML behavioral diagrams into specifications in Rialto. Rialto is a behavioral description language with formally defined semantics that supports multiple models of computation and can be compiled into compact code in different target languages.**

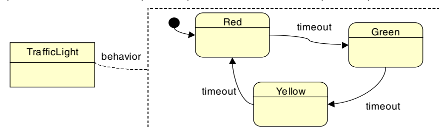
## I. The System Modeling Workbench

There are many different tools that support, in one way or another, the Unified Modeling Language (UML), but most of these tools are targeted to software developers. The System Modeling Workbench (SMW) [6] is a collection of tools targeted to those interested in doing research on new modeling languages and constructing tools to transform and derive new artifacts from models in those languages.

SMW is both a collection of libraries and a toolset constructed using these libraries. It is based on the Python programming language, an object-oriented interpreted scripting language. Python is easy to learn and features a simple yet elegant syntax. The SMW kernel library provides support for XMI [5], the standard file interchange format between UML tools, OCL-like idioms for model query and navigation, and user-defined modeling languages. A SMW modeling language is defined in a metamodel file. The SMW module for the UML language is generated automatically from the official OMG MOF files. It is also possible to create a metamodel from a UML class diagram. This is the usual case for user-defined modeling languages.

The SMW toolset contains a simple UML editor. It supports all the UML behavioral diagrams, including statecharts, activity, collaboration and sequence diagrams. There is also an extension to create real-time data flow diagrams. The editor can also be used to create, run and debug SMW scripts. We expect that the most interesting features of SMW will be provided by small scripts to extract information, transform a model or to create derived artifacts, such as source code, documentation, test cases, etc. As an example of how a SMW script looks like, we show in Figure 1 how to create a text file enumerating all the transitions in a statechart.

```
1  for t in self.behavior.transitions
2   if t.trigger:
3    |From |t.source.name| to |t.target.name| by |t.trigger.signal.name
```




```
Script Output
From Red to Green by timeout
From Green to Yellow by timeout
From Yellow to Red by timeout
```

Fig. 1. SMW Script to Enumerate Transitions

The first line of the script iterates through all the transitions in the state machine. We assume that the self variable refers to a class whose behavior is described in the state machine. In each iteration of the loop, t will refer to a transition. The second line is a conditional statement: the third line will be executed only if the transition contains a trigger. In our example, the transition from the initial state to the Red state does not contain a trigger. The third line will produce the output of the script. The text that is contained between the bar symbols will be copied into the output as such. The other expressions will be evaluated and their result added to the script output. If t is a transition, t.source.name refers to the name of its source state and t.trigger.signal.name to the name of the signal that triggers the transition. The result of applying the script on the model is also shown in the figure.

## II. Rialto

The Rialto [2], [3] language (formerly called SMDL) is a high-level system design language that aims to capture different computational models through scheduling policies that can be assigned to different code blocks of a program. We consider the different diagrams in UML to be models of computation, some of which we show how to represent in this article. We can translate many different UML modeling elements into Rialto state blocks, the semantical differences being captured by the scheduling policies. We have defined a scheduling policy called rtc that executes the run-to-completion algorithm, which is the underlying semantics of UML statecharts. Other policies are e.g. interleaving and step.

Orthogonal regions in statecharts and active objects in an object collaboration diagram are two different types of concurrency. We represent both using the Rialto par statement; however, by applying the rtc scheduling policy to state blocks that represent statecharts, and the interleaving policy on blocks representing a collaboration for example, we capture the correct semantics.

The language has formal semantics that allows for programs to be verified; the semantics is for example easily translatable into B machines that can be verified [4]. The semantics of a Rialto program can also be translated into automata, that can be reduced. From the reduced automata representation, we can generate target language code like C++, assembly or a hardware description language.

## III. Rialto support in SMW

We have created an SMW extension that converts behavioral UML diagrams into specifications in the Rialto language. The short term objective is to explore and validate the Rialto approach. One can easily add new languages or models of computation to the SMW through metamodeling. Rialto aims to support many heterogenous computational models, as well as easy addition of new ones. Together SMW and Rialto make a good pair for exploration of different modeling paradigms. We can rapidly create a new language profile for SMW, after which we should be able to gain access to Rialto's code generators into different target languages.

The long term objective is to make Rialto accessible to the average UML practitioner. The designer just needs to create standard UML diagrams that are then translated automatically into Rialto. It is not necessary to use the SMW editor to create the models since the scripts

Proceedings of the Third International Conference on Application of Concurrency to System Design (ACSD'03)
0-7695-1887-7/03 $17.00 © 2003 IEEE
IEEE COMPUTER SOCIETY

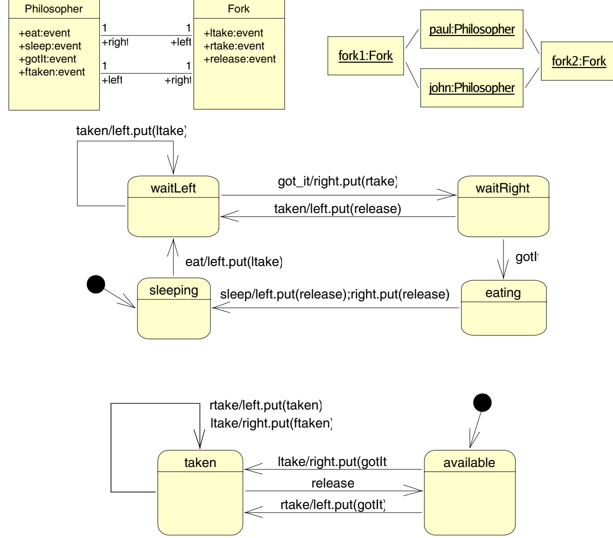



Fig. 2. Dining Philosophers in UML

can import models from any standard-compliant UML tool that can generate XMI files.

## IV. EXAMPLE: DINING PHILOSOPHERS

We will now show as an example, a simplified model of the dining philosophers problem. Figure 2 shows a UML model created using the SMW UML editor. The upper diagram shows a class diagram with the two classes, Philosopher and Fork. Both classes have behavior modeled using statecharts, the behavior of the Fork is shown in the machine in the middle, while the Philosopher machine is depicted at the bottom. Finally, we have instantiations of the classes in the collaboration diagram in the upper right diagram, where we have connected links, instantiating the associations from the class diagram.

The program listing below, shows a part of the Rialto code generated by the script from the model (the code for the Philosophers is left out). First we have the behavior of the Fork and Philosopher as stateTypes, which can be instantiated into state blocks. They are internally scheduled by the rtc policy since they represent statecharts. Then we have the block dinner, which is the collaboration where we connect queues to the Fork and Philosopher instances. The dinner block is scheduled using the interleaving policy, which models active objects running in parallel.

```
1  Fork: stateType(q,left,right)
2     policy rtc(q); event ltake; event rtake; event release;
3  begin
4     available: state
5        trap q.ltake do [right.put(gotIt); goto(taken)]
6        trap q.rtake do [left.put(gotIt); goto(taken)]
7     end
8     taken: state
9        trap q.release do goto(available)
10       trap q.ltake do right.put(ftaken)
11       trap q.rtake do left.put(ftaken)
12    end
13 end
14 ...
15 dinner: state
16    policy interleaving; queue fork1q; queue fork2q;
17    queue johnq; queue paulq;
18 begin
19    par( fork1, fork2, john, paul )
20    fork1: Fork(fork1q,johnq,paulq)
21    fork2: Fork(fork2q,paul1,john1)
22    john: Philosopher(johnq,fork1q,fork2q)
23    paul: Philosopher(paulq,fork2q,fork1q)
24 end
```

We have two approaches for translating the Rialto code into executable code. The simpler approach is based on a virtual machine. There is a small C++ library that implements the operational semantics of each Rialto statement. We can link the code generated by the Rialto compiler against this library in order to obtain a self-contained executable program. The generated code is rather simple: it just instantiates a C++ object for each statement in the Rialto program. This approach is definitely not efficient, but it is well suited for animation purposes, i.e. we can compile and run the code, and observe its execution within the SMW tool.

The second approach for code generation is based on a series of optimization steps that achieve more compact code. The Rialto program is first translated into finite state machines, which can often be reduced using S-Graphs as in the POLIS approach [1]. The reduced format can then easily be translated into different target programming or hardware description languages. Below we show the VHDL code generated for the fork1 object (no reductions were gained using S-Graphs in this toy example). Each object is translated into a VHDL process, which are executed in parallel.

```
1  fork1: process(clk,reset)
2     variable state = StateType := available;
3     variable uml_event : EventType;
4  begin
5     if reset = '0' then state := available;
6     elsif clk'event and clk = '1' then
7        fork1q_get_event( uml_event );
8        if state = available then
9           if uml_event=ltake then
10             paulq_put_event( gotIt );
11             state := taken;
12          elsif uml_event=rtake then
13             johnq_put_event( gotIt );
14             state := taken;
15          endif;
16       else
17          if uml_event=release then
18             state := available;
19          elsif uml_event=ltake then
20             paulq_put_event(ftaken)
21          elseif uml_event=rtake then
22             john_q_put_event(ftaken)
23          end if;
24       end if;
25    end if
26 end process fork1;
```

## V. CONCLUSIONS

Rialto is a behavioral language that supports multiple models of computation. Rialto specifications can be animated and compiled into optimized C++ or VHDL code. Rialto is designed as an intermediate language and, in most cases, specifications will be generated automatically from another modeling language such as UML. In this short article we have shown how the Rialto profile for the SMW tool allows the UML designer to use the Rialto animation, optimization and code generation features transparently.

## REFERENCES


[1] Felice Balarin et al. *Hardware-Software Co-Design of Embedded Systems*. Kluwer Academic Publishers, 1997.

[2] Dag Björklund and Johan Lilius. From UML behavioral models to efficient synthesizable VHDL. In *Proceedings of the 20th IEEE Norchip Conference*, november 2002.

[3] Dag Björklund and Johan Lilius. A language for multiple models of computation. In *Symposium on Hardware/Software Codesign 2002*. ACM, 2002.

[4] Dag Björklund and Johan Lilius. Generating B specifications from programs in a language with Z based formal semantics. submitted paper, 2003.

[5] OMG. OMG XML metadata interchange (XMI) specification. OMG Document formal/00-11-02. Available at www.omg.org.

[6] Ivan Porres. A toolkit for manipulating UML models. Technical Report 441, Turku Centre for Computer Science, 2001. Available at www.tucs.fi .



Proceedings of the Third International Conference on Application of Concurrency to System Design (ACSD'03)
0-7695-1887-7/03 $17.00 © 2003 IEEE



IEEE COMPUTER SOCIETY