A Unified Approach to Code Generation from Behavioral Diagrams

Dag Björklund, Johan Lilius and Ivan Porres
Turku Centre for Computer Science (TUCS)
Department of Computer Science, Åbo Akademi University
Lemminkäisenkatu 14 A, FIN-20520
Turku, Finland

Abstract

In this paper we show how to use Rialto, an intermediate behavioral language, to capture the semantics of UML behavioral diagrams. The Rialto language has a formal semantics given as structural operational rules and it supports semantic variations. It can be used to uniformly describe the behavior of a combination of several diagrams and as a bridge from UML models to animation and production code. We show two different approaches to code generation from UML behavioral models that are suitable for embedded systems.

1 Introduction

The Unified Modeling Language (UML) [OMGa] can be used to model the architecture and behavior of any kind of software project. This is due to the fact that UML provides many different diagrams or views of a system: class, component and deployment diagrams focus on different aspects of the structure of a system while the behavioral diagrams such as use case, statechart, activity and interaction diagrams focus on its dynamics.

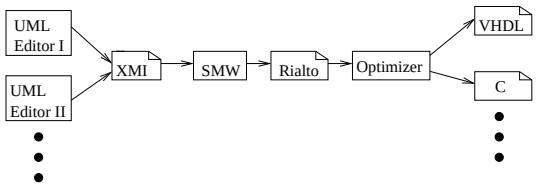
All the behavioral diagrams except use case diagrams are closely related. We can convert a collaboration diagram into a sequence diagram and vice versa. Statecharts are used as the semantic foundation of the activity diagrams and it is possible to represent an execution (a trace) of a statechart or an activity diagram as a sequence or collaboration diagram. However, at the formal level, we consider that the UML standard lacks a consistent and unified description of the dynamics of a system specified using the previous diagrams.

Many authors are working towards a formal semantics for UML [KEE]. Formal semantics give an unambiguous interpretation to UML using a mathematical formalism. Some of the recent works in this direction are the semantics for statecharts e.g. [Kus01] and [vdB01] and activity diagrams [EW01]. Although these articles are sound and inspiring, they do not study the semantics of multiple diagrams combined.

In this paper, we show how we can formalize the behavior of UML diagrams, including the combined behavior of several diagrams, using Rialto. Rialto is an intermediate language that can be used to describe multiple models of computation. A model of computation is a domain specific, often intuitive, understanding of how the computations in that domain are done: it encompasses the designer's notion of physical processes, or as Edward A. Lee puts it, the "laws of physics" that govern component interactions. UML statecharts (a chapter in [OMGa]) is an example of an asynchronous model of computation while languages like ESTEREL [BG92] or Harel's Statecharts [HN96] have semantics based on the synchronicity hypothesis (a variant of the synchronous computational model).

We have had as our starting point the observation that many language constructs are common to different languages, but their semantics differ due to the computational model. E.g. parallelism in UML

has a very different semantics from parallelism in ESTEREL. In [Lee02], Lee touches the subject of decoupling an abstract syntax from the model of computation, i.e. he suggests that a language, or a set of languages, with a given abstract syntax, can be used to model very different things depending on the semantics and the model of computation connected to the syntax. In the context of UML, this phenomenon can be observed e.g. with statecharts and activity diagrams. An activity diagram can be interpreted as a statechart where all computation is done in state activities and the transitions are triggered by completion events. Therefore, we can say that activity diagrams have data flow as their underlying model of computation. In Rialto, we define different scheduling policies for different computational models. The policies define atomic execution steps and hence we can for instance model syncronicity in way similar to that of SystemC. It is this interaction between language constructs and models of computation that is the focus of our research and our motivation to introduce Rialto as a language to describe the behavior of UML.



We are also applying this new insight on the combined behavior of UML models to construct tools to animate the models and generate optimized code. Model animation enables a designer to validate a model before the design and implementation has been completed. Rialto can be used as an execution engine for UML models, and is hence also related to the work on executabel UML [MB02]. Rialto can also be used to generate code for different target languages, including C and VHDL.

The translation from UML models to Rialto is performed using the SMW toolkit [Por02] as shown in the figure. The SMW toolkit can be used to transform and extract information from UML models. SMW can read UML models created by any XMI-compliant UML editor. We have automated the process of converting UML models created by different UML editors into Rialto and then into the target language using SMW scripts.

This article is a thoroughly revised and improved version of [BLP01]. We have extended the scope of the paper from UML statecharts to other UML behavioral diagrams, providing a unified view of the dynamics of UML models. This article is divided into three parts: We start by introducing Rialto, our intermediate language, then we explain how we represent UML statecharts, activity diagrams and collaboration diagrams in this language. Finally, we close the paper describing our strategy for code generation.

2 The Rialto Intermediate Language

Rialto is a state-oriented description language with formal semantics. It can be used as a stable platform for interfacing UML behavioral models with tools for code synthesis, animation, verification etc. We do not expect the average UML practitioner to use Rialto. Instead, we intend it to be both a tool to study the generation of efficient code and a tool for UML scholars to discuss different aspects of UML behavioral semantics. The Rialto language is more abstract than usual programming languages since it supports concepts like traps (high-level transitions), suspension and resuming of threads, and event queues. The language has been specifically designed to describe the behavior of modeling languages and it can be used to combine multiple heterogeneous models of computation [BL02b].

In this section we shortly present some syntactic elements of the Rialto, give a semantics for the statements in terms of structural operational rules and show how we can use different scheduling policies to give different meaning to the syntactic elements.

2.1 Syntax

Every statement in a Rialto program has a unique label, which can e.g. correspond to the name of some UML element, or it can just be a unique name given by the precompiler of the language. The `state`

block, is the key abstraction in the language. It can represent different entities in different domains, it also represents a scope that can be assigned a scheduling policy, as will be described later. In the language we have included some concepts that are common to different models of computation, but can have different semantics depending on different computational models. We will now give a short description of some of the concepts and show how they are represented in the language.

State State is explicit in statecharts for example. In Rialto a state can be represented by a `state` block. As in UML statecharts, Rialto `state`s can contain substates, which can also be orthogonal. A `state` block is declared as follows:

```
state
   # declarations, traps and policy
begin
   # substates, sequential code
end
```

If we plan to use the same state block in more than one state we can define a **stateType** to avoid code duplication. A `stateType` is declared as follows:

```
typename: stateType( ([in|out] port)* )
    # state definition
end
```

Where the "state definition" is a normal `state` block and the ports are queues used by the `stateType` to communicate with other entities. The ports can be used as normal queues inside the block. A `stateType` is instantiated as:
`label: typename( q1 (, q2 )* )`.
The ports declared in the `stateType` will be substituted for the queue instances `q1`, `q2` etc. This means that the following two pieces of code are equivalent:

```
 1  myType: stateType( in myPort )
 2    trap myPort.e1 do null
 3  end
 4
 5  state: main
 6    queue q1;
 7  begin
 8    S1: myType( q1 )
 9    S2: myType( q1 )
10  end
```

```
 1  state: main
 2    queue q1;
 3  begin
 4    S1: state
 5      trap q1.e1 do null
 6    end
 7    S2: state
 8      trap q1.e1 do null
 9    end
10  end
```

Interrupts and transitions An interrupt is an event of high priority. In our language, a `trap` statement is used to monitor interrupts. Interrupts correspond to `trap` in ESTEREL and transitions going upwards in the state hierarchy in both UML and Harel's Statecharts. The `goto` statement is used to do transitions from a state to another. A `trap` that monitors an event e on a queue q and does a transition to a state A is declared as follows: `trap q.e do goto(A)`.

Concurrency Concurrency means that several statements are executed in parallel. In our language, concurrency is indicated using the `par` statement, where the arguments are the labels of, usually `state` statements, which should be run in parallel. However, the parallelism is interpreted differently depending on the execution policy in effect in the current scope.

Communication policy The communication policy states how different modules of the system communicate with each other. The communication mechanism we use with UML statecharts is a simple fifo queue. In state diagrams, an event is an occurrence that may trigger a state transition. In UML statecharts, there is an implicit global event queue; whereas, in our language several queues can be declared and the scope of a queue declaration is the `state` block. The notation in our language for checking for the presence of an event on a queue is `q1.e1`, where `q1` is a queue and `e1` is an event.

$$par \qquad \frac{l \in \alpha \wedge \mathcal{L}[l] = \texttt{par}(l_1, l_2, \cdots, l_n)}{\langle \alpha, q \rangle \xrightarrow{l:par(l_1,l_2,\ldots,l_n)} \langle \alpha - \{l\} \bigcup_{i=1}^{n} \{l_i\}, q \rangle}$$

$$goto \qquad \frac{l \in \alpha \wedge \mathcal{L}[l] = \texttt{goto}(l_1, \cdots, l_n)}{\langle \alpha \rangle \xrightarrow{goto(l_1,\cdots,l_n)} \langle \alpha - subtree(l, l_1, \cdots, l_n) \cup \{l_1, \cdots, l_n\} \cup entry(dom(\downarrow^* \rhd l_1, \cdots, _n)) \rangle}$$

$$state \qquad \frac{l \in \alpha \wedge \mathcal{L}[l] = \texttt{state} \wedge l \succ l_1 \wedge entry(l) = S}{\langle \alpha, q \rangle \xrightarrow{l:state} \langle \alpha - \{l\} \cup \{l_1\} \cup S, q \rangle}$$

Table 1: A subset of the operational rules of the language

2.2 Operational Semantics

To define the operational semantics of the language we need to formalize the structure of a Rialto program. The labels of the statements act as values for program counters. The set of labels Σ is organized in a tree structure by the *child* relation $\downarrow$ (The tuple $\langle \Sigma, \downarrow \rangle$ forms a label-tree). This reflects the scoping of the program. We can take the closure $\downarrow^*$ of $\downarrow$: let l_1 and l_2 be labels; then $l_1 \downarrow^* l_2$ means l_1 is a *descendant* of l_2 or in other words: l_1 is inside the scope of l_2. The *successor* relation $\succ \subset \Sigma \times \Sigma$ orders the labels sequentially, If $l_1, l_2 \in \Sigma$ and $l_1 \succ l_2$ then l_2 succeeds l_1 in the program. $\mathcal{L} : \Sigma \rightarrow statement$ is a total function that maps each label to its statement.

We can now identify a program by the tuple: $\langle \Sigma, \downarrow, \succ, \mathcal{L} \rangle$ The state configuration of a program is represented by a tuple $\langle \alpha, suspend, q \rangle$ where:

- $\alpha \subseteq \Sigma$ is the set of active labels.
- $suspend\ \Sigma \rightarrow \Sigma$ is a partial function. If $suspend(l) = l'$ then l is a suspended state and l' is a substate of l that was active when l was suspended.
- q is the set of event queues.

The active set α represents the current values of the program counters. We sometimes use the abbreviation σ for the state tuple. A subset of the operational rules that update the state configuration is given in Table 1. The rules determine how the statements update the state of the machine [1].

There are in total 18 rules that define the semantics of the language. In this paper, we only show a few due to the space limitation. For a complete description of Rialto please refer to [Bjö01].

The par statement A `par` statement can act when its label is active; it adds all of its arguments to the active set. Note that the par statement just creates new threads. It is the job of the policy to decide how to schedule the threads.

Goto The `goto` statement is used to switch from an active state to another state. The only way to escape a state block, is by using the `goto` statement. It can take many labels as parameters, which allows it to be used to model fork transitions. A `goto` is enabled when its label is active; it removes the subtree containing the source and target labels from the active set, and activates the target labels (belonging to state statements) along with any labels of `trap` statements or entry actions that are ancestors of the target labels.

State A `state` statement can act when its label is active. It will remove its label from the active set and add any `trap` and entry actions it may contain along with its successor to the active set. It marks the beginning of a state block.

An *execution engine* picks labels from the set of active labels and then executes the rule corresponding to the statement of that label. An *execution policy* decides on the order in which labels are picked from the active list. We describe the issue of policies more in depth in section 2.3.

[1] We use the domain/range restriction operators $\lhd$ / $\rhd$ as defined in Z to operate on the relations: Let S be a set and R a relation, then $R \lhd S$ is a restriction of R, where every element in the domain of R is a member of the set S.

2.3 Scheduling Semantics

In the previous section, we presented the concrete syntax and the execution semantics of the language. However, there is still freedom in how the active labels are scheduled. To specify this aspect, we introduce an *execution policy* for each different model of computation.

An execution policy connected to a `state` block, schedules the substates according to a given algorithm. A program is executed by repeatedly running the policy of the topmost state in the hierarchy. The topmost policy will then schedule the states down in the hierarchy, which can have different policies assigned to them. The entity that calls the scheduling policy of the top-level state can be thought of as a global clock in the system.

To define a scheduling policy we need some helper functions. A label that belongs to a simple statement is *enabled* iff the premiss of the rule that corresponds to the statement holds; if a label belongs to a `state` statement, it is enabled iff it is active or has descendants that are enabled. A label can also become blocked by the scheduler, and is then not enabled. The boolean function $IS\_ENABLED(l, \sigma, \beta)$ returns true if label l is enabled when the program is in state σ and the labels in set β are blocked: ($premiss(l, \sigma)$ returns true if the premiss of the rule for l holds in state σ)

IS_ENABLED(l, σ, β)
1 **if** $\mathcal{L}[l] \neq state$
2 **return** $premiss(l, \sigma) \wedge l \notin \beta$
3 **else**
4 **return** $ENABLED(l, \sigma, \beta) \neq \emptyset$

The $ENABLED(l, \sigma, \beta)$ function returns the set of enabled labels that are descendants of l:

ENABLED(l, σ, β)
1 **return** $\{l_i : label \mid IS\_ENABLED(l_i, \sigma, \beta) \wedge l \downarrow^* l_i\}$

We will use these functions in the actual definition of the scheduling policies. A scheduling policy function accepts as parameters the current state of the program σ and the blocked set β and returns the next state of the program. As an example, the following algorithm implements a *step* scheduling policy that executes all enabled statements in the same step. Note that $self$ refers to the `state` block whose RUN function was called.

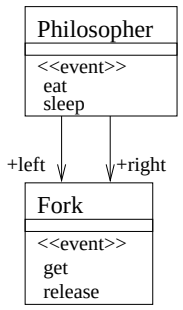
RUN(σ, β)
1 $\rho \leftarrow ENABLED(self, \sigma, \beta)$
2 **for each** l_i **in** ρ **do**
3 $\sigma' \leftarrow l_i.RUN(\sigma, \emptyset)$
4 **return** σ'

In the next section we will show how different scheduling policies can model the behavior in UML behavioral diagrams.

3 Representing UML models in Rialto

In this section we describe how we translate a UML model into Rialto code. This transformation has to deal with the fact that UML is a family of languages, both at the semantic and the syntactic level. It is possible to create different translators for different semantic interpretations of UML. We currently support statecharts, activity diagrams and collaboration diagrams.

We show as an example the translation of a collaboration of objects whose behavior is modeled using




```
1   Philosopher: stateType(out left,out right)
2     event eat, sleep;
3   begin
4     # behavior of Philosopher
5   end
6
7   Fork: stateType( in phil )
8     event get, release;
9   begin
10    # behavior of Fork
11  end
```

Figure 1: A UML class diagram and Rialto code

statecharts or activity diagrams. Figure 1 shows a UML class diagram with two classes `Philosopher` and `Fork` along with the Rialto translation consisting of two corresponding `stateType`s. There are two associations connecting the two classes, indicating that a `Philosopher` has a left and a right fork. This is reflected in the Rialto code by the two outgoing ports of the `Philosopher` type and the incoming one in the `Fork` type (we only use one incoming event queue for all connections). The behaviors of the two classes would be modeled using statecharts and the Rialto translations of the statecharts would appear inside the `stateType`s.

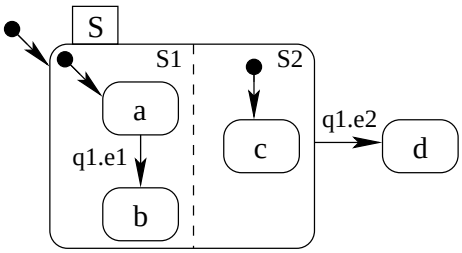
3.1 Statecharts

A **state** in a statechart is represented by a `state` block; hierarchical compound states are naturally written as nested `state` blocks. The first `state` block in a compound state is the **initial state**. **Transitions** can be represented using the `trap` statements for monitoring an event or value of a guard, and the `goto` statement to take the transition. We have so far concentrated on control in the Rialto language, and have ignored data, thus the statechart translations are also limited to binary events, i.e. they can not have values. Also **orthogonal regions** are represented by `state` blocks, with a `par` statement specifying that the states (regions) are to be run orthogonally. **Fork** pseudostates are simply achieved by providing the `goto` statement with several state labels as parameters (`goto(a,b)` does a transition to the two states `a` and `b`, which are run orthogonally). **History pseudostates** are represented by `suspend` statements, that store the state of a block until it is re-entered. We have not yet dealt with sync pseudostates.

The semantics of UML statecharts is based on a run-to-completion (RTC) step. The RTC algorithm fires transitions in the statechart based on the event on the queue, and dispatches it at the end of the RTC step. In addition to the translations described above, we need to provide a scheduling policy that runs the code according to the RTC algorithm. A state executing the RTC policy runs until no more runnable labels exists in its scope. The algorithm is shown below. It first runs all the enabled statements, and then checks if there appeared any new enabled labels, adding any labels that are not ancestors of the original enabled labels to the blocked set (line 5). If there are enabled non-blocked labels, it runs again. By this blocking, we ensure that when a state transition is taken during a RTC step, the newly activated state will not be executed during that same step.

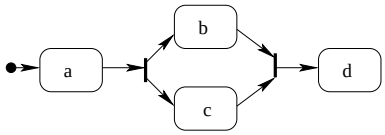
RUN(σ, β)
1 $\rho \leftarrow ENABLED(self, \sigma, \beta)$
2 **for each** l_i **in** ρ **do**
3 $\quad \sigma \leftarrow l_i.RUN(\sigma, \emptyset)$
4 $\rho' \leftarrow ENABLED(\sigma', self, \rho)$
5 $\beta' \leftarrow \{l_j \in \rho' \mid \mathcal{L}[l_j] = state \wedge l_j \notin (\ ran(\rho \lhd \downarrow^*\ ))\}$
6 **if** $\rho' - \beta' \neq \emptyset$ **then**
7 $\quad$ **return** $self.RUN(\sigma', \beta')$
8 $dequeue(q)$;
9 **return** σ

In Figure 2 a) we show a simple statechart with a composite state divided into two orthogonal regions.




```
1   top: state
2     policy rtc;
3     queue q1; event e1, e2;
4     S: state
5       trap q1.e2 do goto(d)
6     begin
7       par( S1, S2 )
8       S1: state
9         a: state
10          trap q1.e1 do goto(b)
11        end
12        b: state end
13      end
14      S2: state
15        c: state end
16      end
17    end
18    d: state end
19  end
```

Figure 2: a) An example UML statechart b) the corresponding Rialto code generated by the tool




```
1   activity: state
2     policy step; queue q1;
3   begin
4     A: state begin # code block for A
5       goto(B, C)
6     end
7     B: state begin # code block for B
8       goto(join)
9     end
10    C: state begin # code block for C
11      goto(join)
12    end
13    join: state
14      trap !B and !C do goto(D)
15    end
16   D: state end
17  end
```

Figure 3: A simple Activity Diagram

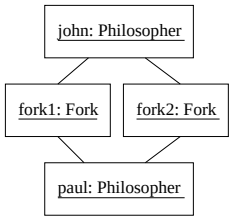
The corresponding Rialto code is shown in Figure 2 b). This code has been generated by a tool as described in Section 4. The scheduling policy for the top-level `state` block in the code is set to RTC in line 2, and the queue `q1`, and the events are declared in line 3. The `S` state listens to event `e2` on queue `q1` and fires the transition to state `d` if it is present (line 5). The `par` statement in line 7 activates the regions `S1` and `S2` orthogonally. If states `a` and `c` are active and the current event is $e1$, both transitions will fire in the same step.

3.2 Activity Diagrams

The translation of activity diagrams follows a similar schema. The **activity states** are represented by `state` blocks, **forks** are dealt with as in the statecharts; however, **join** pseudostates usually need to be explicitly modeled using a `state` block for synchronizing the two incoming transitions.

The RTC scheduling policy is not suitable for activity diagrams. Instead, we use the *step* policy introduced in the previous section, which for each invocation runs all the enabled labels and returns.

As an example, we show a simple activity diagram and its translation into Rialto in Figure 3. After the activity in `a` is completed, we take a forked transition to `b` and `c` (line 5). This transition (`goto(b,c)`) activates both `b` and `c` that is they will run concurrently without using the `par` statement. When the `b` and `c` states are finished, they take transitions to the join state, in which the control remains until both `b` and `c` become inactive (line 14).




```
1  dinner: state
2    policy interleaving; queue q1, q2;
3  begin
4    par(john,paul,fork1,fork2)
5    john: Philosopher(q2,q1);
6    paul: Philosopher(q1,q2);
7    fork1: Fork(q1);
8    fork2: Fork(q2);
9  end
```

Figure 4: Objects and links instantiated

3.3 Collaboration Diagrams

Statecharts and activity diagrams model the behavior of objects, while collaborations show an interaction between objects. We can use Rialto to capture the behavior of systems modeled using collaboration diagrams, where the objects can have behavior described by either statecharts or activity diagrams.

The objects are represented by `state` blocks (or instantiations of `stateType`s), which are defined to run concurrently using the `par` statement. The objects can communicate through queues. The default scheduling policy for a collaboration diagram is interleaving. This means that each object in the collaboration runs in its own task and is scheduled nondeterministically. The interleaving policy picks a label from the enabled set nondeterministically under the fairness assumption (The $:\in$ in line 2 in the algorithm below denotes nondeterministic assignment).

RUN(σ, β)
1 $\rho \leftarrow ENABLED(l, \sigma, \beta)$
2 $l_i \ :\in \ \rho$
3 **return** $l_i.RUN(\sigma, \emptyset)$

Alternatively, the step policy can be used, which allows for active objects to execute at the same time during one step.

The behavior of an object can be described using a statechart or an activity diagram. The internal behavior of that object will then be RTC or step. As an example, we can model a collaboration of two philosophers and two forks as in Figure 4. The ports are connected to event queues when the types are instantiated. The associations from the class diagram are also instantiated as links connecting the objects. In the corresponding Rialto code, we show how the `stateType`s are instantiated. The `par` statement activates all the objects in parallel. Two queues are declared in the top-level `state` block `dinner`. They are connected to the ports of the objects. This actually means that queue `q1` is the event queue of the statechart of `fork1` while `q2` is the queue of the `fork2` object. The Rialto program is run by repeatedly calling the top-level scheduling policy. In this case it is the `interleaving` policy of the `dinner` state. That policy will pick one of the parallel blocks and call its scheduling policy, which in this model will be a `rtc` policy.

3.4 Automatic UML to Rialto Translation

The translation from UML models to Rialto is performed with the help of the SMW toolkit [Por02]. This toolkit is based on the Python programming language [Pyt] and it can be used to transform and extract information from UML models. It can read XMI [OMGb] files created by any XMI-compliant UML editor. Once a model is loaded, we can navigate it using queries like in OCL and transform it using Python imperative statements.

In the code below, we use several functions for text manipulation: The vertical bars add a string to the final Rialto code. E.g. the statement `|2+2=|2+2` will add the string `"2+2=4"` to the output code. The indentation of the code is controlled using `|->>|` and `|<<-|`. The `*` operator on a text list generates a string with the concatenation of a list using the second operand as a separator. For example

`["a","b","c"] * " or "` returns `a or b or c`.

The following SMW script generates Rialto code for a UML StateMachine model element. We assume that the reader is familiar with the OCL language and the UML metamodel for statecharts. The function accepts as parameters an object representing a UML statechart model element, the name of the output Rialto code block, and the name of the input queue to be used. The body of the function is quite simple, since it merely defines the queue, the scheduling policy and iterates through all the subvertices of the top state. It is necessary to place the initial state as the first state of the Rialto block. This is accomplished by sorting the subvertex collection as done by sortByIntialState.

```
def StateChart2Rialto(sc,name,queue):
    name |: state|
    |->>||policy rtc; queue | queue |;||<<-|
    |begin||->>|
    for s in sortByInitialState(sc.top.subvertex):
        StateVertex2Rialto(s,queue)
    |<<-||end|
```

The following function converts a UML state. We use the `trap` statement to model the high-level transitions that exit the state. The function is called recursively in the case of composite states that contain other states. In concurrent composite states, the subvertices are orthogonal regions that run in parallel using the `par` statement.

```
def StateVertex2Rialto(s,queue):
    s.name|: state|
    |->>|
    for t in s.outgoing:
        |trap | queue |.| t.trigger.getDescription() | do | Transition2Rialto(t)
    |<<-|
    if s.oclIsKindOf(CompositeState):
        if s.subvertex.size()!=0:
        |begin||->>|
        if s.isConcurrent:
          |par(| s.subvertex.name*"," |)|
        for subState in sortByInitialState(s.subvertex):
           StateVertex2Rialto(subState,queue)
        |<<-||end|
```

We omit the implementation of the helper functions due to space constraints. The full script is available from the authors.

4 Animation and Code generation

One of the objectives of the Rialto language is to facilitate the animation, code generation and verification of UML models containing complex behavior. Using Rialto and our code generation approach, we can model the system using object-oriented techniques and the UML, while being able to generate efficient code in target languages that may not be object-oriented. This makes our method useful for designing embedded systems that can run on simple processors with limited memory and limited support for high-level programming languages. We are also studying how to generate VHDL hardware descriptions from Rialto.

We plan to develop animation capabilities for the SMW UML editor, using Rialto as the execution engine. The user can execute the model, emit events etc. from the editor while the current state configuration is shown in the UML model. This will help the designer to gain a higher confidence in the UML model in an early stage of development.

Once a model is translated into Rialto, we can proceed to the actual code generation. We currently have two strategies for generating code from Rialto. The simplest one, which we have implemented in C++, is an execution engine approach. The code generated using this approach is not optimized,

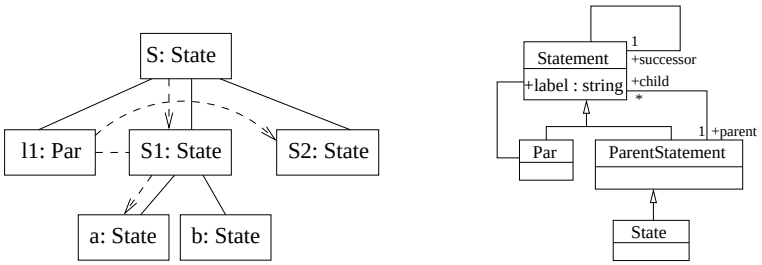


Figure 5: a) A part of the label-tree for the code in Figure 2 b) A metamodel for Rialto

but reflects the structure of the Rialto model, i.e. the generated code has one object for each Rialto statement. Therefore, this approach is suitable for animation. The second strategy for code generation involves translating the model into finite state machines which are reduced using S-Graphs.

4.1 Execution Engine

In this approach we implement a small library containing different functions implementing the operational rule for each statement in Rialto, functions for each scheduling policy as well as data structures for the active set and queues. This library forms an execution engine for the language.

In order to use this library we should store the Rialto model in a certain way. We show in Figure 5.a a part of the label-tree data structure for the model in Figure 2, which can be seen as an instantiation of the metamodel of Rialto. A part of a metamodel of Rialto is depicted in Figure 5.b.

The C++ code generation is straightforward. A parser reads a Rialto file and builds up the label-tree $\langle \Sigma, \uparrow \rangle$ for the program (See Section 2.2 for an explanation of the label-tree). Then, we write down the data structure for the tree in C++ and link it against the execution engine library to obtain an executable program that runs on the target machine. Each node in the label-tree is represented as a C++ object. The implementation of the different schedulers can use the services of the operating system such as threads. If a target platform lacks an operating system, or it does not support threads, we can implement the scheduler ourselves following e.g. the `interleaving` policy. This scheduler will execute the objects in a round robin fashion.

4.2 Optimized Code Generation

The optimized code generation process translates the Rialto program into a FSM like representation, which can be reduced in a way similar to that of the POLIS approach [B$^+$97]. This method is based on Software Graphs or S-Graphs and it is described in more detail in [BLP01] and [BL02a]. An S-Graph is a directed acyclic graph used to describe a decision tree with assignments, which can be reduced. From the obtained optimized low-level representation, we can easily generate target language code in e.g. C or a hardware description language such as VHDL. Compact assembly code could also be synthesized directly for targets lacking higher level language compilers.

The translation of a Rialto model into optimized code proceeds in five steps: Translation (flattening) of the Rialto code to a simple finite state machine, translation of the FSM into an S-graph, optimization of the S-graph, translation of the S-graph into a target language and, finally, compilation into machine code or synthesis of a hardware netlist.

We have achieved up to 30 percent code footprint reduction when generating C code and compiling it into object code. We expect also good results in area reduction when synthesizing hardware from VHDL code generated from Rialto.

5 Conclusions

We have shown in this paper how we can use Rialto to describe the combined behavior of different UML diagrams. Rialto can be converted into program code with the purpose of model animation or as a final production code to implement embedded systems with limited computational resources.

The UML behavior diagrams include many concepts such as actions, events, states, etc. that are not present in most popular programming languages, like C++ or Java. This means there is not a one-to-one mapping between a behavioral diagram and its implementation. Some model elements, like history states, can be implemented in many different ways; this clearly contrasts with class diagrams, that often can be easily implemented in a programming language supporting concepts like classes and objects, composition and inheritance. Rialto can be used as an intermediate language between models and code and supports semantic variations thanks to our two-phase approach for code generation.

An important decision in a code generation method is whether the programmer will be allowed to edit the produced code or not. We have opted to hide the final implementation from the programmer. This implies that the code does not need to be intelligible by a human programmer, and that it is not necessary to reverse engineer the code back into a UML model. However, this approach requires, in order to be practical, that the produced code is so efficient that the programmer does not need to tweak it by hand.

We are currently working on an extension of the SMW toolkit for early simulation and animation of UML models. This will allow the designer to validate and debug the models directly using a modeling tool instead of examining and debugging the generated program code.

Acknowledgments Dag Björklund gratefully acknowledges financial support for this work from the Nokia foundation.

References

[B+97] Felice Balarin et al. *Hardware-Software Co-Design of Embedded Systems.* Kluwer Academic Publishers, 1997.

[BG92] Gérard Berry and Georges Gonthier. The Esterel synchronous programming language: Design, semantics, implementation. *Science of Computer Programming*, 19(2), 1992.

[Bjö01] Dag Björklund. The SMDL statechart description language: Design, semantics and implementation. Master's thesis, Åbo Akademi University, 2001.

[BL02a] Dag Björklund and Johan Lilius. From UML behavioral models to efficient synthesizable VHDL. In *Proceedings of the 20th IEEE Norchip Conference*, november 2002.

[BL02b] Dag Björklund and Johan Lilius. A language for multiple models of computation. In *Symposium on Hardware/Software Codesign 2002*. ACM, 2002.

[BLP01] Dag Björklund, Johan Lilius, and Ivan Porres. Towards efficient code synthesis from statecharts. In Andy Evans, Robert France, Ana Moreira, and Bernhard Rumpe, editors, *pUML Workshop at UML2001*, october 2001.

[EW01] Rik Eshuis and Roel Wieringa. An execution algorithm for UML activity graphs. In Martin Gogolla and Cris Kobryn, editors, *UML2001 - The Unified Modeling Language. Modeling Languages, Concepts, and Tools*. Springer, 2001.

[HN96] D. Harel and A. Naamad. The statemate semantics of statecharts. *ACM Tran. of Software Engineering and Methodology*, 5(4) Oct 1996.

[KEE] S. Kent, A. Evans, and B. Rumpe (Eds.). UML semantics FAQ. Available at www.univ-pau.fr/OOPSLA99/samplewr99.pdf.

[Kus01] Sabine Kuske. A formal semantics of UML state machines based on structured graph transformations. In Martin Gogolla and Cris Kobryn, editors, *UML2002 - The Unified Modeling Language. Modeling Languages, Concepts, and Tools*. Springer, 2001.

[Lee02] Edward A. Lee. Embedded software. *Advances in Computers*, 56, 2002.

[MB02] Stephen J Mellor and Marc J Balcer. *Executable UML, A Foundation for Model-Driven Architecture*. Addison-Wesley, 2002.

[OMGa] OMG. OMG Unified Modeling Language Specification. Version 1.4, September 2001, available at www.omg.org.

[OMGb] OMG. OMG XML Metadata Interchange (XMI) Specification. Available at www.omg.org.

[Por02] Ivan Porres. A toolkit for manipulating UML models. Technical Report 441, Turku Centre for Computer Science, 2002.

[Pyt] The Python web site. *Internet: http://www.python.org*.

[vdB01] Michael von der Beeck. Formalization of UML-statecharts. In Martin Gogolla and Cris Kobryn, editors, *UML2002 - The Unified Modeling Language. Modeling Languages, Concepts, and Tools*. Springer, 2001.