Towards Efficient Code Synthesis from Statecharts

Dag Björklund, Johan Lilius and Ivan Porres

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

Abstract: This paper describes a strategy for synthesizing efficient code from UML statecharts based on SMDL, an intermediate language with formal operational semantics. We use an intermediate language to support semantic variations in UML models and different target programming languages. SMDL models are implemented using Software Graphs that can be reduced to generated optimized code.

1 Introduction

The Unified Modeling Language (UML) [**?**] is currently the most widespread software modeling language. UML can be used in any kind of software project and it is accepted by the industry as the standard language for software analysis and design.

UML has five different behavioral diagrams that can be used to describe the dynamics of a system. They are the use case, statechart, activity, sequence and collaboration diagrams. All of these diagrams except use case are closely related: statecharts are used as the semantic foundation of the activity diagrams and it is possible to represent the execution (a trace) of a statechart or an activity diagram as a sequence or collaboration diagram. Use cases differ completely from the previous four diagrams in the sense that they always show the external behavior of a system as perceived by its users, while the other four diagrams can describe the internal behavior as well.

In this paper we present a strategy for synthesizing efficient code from UML statecharts. A UML statechart describes the dynamics of a model element as it changes its internal state as the reaction of receiving some external stimuli. UML statecharts can describe the behavior of a classifier (a class) or a behavioral feature (a method of a class).

The UML behavior diagrams include many concepts that are not present in most popular programming languages, like C++ or Java, e.g. events, states etc. This means there is not a one-to-one mapping between a statechart 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.

We plan to overcome this problem by introducing an intermediate language that we call SMDL. SMDL can be used to synthesize program code from behavioral models. We feel that a successful strategy for code synthesis should be based on the observation that the UML notation is used in a family of languages [**?**]. UML is used in many application domains and there exists variations in the interpretation of diagrams depending on the application, development group, programming language, etc. Using an intermediate language allows us to describe the code synthesis as a two-step process. The first step deals with the variations on the interpretations of UML models with respect to SMDL. The second step deals with the implementation of SMDL models into the chosen target language. In both steps we benefit from the fact that SMDL has a simple syntax and precise semantics.

An important decision when synthesizing code from a modeling language 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. That 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 should be efficient so the programmer does not need to tweak it by hand and it also needs to provide an environment for early simulation and animation of the models, so the designer can validate or debug the models directly using a modeling tool instead of examining and debugging the generated code.

The Rhapsody tool from I-Logix, for example, has adopted the the other approach, i.e. the tool generates a framework of C++ or Java code from the UML model. The tool does not optimize the generated code and the dynamics of the model are both defined in the framework classes and hard-coded in the code generator.

Another difficulty in the synthesis of efficient code from statecharts is that most non-trivial UML models are described as a combination of many classes. This is one of the principles of OOD, a system is designed as a collaboration of interacting objects. This also means that in order for our work to be practical, we should also study how to generate code for collaborations of objects, including their communication mechanisms. In the case of UML, statecharts communicate via event queues.

We proceed as follows. In the following section we present the SMDL language, its syntax and semantics. Section 3 discusses the translation process from UML models to SMDL models while Section 4 discusses the implementation of SMDL models in the C programming language. Finally, Section **??** contains some conclusions and related work.

2 Definition of the Intermediate Language

SMDL 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 SMDL. Instead, we intend it to be both as a tool study the generation of efficient code and a tool for UML scholars to discuss different aspects of UML behavioral semantics. The SMDL language is more abstract than usual programming languages since it supports concepts like traps (high level transitions), suspension and resumption of threads, and event queues. The language has been specifically designed to describe the behavior of modeling languages[1] and, in fact at the moment, it has only control structures but no data primitives. SMDL has a simple syntax, formal operational semantics and there exists a reference implementation and simulation engine available as open source [**?**].

The syntax of SMDL is similar to the syntax of conventional programming languages

[1]The SMDL language is currently being extended to deal with phenomena from other languages that are used in system modeling, like ESTEREL [**?**].

and it is easy to read and write by both humans and machines. The semantics of the language is given in terms of structural operational rules and they are explained in detail in Section 2.2.

2.1 SMDL Syntax

A SMDL model is composed as a series of labeled statements of the form $label$: **statement**. Each label is unique. Statements that directly map to a construct in a UML model, e.g. `state` statements, can be labeled using the corresponding name in the statechart diagram. When SMDL code is executed, there are one or more active labels. If two labels are active at the same time, their associated statements can be executed in parallel.

The basic set of SMDL statements is as follows:

null	null statement
if exp **then** $l_1 : stat_1$ [**else** $l_2 : stat_2$] **endif**	conditional branch
emit $e\ q$	add an event to a queue
dequeue q	delete an event from a queue
par $l_1\ l_2 \ldots l_n$	parallel statement
goto $l_1 \cdots l_n$	state transition statement
state [**entry** $string$ **exit** $string$]	start of state block
endstate l	end of state block
trap exp [**entry** $string$ **exit** $string$]	start of trap block
endtrap l **do** $l_1 : stat_1$	end of trap block
suspend l	suspend a state
resume l_1 **default** l_2	resume a state
$[\![\ statements\ ]\!]$	atomic block

In addition to the statements and the atomic brackets, SMDL has one more important construct, namely the *sequencing operator* `';'`.

Consider two SMDL statements `stat1` and `stat2` with labels `l1` and `l2`. If one wants `stat2` to be executed [2] after `stat1`, a `';'` needs to be inserted between the statements. This will add `l2` to the active set after executing `stat1`.

There are also statements who are forbidden to activate the statement that follows them, for example the `endstate` statement. A semicolon after a `endstate` statement could lead to implicit, or accidental, state transitions.

Figure 1 shows a machine with two states, S1 and S2, encoded in SMDL. The contents of the active set as the code is executed is shown in the comments (comments can be inserted into SMDL code after a #). We see that the sequential code of state S1 will be executed and after this the endstate statement will put S1 back into the active set where-after it can be executed again, i.e. we remain in state S1.

The state `S2` will never be executed, since there is no `';'` after the endstate with label `l3` that could put `S2` to the active set, and there is no state transition from `S1` to `S2`. Writing `l3:endstate S1;` would lead to a state transition from `S1` to `S2`; a '`;`' after a endstate is however forbidden, as mentioned earlier, because we want to allow only explicit state transitions using the goto statement.

[2] A statement can not be forced to be executed next, except when using the atomic brackets; its label is just activated, after which the execution engine may choose to execute it

```
#                         active set
# initial state             {S1}
S1: state           #      {l1}
   l1: null;        #      {l2}
   l2: null;        #      {l3}
l3: endstate S1     #      {S1}
S2: state           #       -
   l4: null;        #       -
l5: endstate S2     #       -
```

program 1: An example of using the sequencing operator

2.2 Semantics

The intuition behind the semantics is very simple. One SMDL program represents a state-machine. The state of a SMDL program is then represented as a tuple $< \alpha, suspend, q >$ 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 represents the active threads of the execution, and the labels represent the current values of the program counters. Some of the threads may be suspended. The operational semantics is given by a set of rules that determine actions of the form:

$$\frac{premiss}{< \alpha, suspend, q > \overset{statement}{\longrightarrow} < \alpha', suspend', q' >}$$

The rules determine how the statements update the state of the machine.

An *execution engine* picks labels from the set of active states and the executes the rule corresponding to the statement at the label. An *execution policy* decides on the order in which labels are picked from the active list. A simple execution policy would view the active set as queue and pick the first label on the queue, and append the resulting new states to the active-set. This would correspond to a Round-Robin scheduler. We can also define a policy that corresponds to the interpretation of orthogonality as defined in the UML standard. In this case the policy dictates that we should look for all enabled labels (i.e. active labels whose statement can be executed). The statements defined by this set of labels are then executed sequentially in some order before we start looking for a new set of statements. Other execution polices are also possible: e.g. it is easy to introduce a priority based policy by attaching priorities to all the labels.

Before we can look in detail at the operational rules we need to formalize the structure of an SMDL program. An SMDL program is defined by the tuple $\langle\ \Sigma,\ \uparrow, \succ, \mathcal{L}\ \rangle$ where:

- Σ is the set of labels.
- $\uparrow\ \subset \Sigma \times \Sigma$ is the *parent* relation. The parent relation organizes the labels in Σ hierarchically in a tree.

- $\succ \subset \Sigma \times \Sigma$ is the *immediate successor* relation. 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 take the closure $\uparrow^*$ of the parent relation $\uparrow$. Let l_1 and l_2 be labels, then $l_1 \uparrow^* l_2$ means that l_1 is an *ancestor* of l_2.

We use the domain/range restriction operators $\lhd$ / $\rhd$ and the domain/range subtraction $\ntrianglelefteq$ / $\ntrianglerighteq$ operators as defined in [**?**] 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. The domain/range subtraction operators are similar, except they remove the elements in the set from the relation.

We now discuss the rules that define the semantics of the indivudial statements. There are in total 18 rules that define the semantics of the language. In this paper we only show the most basic rules but the reader can find in [**?**] a complete description of all rules.

Rule 1 Parallel statements

For a statement to be executed, its label has to be active i.e. a member of α. The `par` statement updates the active set by, as most statements, removing its own label from the set; it also adds all its argument labels to the active set. This means that the arguments then can be executed in parallel. The `par` statement is used to model orthogonal regions.

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

Rule 2 State

A `state` statement models a state in a state machine; it can act when its label is active. It removes its label from the active set and adds its successor to the set. It will also remove all its ancestors from the domain of the $suspend$ relation. This, because a state that is suspended may be entered without calling a resume statement, which would leave garbage in the $suspend$ relation without this procedure. History states e.g. are modeled in SMDL using the `suspend` and `resume` statements.

$$\frac{l \in \alpha \wedge \mathcal{L}[l] = state \wedge l \succ l_1}{<\alpha, suspend, q> \overset{l:state}{\longrightarrow} <\alpha - \{l\} \cup \{l_1\}, (\uparrow^* \rhd \{l\}) \ntrianglelefteq suspend, q>}$$

Rule 3 Endstate

An endstate statement can act when its label is active. It removes its label from the active set, and reactivates the label of the corresponding state statement i.e. the `state` block can be executed again, until a state transition is taken out of it.

$$\frac{l \in \alpha \wedge \mathcal{L}[l] = endstate\ l'}{<\alpha, q> \overset{l:endstate}{\longrightarrow} <\alpha - \{l\} \cup \{l'\}, q>}$$

Rule 4 State transition

$$\begin{tikzcd}[column sep=tiny]
 & & & & l_1 \arrow[dl] \arrow[dr] & & & & & & & & l_1 \arrow[dl] \arrow[dr] & & \\
 & & & \mathbf{l_2} \arrow[dll] \arrow[drr] & & \mathbf{l_3} & & & & & & \mathbf{l_2} \arrow[dll] \arrow[drr] & & \mathbf{l_3} & \\
 & l_4 \arrow[dl] \arrow[dr] & & & & l_5 \arrow[dl] \arrow[dr] & & \longrightarrow & & l_4 \arrow[dl] \arrow[dr] & & & & \mathbf{l_5} \arrow[dl] \arrow[dr] & \\
\mathbf{l_6} & & l_7 & & l_8 & & l_9 & & l_6 & & l_7 & & l_8 & & \mathbf{l_9}
\end{tikzcd}$$

Figure 1: Label trees: a) before the transition b) after the transition

The `goto` statement is used to model state transitions in SMDL. It takes one or many labels as parameters; a fork pseudo state can be modeled by giving a `goto` statement several parameters. A goto statement can act when its label is active. It removes all the labels in the subtree containing both the source and the destination(s) of the transition from the active set. Then all the labels of any trap statements that are ancestors of the destination labels are added to the active set, and of course all the destination labels.

$$\frac{l \in \alpha \wedge \mathcal{L}[l] = goto\ l_1\ l_2\ \cdots\ l_n}{< \alpha > \overset{goto\, l_1\ l_2 \cdots l_n}{\longrightarrow} < \alpha - ran(\{l_{min}\} \lhd \uparrow^+) \bigcup_{i=1}^{n} trapanc(l_i) \cup \{l_1, \ldots, l_n\} >}$$

where l_{min} is the root of the minimal subtree containing l and $l_1, \ldots, l_n$, and $trapanc(l)$ gives the set of labels belonging to trap statements that are ancestors of label l.
Example
As an example, consider a machine with labels:
$\Sigma = \{l_1, l_2, l_3, l_4, l_5, l_6, l_7, l_8, l_9\}$
parent relation:
$\uparrow\ = \{l_1 \mapsto l_2, l_1 \mapsto l_3, l_2 \mapsto l_4, l_2 \mapsto l_5, l_4 \mapsto l_6, l_4 \mapsto l_7, l_5 \mapsto l_8, l_5 \mapsto l_9\}$
current active set:
$\alpha = \{l_2, l_3, l_6\}$
and some of the labels mapped to statements as follows:
$\mathcal{L}[l_2] = trap\ e_1\ q_1\ in$
$\mathcal{L}[l_5] = trap\ e_2\ q_1\ in$
$\mathcal{L}[l_6] = goto\ l_9$
$\mathcal{L}[l_9] = state$
The label tree before the transition is shown in Figure 1. The labels that are active are bolded.

First we find that the root of the minimal subtree spanning l_6 and l_9 is l_2.
The active set after the transition then becomes:
$\{l_2, l_3, l_6\} - ran(\{l_2 \mapsto l_4, l_2 \mapsto l_5, l_2 \mapsto l_6, l_2 \mapsto l_7, l_2 \mapsto l_8, l_2 \mapsto l_9\}) \cup l_5 \cup l_9$
$= \{l_2, l_3, l_5, l_9\}$

The label tree after the transition is shown in Figure 1 b).

Rule 5 Trap

The `trap` statement can be used to model states with top level transitions. It can also be thought of as an interrupt handler. It does not remove its label from the active set when its block is entered; hence, it can be executed at any time while inside its block. It chooses its branch depending on whether the expression evaluates to true or false, and whether it has active children.

The expression is true; execute the do statement without allowing interrupts.

$$\frac{l \in \alpha \wedge exp = true \wedge \mathcal{L}[l] = trap\ exp \wedge < \alpha \cup \{l_1\}, q > \stackrel{l_1:stat_1}{\longrightarrow} < \alpha', q' >}{< \alpha, q > \stackrel{l:trap\ exp}{\longrightarrow} < \alpha', q' >}$$

Where $l_1 : stat_1$ is the statement in $endtrap\ do\ l_1 : stat_1$

The expression is false and the trap statement has no active descendants; activate the successor of l:

$$\frac{l \in \alpha \wedge min(l, \alpha) \wedge exp = false \wedge l \succ l_1 \wedge \mathcal{L}[l] = trap\ exp}{< \alpha, q > \stackrel{l:trap\,exp}{\longrightarrow} < \alpha \cup \{l_1\}, q >}$$

The expression is false and the trap statement has active descendants; do nothing:

$$\frac{l \in \alpha \wedge \neg min(l, \alpha) \wedge exp = false \wedge \mathcal{L}[l] = trap\ exp}{< \alpha, q > \stackrel{l:trap\ exp}{\longrightarrow} < \alpha, q >}$$

Rule 6 Endtrap

An `endtrap` statement can act when its label is active. An endtrap statement marks the end of a trap block. It is of interest mainly for the preprocessor, it has no semantic function in runtime. The `do` statement is executed by the corresponding trap statement.

$$\frac{l \in \alpha \wedge \mathcal{L}[l] = endtrap\ l'\ do\ l_1 : stat}{< \alpha, q > \stackrel{endtrap\ l'\ do\ l_1:stat}{\longrightarrow} < \alpha - \{l\}, q >}$$

3 Converting UML Models to SMDL

SMDL models are generated and interpreted by a software tool. The transformation from a UML model into a SMDL model has to deal with the fact that UML is a family of languages, both at the semantic and the syntactic level. At the syntactic level there exists differences between the different implementations of the XMI, although we expect that, in the future, most UML tools will adopt the XMI metamodel as specified in [**?**]. However, the most important differences exist in the interpretation of UML models.

These differences are created because UML can be applied in all kind of software projects and it has to be adapted to different models of computation, communication, concurrency and time. Also, the UML language is rich and provides many different modeling concepts that can be combined in many different ways. A simple example of this is the interpretation of the statechart shown in Figure 2. If states S1 and S3 are active and the current event is e, both transitions will fire in the same step. However, depending on our interpretation of orthogonality the actions associated with the transitions will be executed in different orders. For example, if we fire the transitions sequentially, in a deterministic order, the actions could be executed in this order: $a1; a2; a3; a4; a5; a6$. We use the semicolon to denote the sequential composition of actions. The ordering $a4; a5; a6; a1; a2; a3$ is allowed as well.

A concurrent implementation for statecharts could arrange the execution in the following way: $(a1; a2; a3)||(a4; a5; a6)$ where $||$ stands for the parallel composition of actions.

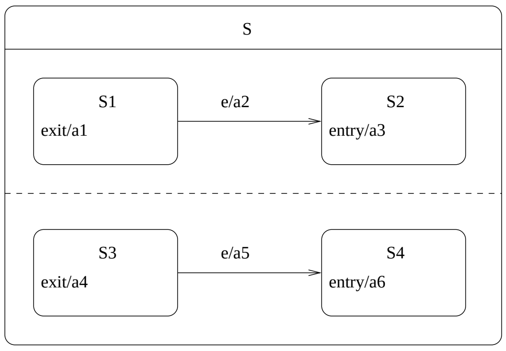


Figure 2: Interpretation of orthogonality

.

However, the definition of entry and exit actions in the UML semantics specification also allows other orders of execution, like $(a1||a4); (a2||a5); (a3||a6)$. All these different ways to interpret the previous diagram are valid and their adoption can be justified by the application domain of the model or the target language.

We plan to support semantic variation in the UML language by providing different translations from UML to SMDL. However, the semantics of SMDL is fixed as well as the code generation of SMDL models into the target programming language and we can reuse them in any variant of UML.

SMDL does not support directly all the UML model elements since this would increase the complexity of the language and prevent semantic variations. Instead, we are developing a set of transformation rules that show how to represent a complex UML model element using a combination of more basic model elements. The objective is to apply these rules until the UML model only contains basic elements that have a counterpart in SMDL so the model can be translated almost trivially. Currently, the transformation rules are hard-coded into the tool but in the future we want to be able to define them in a configuration file. As an example of these rules, we describe below how to remove activities and initial states.

3.1 Activities

Each state can be associated with an activity that is performed while the state is active. We assume that we know how to start and stop an activity and to detect when it has terminated, since these concepts appear implicitly in the UML Semantic guide. Given an activity a we name these actions $a.start$ and $a.stop$ and the query as $a.finished$.

We can remove the "do Activity" association from a state by starting and stopping it using the entry and exit actions of the state. If there is a completion transition associated with the state, then the completion transition is changed by a transition triggered by the change event $a.finished()$. The following algorithm shows how to apply these changes in a state:

```
removeActivityFromState(s: State):
 s.entry:= ActionSequence(action=(s.entry,s.doActivity.start))
 s.exit := ActionSequence(action=(s.doActivity.stop ,s.exit))
 forAll transition t in s.outgoing:
```

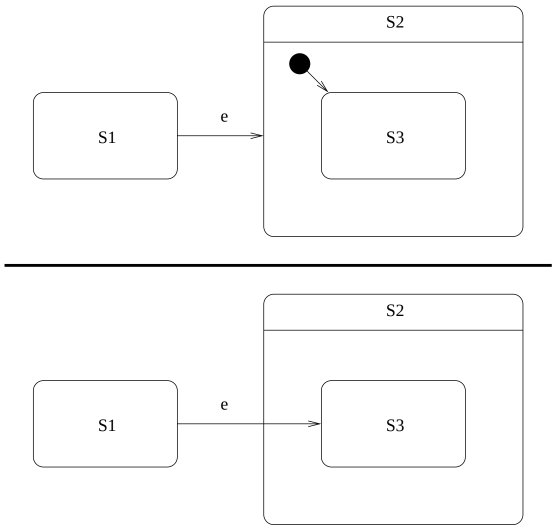


Figure 3: Removing initial states

.

```
  if t.trigger=None then
   t.trigger:=ChangeEvent(changeExpression=s.doActivity.finished)
 s.doActivity:=None
```

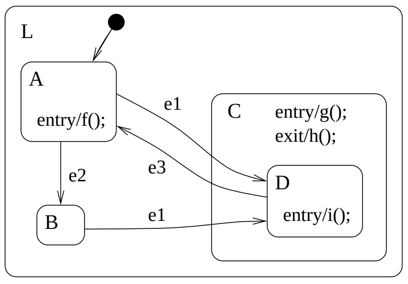
3.2 Initial states

Another transformation is the removal of initial pseudostates. We can remove an initial pseudostate by redirecting the transitions directed to its container to the state vertex pointed by the initial pseudostate. Figure 3 shows an instance of this transformation.

The transformation is described in the following algorithm. To simplify the exposition, we assume that the composite state is not concurrent, i.e. it does not contain orthogonal regions. Otherwise, it is necessary to add a fork pseudostate so all transitions directed to the composite state are forked to the default state of each orthogonal region.

```
defaultState(cs: CompositeState}: StateVertex
 assume not cs.isConcurrent
 result=None
 forAll statevertex sv in cs.vertex:
  if sv.kind = PseudoStateKind.pk_initial then
   result=sv.outgoing.target

removeInitialState(cs: CompositeState):
 assume not cs.isConcurrent
 if defaultState(cs)!= None then
  -- redirect transitions
  forAll transition t in cs.incoming:
   t.target=defaultState(cs)
  -- remove intial pseudostates
  forAll statevertex sv in cs.vertex:
   if sv.kind= PseudoStateKind.pk_initial then
```



```
L: state
  A: state entry 'f();'
    l1:  if e1 in q then
      l2:  goto D
    endif;
    l3:  if e2 in q then
      l4:  goto B
    endif;
  l5:  endstate A
  B: state
    l6:  if e1 in q then
      l7:  goto D
    endif;
  l8:  endstate B
  C: state entry 'g();' exit 'h();'
    D: state entry 'i()'
      l9:  if e3 in q then
        l10:  goto A
      endif;
    l11:  endstate D
  l12:  endstate C
l13:  endstate L
```

Figure 4: Example UML statechart and its translation into SMDL

```
    cs.vertex.remove(sv)
```

We do not use the algorithm to remove the initial state associated with the top state of the statechart (the state that transitively contains all other states). In this case, the initial state is translated into the first label of the SMDL model.

Figure 4 shows an example statechart and its transformation into SMDL. In SMDL we do not give any interpretation for the actions in the state machine and we assume that they are written in the target language.

Once we have translated a UML model into SMDL we can animate the model to validate it. We have implemented a prototype of an animation tool based on the operational semantics presented before. The animation tool presents a dialog box that allows the user to generate events for the model and observe the active set of states in the model.

It could be possible to interface the animation tool with a UML editor to represent graphically the activity of the model as it is done in tools like Rhapsody. The information needed to trace back the activity of the SMDL model into the UML model can be encoded in the labels of the SMDL statements.

4 Implementation of SMDL Models

We are working towards efficient code generation from SMDL models. Our intention is that the generated code does not need to be optimized by hand by a programmer. The code synthesis process is similar to that of the POLIS approach [**?**] 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.

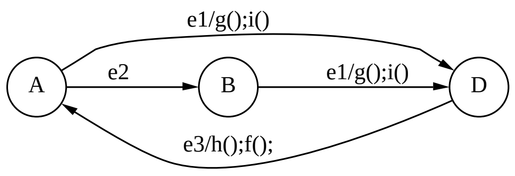


Figure 5: Flat state machine
.

The S-Graphs can be minimized, which allows us to generate code that is optimized on a higher level than what the target language compiler is capable of; furthermore, compact assembly code could be synthesized directly for targets lacking higher level language compilers. 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.

The translation of a SMDL model into optimized code proceeds in five steps.

1. Translation (flattening) of the SMDL code to a simple finite stat machine (FSM)
2. Translation of the FSM into an S-graph
3. S-graph optimization
4. Translation of the S-graph into a target language
5. Compilation into machine code.

The first step is the flattening of the state machine. Flattening means transforming the SMDL model into a flat finite state machine FSM, or a transition table. We use an algorithmic method to flatten the model, and we also have prototype software performing the task. The FSM produced by our software from the example SMDL code is shown in Figure 5.

The flat state machine is translated into an S-graph, which is optimized for size. 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. 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.

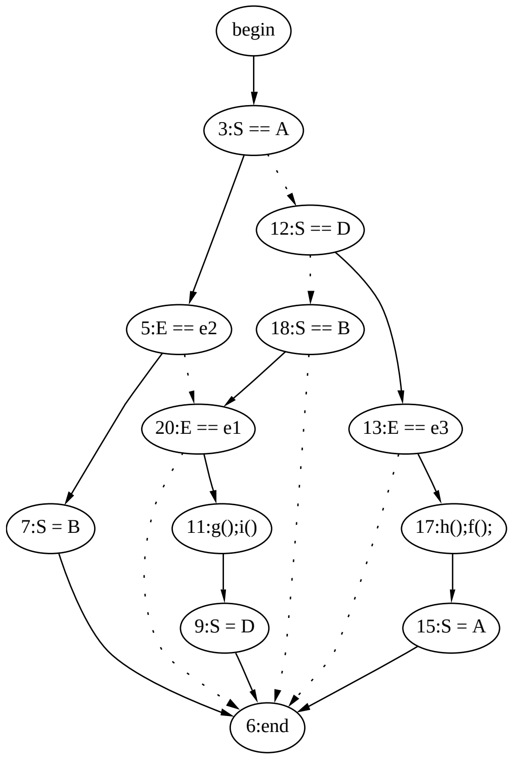
The code synthesis will produce a function that takes the current state and an event as parameters, and returns the next state. The state parameter is called S, and the event parameter is called E. For example, to test if the machine is in state A, a test node with the label $S = A$ is generated. To do a state transition to state B, an assign node with label $S := B$ is produced.

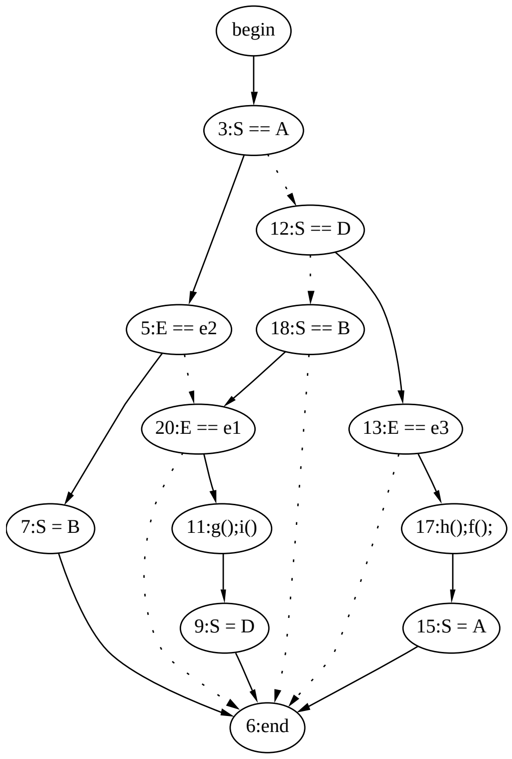
The optimized S-graph of the machine in Figure 4 is shown in Figure **??**. The dashed lines denote false-branches; solid lines denote true-branches.

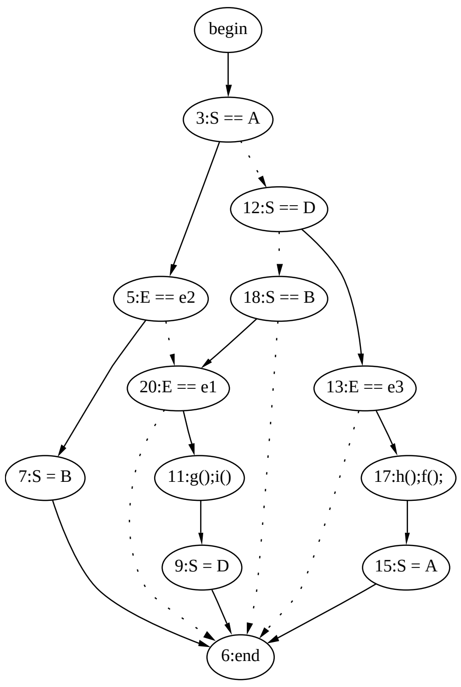
The translation of the S-graph to C code is straightforward due to the direct correspondence between the nodes in the graph and the C primitives. Also e.g. assembly code or even logic gates could easily be generated from the S-graphs.

As can be seen in the figures, each node in the S-graphs have a unique index. These are used as labels that can be jumped to using the goto statement in the C code. Each operation performed by a node is labeled in the code, thus nodes that are reused, that is, have multiple parents in the S-graph can also be reused in the code when all the parents generate goto statements to the same label. This unstructured code hinders its readability, but as we stressed earlier, no human should ever read this code.

From the test nodes, a conditional branch structure is generated; if the test is true, a jump is taken to the label of the true branch of the test node, otherwise a jump to the false branch is taken. Assign nodes result in the action in the label of the node, followed by a jump to the label of the next node of the assign node. The code can be slightly reduced further by combining subsequent assign nodes with single parents. In the C code this results in a goto statement being saved when two assign nodes are combined. The C code generated from the running example is included in Figure **??**. We explain the algorithms used for S-graph optimization and implementation in [**?**].







```
int trans( int S, int E ) {
  l3:  if ( S == A )
    goto l5;
  else
    goto l12;
  l5:  if ( E == e2 )
    goto l7;
  else
    goto l20;
  l7:  S = B;
  l6:  return S;
  l20:  if ( E == e1 )
    goto l24;
  else
    goto l6;
  l24:  g();i(); S = D; goto l6;
  l12:  if ( S == D )
    goto l13;
  else
    goto l18;
  l13:  if ( E == e3 )
    goto l17;
  else
    goto l6;
  l17:  h();f();S = A; goto l6;
  l18:  if ( S == B )
    goto l20;
  else
    goto l6;
}
```

Figure 6: Optimal S-graph with 13 nodes and its implementation in C

5 Summary

We have presented a strategy for code generation from statecharts based on SMDL. SMDL is a language for modeling behaviors and it can be used to capture the dynamics of UML behavioral diagrams. SMDL cannot describe data structures but it has built in support for concurrency, event queues, etc. It has a simple syntax and formal semantics given as structural operational rules.

This is still work in progress. We have only applied SMDL to a subset of UML statecharts and their implementation in C. However, we think that this approach can be extended to other behavioral diagrams like activity and interaction diagrams and other target languages. There is a prototype implementation of a tool for code generation and animation of SMDL models. Currently, the transformation rules are hard-coded into the tool and it only support a limited subset of UML features.

There are many interesting articles discussing the semantics of UML behavioral diagrams and in concrete of UML statecharts, for example see [**?**, **?**]. However, it seems that their implementation is not discussed so often.

Bibliography

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

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

[BL01] Dag Björklund and Johan Lilius. Towards a kernel language for heterogenous computing. submitted paper, 2001.

[Bör01] Dag Börklund. *The cSMDL System, homepage*. Internet: http://infa.abo.fi/˜dbjorklu/cSMDL, 2001.

[CKM$^+$99] S. Cook, A. Kleppe, R. Mitchell, B. Rumpe, J. Warmer, and A. Wills. Defining UML family members using prefaces. In IEEE, editor, *Procceedings of TOOLS 32*, 1999.

[LMM99] Diego Latella, Istvan Majzik, and Mieke Massink. Towards a formal operational semantics of UML statechart diagrams. In *3rd International Conference on Formal Methods for Open Object-Oriented Distributed Systems,* Boston, 1999. Kluwer Academic Publishers.

[LP99] J. Lilius and I. Porres. Formalising UML state machines for model checking. In Robert France and Bernhard Rumpe, editors, *UML'99 - The Unified Modeling Language. Beyond the Standard. Second International Conference, Fort Collins, CO, USA, October 28-30. 1999, Proceedings*, volume 1723 of *LNCS*. Springer, 1999.

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

[OMGb] OMG. OMG XML Metadata Interchange (XMI) Specification. 2000, available from http://www.omg.org.

[PST91] Ben Potter, Jane Sinclair, and David Till. *An Introduction to Formal Specification and Z.* Prentice Hall, 1991.