From UML Behavioral Descriptions to Efficient Synthesizable VHDL

Dag Björklund and Johan Lilius

TUCS Turku Centre for Computer Science
Lemminkäisenkatu 14, FIN-20520 Turku, Finland
Åbo Akademi University, Department of Computer Science
{dbjorklu,jolilius}@abo.fi

Abstract: *Different approaches to high-level synthesis are currently being studied for different specification language - target language pairs. In the paper we describe a strategy for high-level synthesis that can be used for code generation from several specification languages into several target languages. We demonstrate the approach using VHDL synthesis from UML behavioral models as an example. The UML models are first translated into textual code in a language called SMDL. SMDL is a high level language for multiple models of computation that can be compiled into efficient target language code e.g. VHDL.*

1 INTRODUCTION

There are numerous languages and visual specification tools seeking to take system development to the next level of abstraction. Most languages provide simulation tools, many of them also have code generation of some quality to some target languages. Usually the concepts of the specification language are mapped as directly as possible to the target language, closing the gap between abstraction levels with some extra code. This approach makes the generated code somewhat readable, which is a requirement, since the code is often also quite inefficient forcing the designer to tweak it by hand.

We are designing a language called SMDL for interfacing different high-level models of computation with different target language implementations [3]. The language is compiled into a low level optimized representation, from which target language code like C or VHDL can be generated.(We should use the term code synthesis instead of code generation, since synthesis denotes an optimized translation process from spec to a target language). Out method also allows for direct synthesis of assembly code, or hardware netlists.

In this paper we will demonstrate the use of our approach for one particular purpose, namely that of VHDL generation from UML behavioral models. Many embedded systems designers are hoping the Unified Modeling Language (UML) [7] will become a comprehensive system-level modeling language. The 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 collaboration, statechart, activity and interaction diagrams focus on its dynamics.

Since it has become a widely used standard, tools for simulating and verifying UML models exist and more are to come. When going from UML to VHDL, also the existing tools for VHDL simulation can be used; however, in our methodology we feel that simulating the UML models directly would be preferable, since the generated code does not easily map back to the original model due to the optimizing synthesis process.

The translation from UML models to SMDL is performed using the aUML toolkit [9]. The aUML tool can be used to transform and extract information from UML models. The models are read from XMI [8] files created by any XMI-compliant UML editor. The process of converting UML models created by different UML editors into different target language code, e.g. VHDL or C, is illustrated in Figure 1.

Several people have worked on system-level synthesis into VHDL [5,6], but not many address UML. Furthermore, our approach is more generic and can be used with many system level specification languages into many target languages.

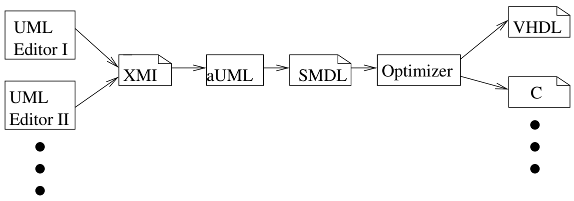


Figure 1: The code generation process

In section 2, we introduce the SMDL language, section 3 deals with UML to SMDL translation, section 5 covers the SMDL to VHDL compilation, and finally we give a conclusion.

2 THE SMDL LANGUAGE

SMDL is a language with formal semantics and high-level concepts such as states, queues and events. It can be used as a stable platform for interfacing system specifications with tools for code synthesis, animation, verification etc. We will briefly introduce the language in this section, refer to [2–4] for further information.

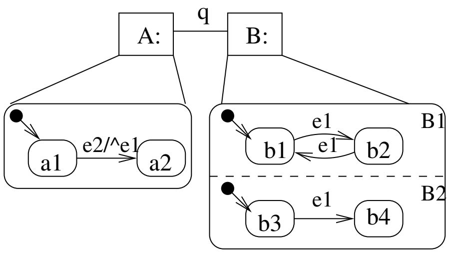
The main programming unit in the language is the `state` block. A `state` block can represent different model elements in UML diagrams e.g. a state in a statechart, an activity in an activity diagram, an active object in a collaboration diagram etc. State transitions are represented using the `goto` statement with a `if` statement checking for presence of events. The `par` statement is used to denote concurrency. Each statement in a program has a unique label. The state configuration of a running program is the set of active labels and the state of the event queues. An execution engine picks a label from the active set and executes the corresponding statement. The semantics of the statements are defined using *structural operational rules*. As an example, we show the rule for the `par` statement below.

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

Informally the rule states: if the label l belongs to the active set α, and l belongs to a `par` statement ($\mathcal{L}[l] = par(..)$), then the `par` statement can be executed, resulting in a new state configuration, where the parameters $l_1, l_2, \cdots, l_n$ have been added to the active set, while l was removed. Since there may be several labels active simultaneously we have introduced the notion of *scheduling policy*. A scheduling policy defines in which order the active labels are executed. Each `state` block can be assigned a different scheduling policy. In this way we can simulate different computation models. Some of the scheduling policies we have defined are e.g. the *interleaving* policy that picks one active label per cycle and runs it and the *rtc* policy that executes a run-to-completion step at each cycle. The run-to-completion algorithm is the underlying execution semantics of UML statecharts.

3 FROM UML TO SMDL

The translation from UML to SMDL is fairly straightforward due to the high-level concepts like queues, states etc. present in both languages. We will demonstrate the translation from UML statecharts and collaboration diagrams by a running example. Figure 2 a) shows a UML collaboration diagram containing two active objects `A` and `B`, with behavior modeled using statecharts. The statechart for the `B` object contains two orthogonal regions `B1` and `B2`. The UML model can be translated into the SMDL code in Figure 2 b). In this piece of code, all




```
1   sys:  state
2     event e1; queue q;
3     policy interleaving;
4     l1:  par(A,B)
5     A: state
6       queue qA; event e2;
7       policy rtc;
8       a1:  state
9         l2:  if qA.e1 then
10          l3:  emit(q.e1); l4:  goto(a2) endif;
11      l5:  endstate a1
12      a2:  state l6:  endstate a2
13    l7:  endstate A
14    B: state
15      policy rtc;
16      l8:  par(B1,B2)
17      B1:  state
18        b1:  state
19          l9:  if q.e1 then l10:  goto(b2) endif;
20        l11:  endstate b1
21        b2:  state
22          l12:  if q.e1 then l13:  goto(b1) endif;
23        l14:  endstate b2
24      l15:  endstate B1
25      B2:  state
26        b3:  state
27          l16:  if q.e1 then l17:  goto(b4) endif;
28        l18:  endstate b3
29        b4:  state l19:  endstate b4
30      l20:  endstate B2
31    l21:  endstate B
32  l22:  endstate sys
```

(a) (b)

Figure 2: A UML behavioral model and its SMDL translation

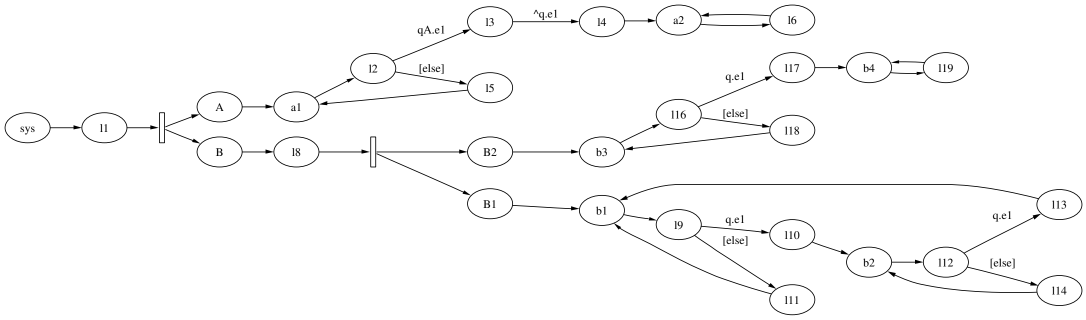


Figure 3: An activity diagram that illustrates the operational semantics of the program in Figure 2 b)

statements are given labels by hand in order to make it easier to demonstrate some issues. The objects (or threads) `A` and `B` are modeled as `state` blocks (lines 5 and 14) that are run concurrently (line 4). The interleaving policy is used to schedule the top-level state block (line 3). Since the behavior of the objects is modeled using statecharts, the `A` and `B` blocks are scheduled internally by rtc policies (lines 7 and 15). The state transitions in the statecharts are represented by the `if`, `goto` statements (e.g. line 19). Notice that we now are using two different scheduling policies i.e. we are using *heterogenous models of computation*.

4 FROM SMDL TO AUTOMATA

Most approaches for high-level synthesis try to map the high-level concepts as directly as possible to the target language concepts. In our approach, we go through yet a few intermediate steps, ending up with low-level finite state machine descriptions, that we can optimize, and from which we can easily generate code in different target languages. We first reduce the SMDL program to an automaton where the scheduling policy has been applied. This is illustrated in figures 3 to 5.

If we ignore the scheduling policies and only consider the operational semantics of the statements, we can illustrate the SMDL program in the example by the activity diagram in Figure 3. This diagram shows the state of the active set, as well as the event emissions etc. At first, the `sys` label of the top-level `state` statement is active. Running the statement simply activates the next label namely `l1`, belonging to the `par` statement. Running the `par` statement activates the `A` and `B` labels. The vertical bar denotes *splitting of control*, i.e. both statements after the bar are active. By removing trivial states and transitions, that only activate some label(s) and deactivate themselves, results in the diagram in Figure 4.

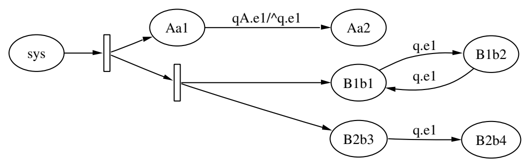


Figure 4: Trivial states removed

Two consecutive `state` labels like `A` and `a1` (where `A` is a superstate of `a1`) for example, have been truncated into the `A_a1` state. States belonging to `par` statements like `l1` have been removed etc. Applying the rtc policy for the statechart substates compiles away the orthogonality of the `B` statechart [1] by taking the cartesian product of the states. The initial states of the orthogonal regions in the original UML model are `b1` and `b3` or `B1b1` and `B2b3` in the activity diagram in Figure 4. We replace this initial state configuration by a state `B1b1_B2b3`.

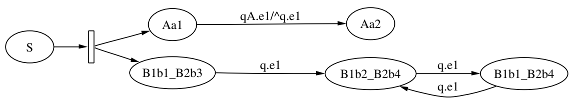


Figure 5: Scheduling policies applied

When the event `e1` is received, both regions take transitions, ending up in states `B1b2` and `B2b4`. Again we replace this state configuration with the state `B1b2_B2b4`. Carrying on, we obtain the diagram of Figure 5 with two parallel state machines (applying the interleaving policy preserves the parallelism). From here it is easy to proceed to code generation.

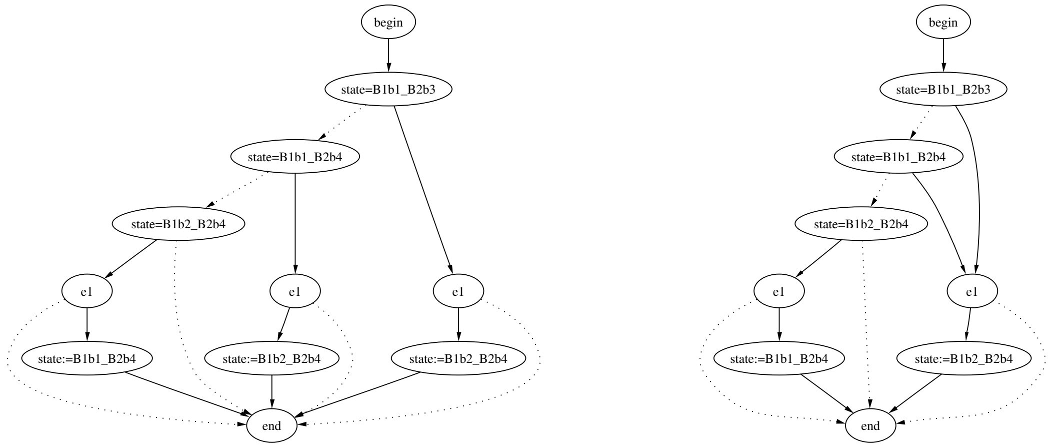
5 FROM AUTOMATA TO VHDL

The parallel FSM's from the previous sections will be translated into two VHDL processes implementing a state machine each. Before doing so, however, we will show that optimizations can be performed on the FSM's. The result will be a minimal low-level representation of the original model from which we can generate target language code, in this case VHDL. Often the VHDL synthesis tool could perform the same optimizations if we were to skip this step. We could actually proceed to netlist generation directly, instead of generating VHDL. We have also experimented with C code generation and have achieved significant object code footprint reductions using the optimization.

Our optimization process is similar to that of the POLIS approach [1] and it is based on Software Graphs or S-Graphs. An S-Graph is a directed acyclic graph used to describe a decision tree with assignments. The S-Graphs can be minimized, which allows us to generate code that is optimized for size, another property of the S-Graphs is that they are very well-suited for code-size and performance estimation, which is often important in embedded systems.

We will here demonstrate how the FSM in the lower thread in Figure 5 can be reduced using S-Graphs. An S-graph consists of a set of vertices `V` which contains four types of vertices: BEGIN, END, TEST and ASSIGN. Every S-graph has one vertex of type BEGIN, called the source and one vertex of type END, called the sink. All other vertices are of type TEST or ASSIGN. Each TEST vertex `v` has two children, which are called `true(v)` and `false(v)`. Each BEGIN or ASSIGN vertex `u` has only one child `next(u)`. Each vertex is labeled with a function. Two nodes are *isomorphic* if they have the same label, and their child or children are isomorphic. A test node is redundant if both its true- and false-branch lead to the same node.

[1] Orthogonality in UML statecharts is not actual concurrency



(a) (b)

Figure 6: (a) An S-Graph of the lower thread of Figure 5 (b) Reduced S-Graph

If there are no two isomorphic nodes in an S-graph, and all redundant tests are eliminated, the graph is said to be *reduced*. A reduced S-Graph can usually be optimized further by reordering the nodes, but this procedure is beyond the scope of this paper.

The S-graph of the FSM of thread `B` is shown in Figure 6. The dashed lines denote false-branches; solid lines denote true-branches. The node labeled `state=B1b1_B2b3` is a TEST node checking if the machine is in state `B1b1_B2b3`, the `e1` nodes check for the presence of event `e1`, while the nodes with labels containing assignments (`:=`) are ASSIGN nodes doing state transitions. Figure 6 (b) shows the reduced S-graph; the reduction resulted in two fever nodes in this minimal example.

The VHDL code for the `B` process generated from an unreduced S-Graph is shown in Figure 7 (a) while Figure 7 (b) shows the code generated from the reduced S-Graph. The code generation from the S-Graphs proceeds in a bottom up fashion node by node. For example in the reduced graph (Figure 7 b)) , we start by translating one of the assignment nodes, e.g. `state:=B1b2_B2b4` directly into VHDL (line 11). Traversing the graph upwards, we find the TEST node labeled `e1`, which is translated into the procedure call and if statement in lines 9 and 10. Continuing up in the graph, we find edges to two TEST nodes resulting in an `or` guard in the `if` statement (line 8). As we reach the begin node, we start over from the bottom choosing the other assign node (`state:=B1b1_B2b4`) etc.

Often the biggest gap in going from system level specifications to an implementation lies in the communication schemes. Each UML statechart is equipped with a FIFO event queue. The communication between state machines in a collaboration diagram is not well defined in the UML standard. The topic of communication synthesis from high level specifications is covered in some detail in e.g. [5]. We take a similar approach, and e.g. generate a procedure `queue_q_get_event( uml_event )` for checking the event on the top of queue `q`. We will not further cover the communication issue in this paper, though an important one.

Synthesizing these two pieces of code from our toy example resulted in about 2 percent smaller area for the reduced version, that is no significant improvements in this case; however, a minimal representation for code synthesis often has bigger advantages.

6 CONCLUSIONS AND FUTURE WORK

We have presented an approach using which, UML behavioral diagrams can be used for system level synthesis of VHDL code. The method can also be used with other visual formalisms. We

```
B: process (clk,reset)
  variable state : StateType := B1b1_B2b3;
  variable uml_event : EventType;
begin
 if reset = '0' then
   state := B1b1_B2b3;
 elsif clk'event and clk = '1' then
   if state = B1b1_B2b3 then
     queue_q_get_event( uml_event );
      if uml_event=e1 then
        state := B1b2_B2b4;
      end if;
    elsif state = B1b1_B2b4 then
      queue_q_get_event( uml_event );
      if uml_event=e1 then
        state := B1b2_B2b4;
      end if;
    elsif state = B1b2_B2b4 then
      queue_q_get_event( uml_event );
      if uml_event=e1 then
        state := B1b1_B2b4;
      end if;
    end if;
  end if;
end process B;
```

(a)

```
B: process(clk,reset )
 variable uml_event : EventType;
 variable state : StateType := B1b1_B2b3;
begin
 if reset = '0' then
   state := B1b1_B2b3;
 elsif clk'event and clk = '1' then
   if state = B1b1_B2b3 or state = B1b1_B2b4 then
     queue_q_get_event( uml_event );
      if uml_event = e1 then
        state := B1b2_B2b4;
      end if;
    elsif state = B1b2_B2b4 then
      queue_q_get_event( uml_event );
      if uml_event = e1 then
        state := B1b1_B2b4;
      end if;
    end if;
  end if;
end process B;
```

(b)

Figure 7: Generated VHDL code (a) non-optimized (b) optimized

translate the system-level specifications into an optimized format, from which we are able to synthesize compact code in different target languages. The ultimate goal of our process would be to skip the software compilers, VHDL synthesis tools etc. and go straight in to assembly code, and hardware netlists.

There are still ambiguities in many UML diagram types, and above all, the semantics of combinations of diagram types, like collaboration and statechart diagrams is not clear. We need to study more in depth the existing communication schemes between active objects in UML etc. And identify missing and ambiguous concepts.

The tools for performing the different transformations between SMDL and the target languages are still under development.

REFERENCES

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

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

[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, Johan Lilius, and Ivan Porres. Towards efficient code synthesis from statecharts. In *pUML Workshop at UML2001*, october 2001.

[5] G. Fernandes Marchioro J.-M. Daveau and A.A. Jerraya. VHDL generation from SDL specification. In C. Delgado Kloos and E. Cerny, editors, *Hardware Description Languages and their Applications (CHDL '97)*, Toledo, Spain, 1997. Chapman and Hall.

[6] Sanjiv Narayan, Frank Vahid, and Daniel D. Gajski. Translating system specifications to VHDL. In *IEEE European Design Automation Conference*, pages 390–394, Amsterdam, The Netherlands, 1991.

[7] OMG. OMG Unified Language Specification. Version 1.3 , March 2000, available from http://www.omg.org.

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

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