A Translation of Statecharts to Esterel

S. A. Seshia[1*], R. K. Shyamasundar[1], A. K. Bhattacharjee[2], and S. D. Dhodapkar[2]

[1] School of Technology & Computer Science,
Tata Institute of Fundamental Research, Mumbai 400 005, India
shyam@tcs.tifr.res.in

[2] Reactor Control Division, Bhabha Atomic Research Centre, Mumbai 400 025, India
{anup,sdd}@magnum.barc.ernet.in

Abstract. Statecharts and Esterel are two formalisms that have been widely used in the development of reactive systems. Statecharts are a powerful graphical formalism for system specification. Esterel is a rich synchronous programming language with supporting tools for formal verification. In this paper, we propose a translation of Statecharts to Esterel and discuss such an implementation. A characteristic feature of the translation is that deterministic Statechart programs can be effectively translated to Esterel and hence, the tools of verification of Esterel can be used for verifying Statechart programs as well. The translation serves as a diagnostic tool for checking nondeterminism. The translation is syntax-directed and is applicable for synchronous and asynchronous (referred to as the superstep model) models. In the paper, we shall describe the main algorithms for translation and implementation and illustrate the same with examples. We have built a prototype system based on the translation. It has the advantages of the visual power usually liked by engineers reflected in Statecharts and of a language that has a good semantic and implementation basis such as Esterel that can be gainfully exploited in the design of reliable reactive systems.

1 Introduction

Significant amount of research has been done in the last decade in the design and development of reactive systems. The class of synchronous languages and various visual formalisms are two approaches that have been widely used in the study of reactive systems. The family of synchronous languages has based on *perfect synchrony hypothesis* which can be interpreted to mean that the program reacts rapidly enough to perceive all the external events in a suitable order and produces the output reactions before reacting to a new input event set. Embedded controllers can be abstracted in this way. Some of the prominent languages of the family include Esterel, Lustre, Signal etc. These languages are also being used widely in industry. Significant advantages of the family of synchronous

---

* Current address:School of Computer Science, Carnegie Mellon University, Pittsburgh,PA 15217, USA, email: Sanjit.Seshia@cs.cmu.edu

languages include the availability of idealized primitives for concurrency, communication and preemption, a clean rigorous semantics, a powerful programming environment with the capability of formal verification. The advantages of these languages are nicely paraphrased by Gerard Berry, the inventor of ESTEREL, as follows: *What you prove is what you execute.*

Statecharts is a visual formalism which can be seen as a generalization of the conventional finite automata to include features such as hierarchy, orthogonality and broadcast communication between system components. Being a formalism rather than a language, there is no unique semantics in the various implementations and further Statechart specifications can be nondeterministic. For these reasons, even though there are powerful programming environments for Statecharts such as STATEMATE[1] (which includes simulators), environments lack formal verification tools.

Textual and graphical formalisms have their own intrinsic merits and demerits. For instance consider the following reactive system design:

> Consider the specification of control flow (switching of tasks) among various computing tasks and interrupt service tasks in a control software. The computing tasks switch from one to another in cyclic fashion and are shown as substates of `compute_proc`. The interrupt service tasks are entered as a result of the occurrence of interrupt events. The history notation has been used to indicate that on return from interrupt tasks, the system returns to last executing compute task (except when event 100 ms occurs, the control returns to `compute task hpt`). The event `wdt_int` occurs on system failure and it can be verified that when `wdt_isr` is entered, the system will toggle between states `wdt_isr` and `nmi_isr`, which is the intended behavior.

Such systems can be specified using graphical formalisms easily. The statechart for the above system is shown in Figure 1. Arguing the formal correctness from such descriptions, however, is not easy. Our work is concerned with methods that will combine advantages of using graphical formalisms for the design of reactive systems with that of using formal verification tools in textual formalisms.

In this paper, we study a method of translating Statechart formalisms into ESTEREL with the idea that the powerful verification tools and code optimization tools of ESTEREL can be applied for Statechart programs. Our aim has been to provide a clean formally verifiable code for Statechart programs rather than yet another attempt to define the semantics of Statecharts. For this reason, we stick to using the STATEMATE semantics (refer [7]), which is an industrial strength version of Statecharts. It must be noted that ESTEREL is deterministic and hence, our study will confine to the deterministic class of Statecharts. However, it may be noted that the translation procedure will detect the underlying nondeterminism if any.

We discuss algorithms of translation and discuss the implementations and also compare our study with respect to other similar translations of Statecharts.

[1] STATEMATE is a registered trademark of I-Logix Inc.

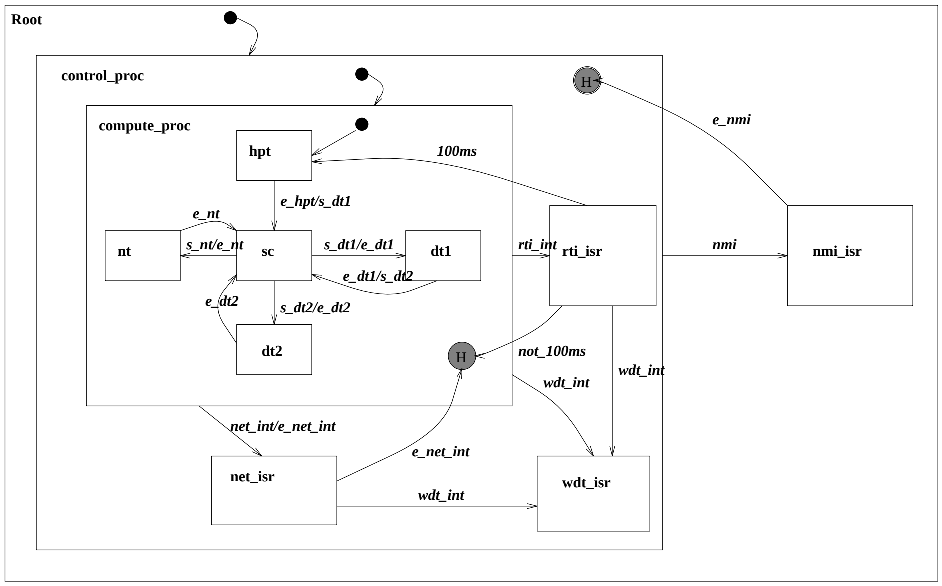


Fig. 1. Example of Switching Interrupts

The main advantage of our translation is that the code generated is verifiable and also, ESTEREL optimizers can used for efficient code generation. We are currently in the process of evaluating the quality of code generated vis-a-vis other Statechart code generators.

The rest of the paper is organized as follows: Section 2 briefly introduces Statecharts, ESTEREL and the STATEMATE semantics. In section 3, we discuss how we abstract out the essential details of the Statechart and the core ideas in the translation process, along with illustrative examples. Section 4 sums up the work along with comparisons to other works.

2 Background

2.1 Statecharts

In this section, we present a brief overview of Statecharts (see [6] for complete details). Statechart shown in Figure 2 (derived from the example in [6]) is used for illustrative purposes.

Components of a Statechart:

States: These are of three types: *basic states*, *and states* and *or states*. Basic States are those states which do not contain any other state, e.g. *lap* is a basic state.

An Or-state is a compound state containing two or more other states. To be in a Or-state is to be in one of its component states. In this paper, we will

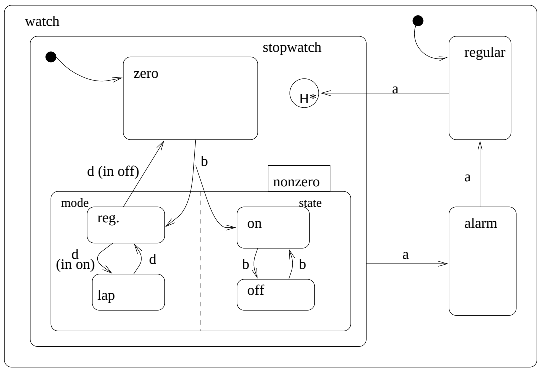


Fig. 2. Statechart of stopwatch within a wristwatch - with deep history

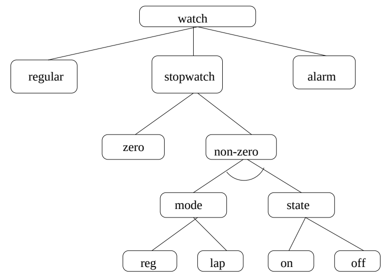


Fig. 3. AND-OR tree representation of the Statechart of wristwatch

use Or-State synonymously with XOR-state, i.e. we can be in only one of the component states at a given time. An example of an Or-state in Figure 2 is *stopwatch*.

An And-state is also a compound state and staying in an And-state implies staying in each one of its substates. These are provided to model concurrency. The substates of an And-state may contain transitions which may be executed simultaneously. *nonzero* shown in Figure 2 is an And-state.

Transitions: A Transition in the Statechart is a five-tuple *(source, target, event, action, condition)*. The arrow on the Statechart goes from source to target and is labelled as **e[C]/a**, meaning that event **e** triggered the transition when condition **C** was valid and action **a** was carried out when the transition was taken. In general, **a** could be a list of actions to be taken.

History and Defaults: Statecharts incorporates the idea of a *history* state in a OR-State. The history state keeps track of the substate most recently visited. This is denoted by **H** in a Or-state, as in the or-state stopwatch in Figure 2. A *default* state, marked by a shaded circle, is a substate of an or-state such that if a transition is made to the or-state and no other condition (e.g. enter-by-history) is specified, then that substate must be entered by default. e.g. *regular* is the default substate for the *watch*. In Figure 2, we have a *deep-history* state, which means that a transition to that state implies being in the maximal most recent set of basic substates. This can be represented by history states in each one of the Or-substates.

2.2 Statemate

The informal semantics of the Statemate version of Statecharts is provided through rules describing the semantics of a step. The main rules are listed below. For detailed discussions, the reader is referred to [7].

1. Reactions to external/internal events and changes that occur in a step can be sensed only after completion of the step.
2. Events are "live" for the duration of the step following the one in which they occur only.
3. Calculations in a step are based on the situation at the beginning of the step
4. If two transitions are in conflict, then priority is given to that transition whose scope is higher in the hierarchy. The scope as defined in [7] is: The scope of a transition *tr* is the lowest Or-state in the hierarchy of states that is a proper common ancestor of all sources or targets of *tr*, including nonbasic states that are explicit sources or targets of transition arrows appearing in *tr*.
5. Each step follows the Basic Step Algorithm as described in [7].

2.3 Esterel

The basic object of Esterel without value passing (referred to as Pure Esterel) is the signal. Signals are used for communication with the environment as well as for internal communication.

The programming unit is the module. A module has an interface that defines its input and output signals and a body that is an executable statement:

```
module M:
   input I1, I2;
   output O1, O2;
   input relations
   statement
end module
```

Input relations can be used to restrict input events and a typical exclusion relation is declared as

```
relation I1 # I2;
```

Such a relation states that input events cannot contain `I1` and `I2` together. That is, it is an assertion on the behavior of the asynchronous environment.

At execution time, a module is activated by repeatedly giving it an input event consisting of a possibly empty set of input signals assumed to be present and satisfying the input relations. The module reacts by executing its body and outputs the emitted output signals. We assume that the reaction is *instantaneous* or *perfectly synchronous* in the sense that the outputs are produced in no time. Hence, all necessary computations are also done in no time. In Pure Esterel these computations are either signal emissions or control transmissions between statements; in full Esterel they can be value computations and variable updates as well. The only statements that consume time are the ones explicitly requested to do so. The reaction is also required to be deterministic: for any state of the program and any input event, there is exactly one possible output event. In perfectly synchronous languages, a *reaction* is also called an *instant*. There is one predefined signal, the *tick*, which represents the activation clock of the reactive program.

Statements: ESTEREL has two kinds of statements: the kernel statements, and the derived statements (those that can be expanded by macro-expansions) to make the language user-friendly. The list of kernel statements is:

```
nothing
halt
emit S
stat1; stat2
loop stat end
present S then stat1 else stat2 end
do stat watching S
stat1 || stat2
trap T in stat end
exit T
signal S in stat end
```

Kernel statements are imperative in nature, and most of them are classical in appearance. The `trap-exit` constructs form an exception mechanism fully compatible with parallelism. Traps are lexically scoped. The local signal declaration "`signal in` *stat* `end`" declares a lexically scoped signal S that can be used for internal broadcast communication within stat. The `then` and `else` parts are optional in a present statement. If omitted, they are supposed to be `nothing`. Informal semantics of the kernel constructs are given in Appendix C.

3 The Translation

A Statechart basically denotes a network of automata with hierarchy and other properties. The crux of the translation lies in

(A) Extracting the hierarchy of states and transitions,
(B) Resolving the conflict in the transitions as per the STATEMATE semantics,
(C) Generating the code corresponding to the transitions between states,
(D) Generating code that models system state between transitions, and,
(E) Generating code that supports communication via events and actions.

In the following, we shall highlight the underlying issues of representation, resolution of conflicts and code generation. Note that we refer to signals in the Statechart as *actions* or *events*, while those in ESTEREL are referred to simply as *signals*. We first present the underlying ideas and the full code generation algorithm is presented at the end.

3.1 AND-OR Tree Representation of Statecharts

The Statechart can be represented as an AND-OR tree: being in an AND-node meaning that the system is in each of its child nodes, while being in an OR-node means that we are in exactly one of its child nodes. Such a representation allows us to express the hierarchy of states of the Statecharts in a convenient manner to trace the path of arbitrary transitions. This also allows us to resolve conflicts between enabled transitions, by calculating the *scope*(refer to section 2.2).

For purposes of code generation, we actually use an annotated representation of AND-OR tree described in the following section. An AND-OR tree representation of the Statechart of Figure 2 is shown in Figure 3.

Annotated AND-OR Tree Representation: The annotated AND-OR tree keeps track of information about the Statechart pertinent for the translation, such as (i) the states and their types, (ii) hierarchy of States, and (iii) Transitions between states, which includes Entry and Exit points for each transition & Inner states that need to be hidden (signals suppressed) during a transition that exits a state.
Each node A of the AND-OR tree is represented as a seven-tuple[2]:

$$(Name, Type, T_{entry}, T_{exit}, T_{loop}, T_{default}, T_{history}), \text{ where,}$$

$Name$: Name of the state, viz. A.
$Type$: AND, OR or BASIC.
T_{entry}: The set of all transitions that enter A.
T_{exit}: The set of all transitions that exit A.
T_{loop}: The set of all transitions that exit one of A's immediate child states and enters another(possibly same) child state.
$T_{default}$: The single transition to an immediate child state from A.
$T_{history}$: The set of transitions to the history state of A.

For translating Statecharts, we need to keep track of the Entry and Exit Point Information so that the transitions including the inter-level transitions can be enabled in the translated Esterel code preserving the Statemate semantics. The actual information we need to keep track of will be clear by considering the states between which the transition takes place. Transitions in Statecharts can be broadly classified as:

T1: Between child states of the same parent.
T2: From a parent state to its (not necessarily immediate) child state.
T3: From a child state to its (not necessarily immediate) parent state.
T4: Any transition that is not of type T1, T2 or T3.

Note that all of these transitions may not occur in a given Statechart. In particular, types T2 and T3 may not occur, but the way they are translated forms part of the translation for type T4. The book keeping of the above classes of transitions is achieved through the Node-Labelling Algorithm by keeping the appropriate entry and exit information in each node of the AND-OR tree.
Node-Labelling Algorithm: Assuming levels of the nodes in the tree have already been computed (with root node having level 0, and increasing level for its child nodes), for each transition in the set Tr of transitions, the algorithm

[2] We shall use *node* synonymously with *state* and vice-versa.

traverses the path from source node n_1 to target node n_2, labelling these two nodes as well as intermediate nodes with: (i) name of the transition, (ii) type of the transition, viz. T1, T2, T3 or T4 and (iii) the fact whether the transition is entering that node or exiting it. This information is used to generate code in the translation.

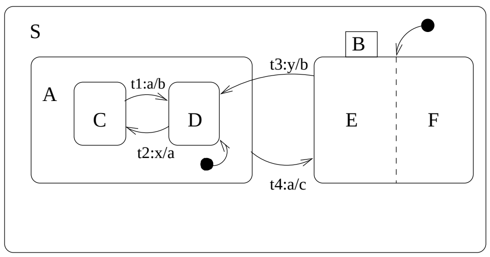


Fig. 4. Example Statechart

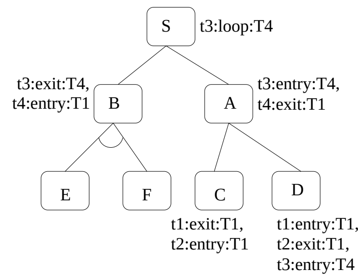


Fig. 5. Corresponding Node-labelled AND-OR tree

3.2 Labelling for Transition Conflict Resolution

As per STATEMATE semantics, two transitions are in conflict if they exit a common state A. Further, conflict resolution is based on the following: Transition t_1 has priority over transition t_2 if the lowest[3] Or-state exited by t_1 is lower than the lowest Or-state exited by t_2.

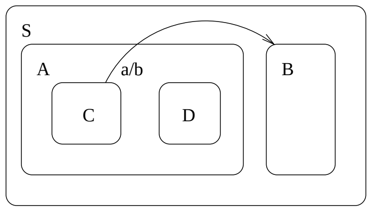
Given this, if trigger events for t_1 and t_2 occur simultaneously then, we must ensure that t_2 is not taken along with its actions. This is done by a signal $hide\_A$. On taking t_1, $hide\_A$ will be emitted. Therefore, before t_2 is taken, a check must be made for the presence of signal $hide\_A$.

This is indicated in the AND-OR tree by traversing the tree top-down, maintaining a list of *"hide-signals"* that we need to label the nodes with. At a node, which has *at least* one transition that exits it, and which is either the source of that transition, or the last state exited by it, we label all of its children with $hide\_A$. This is to ensure that while translating, a statement to check for the presence of $hide\_A$ is executed before any transition is taken. This will perform the job of hiding internal signals. The algorithm to implement hide_signal labeling is omitted here for brevity.

3.3 Code-Generation

The Code-Generation is done in a top down manner traversing the AND-OR tree. In short, the process is as follows : (1) Declare all necessary signals, (2) generate code for states and transitions between states, (3) generate code to

[3] Lowest means closest to the root node.




```
module S :
....
loop [ await immediate goA;
   trap TA in [
      [[A]]; exit TA;
   end trap
   present sig_A_to_S then
        emit goB;
   end present
] end loop
||
loop [ await immedaite goB;
   trap TB in [
      [[ B ]]
   end trap
] end loop
```

```
module A :
....
trap TA' in [
loop [ await immediate goC;
    trap TC in [
       [[ C ]]
     ||
       await immedaite a; await STEP; emit b; emit sig_C_to_A; exit TC   ]
     end trap ]
     present sig_C_to_A then
           emit   sig_A_to_S;
           exit TA'  /* module A exits and returns to S */
     end present;
] end loop
....
]  end trap
] end module
```

Fig. 6. Translation of a transition of type T4

do communication within the Statechart, (4) generate code to deal with special constructs such as history substates.

Declarations: Information about the following kinds of signals is stored in the annotated AND-OR tree and these are declared at each node while generating code for the module corresponding to that node.

1. External Input signals.
2. Internal Input events generated during transitions out of substates of parent node A.
3. Internal Output events(actions) generated during transitions out of substates of parent node A.
4. If A is a substate of an Or-state with history, then a valued signal $new\_history\_A$ is used so that the history can be changed appropriately whenever transition to a substate A_i of A takes place.
5. Dummy signals for T2 or T4 transitions that enter A: In this case signals of the form sig_BtoA or sig_AtoB would be needed, where B is either an immediate parent or an immediate child of A. This list is built up for each such node A, during the *Node -Labelling* Algorithm. These signals are used to build a chain of signals that trigger transitions between immediate parent-child states, and the whole chain generates the entire transition.
6. Dummy signals for T3 or T4 transitions that exit A. Similar to 5 above.

7. Valued History signals for all Or-sub-states having history; for each such OR-state these store the value of the most recent substate. While building the AND-OR tree we can maintain a list of Or-states which have history.
8. Signals that indicate transition to a history[4] substate of a substate of A, or if A is an Or-state, to indicate transition to history substate of A.
9. Characteristic signals (in, enter, exit) for each substate of A. To generate this list, traverse the AND-OR tree bottom-up (postorder) and at each node, add to a list of child nodes. Then while generating code for node A, declare all characteristic signals for each of its child nodes as listed.

We have a new module only for each OR-node, therefore, we need not keep a list of all nine types of signals with an AND-node or BASIC-node unless it is the ROOT node.

The *STEP* signal: In the ESTEREL code generated, each step occurs on receipt of an external signal called $STEP$. This signal is needed to provide a tick on which transitions can be made even when there are no input signals from the environment (i.e. when all triggering events are internally generated). Use of STEP is necessary to implement the super-step semantics of STATEMATE, wherein several steps are executed starting with some initial triggering events, till the system reaches a set of stable states (i.e., states with no enabled transitions out of them).

Transitions: Consider code generation for the translation for a transition t of type T, with source state A and target state B.

In brief, the translation involves the following : (1) Generate code to await the occurrence of the triggering event, and, (2) on occurrence of the STEP (as in STATEMATE semantics), if the triggering condition is true and no transition pre-empts t, emit : (a) a signal to activate the next state (called a "go" signal), (b) a signal to activate a chain of transitions (for types T2 through T4), (c) signals to exit the current state, i.e., to terminate emission of signals that depict the current state as active.

Figure 6 illustrates translations with respect to T4 transition.
The complete procedure `translate-transition` is given in Appendix A. The parameter curr_node is the node for which we are generating code.
Note: For lack of space, we give only snippets of the most essential parts of the ESTEREL code. The full code generated is well formed and syntactically correct.

Code-Generation Algorithm: In the following, we describe the basic-code generation algorithm. Code to be emitted for immediate states like *history* and special actions are omitted for brevity.
Notation: In the code-generation algorithms, algorithm details are in Roman font while the code is boxed in `Typewriter font`.

[4] We have implemented *deep-history* as a sequence of transitions between history states. Such signals are used to make ϵ transitions between history states.

Algorithm 1 *Basic Code-Generation Algorithm: The main algorithmic steps are sketched below.*

1. Traverse the AND-OR tree top-down. (in preorder)
2. For each node A do
   - If A is an OR-node:

   (a) Begin a new module, and declare all signals that occur A's signal list, or in the signal list of child nodes of A, till the first child Or-node is encountered.

   (b) Generate code for each block representing the substate of A. Let A_1, A_2, ..., A_n be the immediate child nodes of node A. Let e_{i1}, e_{i2}, ... e_{im} be the external or internal events on which transitions are made out of the A_i. Let the corresponding actions be act_{i1} to act_{im}. Further, let the conditions under which the transitions are to be taken be C_{i1} to C_{im}. Let the list of *hide* signals for the nodes $A_i, \forall\, i$ be $hide_1$ to $hide_t$. $STEP$ is a signal that indicates that the next step must be performed. It is an external signal. Steps of the translation are described below:

   Step 1. Emit *preamble* code. If A is a substate of an OR-state B with history, then appropriate *newhist* signals are emitted to update history. Code to be emitted *from this step* is given below:

```
emit enter_A;
[ trap T'_A in [
            sustain in_A;
          || [ await tick; emit newhistB(A); ]
          || [ signal goA1, goA2, ..., goAn in [
          ··· % to be completed in other steps
```

   Step 2. Emit code to check for T2 and T4 transitions, or for transitions to the history substate of A. If none of these are true then default state is entered. Code *from this step* is given below:

```
present
    case sig_AtoAj do
          emit goAj % This is repeated for each sig_AtoAj%
    case enterhist_A do
          [ if histA = 1 then
                   emit goA1 %  Check for each i%
            elseif histA = 2 then
                   emit goA2
            else emit goAk % Ak is the default substate
                         for A%
            end if
end present;
```

   Step 3. For each i, emit code to handle transitions out of A_i and also the refinement of A_i. The code for each of the i are composed in parallel. The respective codes *to be emitted* are given in the substeps below:

Substep 1. Preamble code for A_i.

```
[ loop [
      await immediate goA_i;
      trap T_{A_i} in
        ··· % Subsequent codes will be completed by other
steps %
```

Substep 2. Emit code corresponding to the refinement of A_i. We indicate the refinement of A_i by $<< A_i >>$. If A_i is an AND-node or BASIC-node then this is the block for A_i. If A_i is an Or-node, then this is a "**run** A_i" statement. In this case, add it to a queue[5] of Or-nodes Q_{nodes}, so that we emit code for it and its child nodes later. When the node is visited during the preorder tree traversal, the entire subtree under that node A_i is skipped to be processed later.

```
 [ << A_i >>; exit T_{A_i};
|| ··· % subsequent codes will be completed by other steps %
```

Substep 3. Emit code for each transition triggered by e_{ij}, j = 1..m, and compose in parallel with the above code. i.e., [6] $\forall\ t_i \in T^i_{exit}$,

```
call translate-transition(t_i,TYPE_of_t_i,A_i);
end trap % T_{A_i}
```

Substep 4. Code emitted in case there are transitions of type T3 or T4. Thus, for all transitions t of type T3 or T4 which exit state A_i we would have:

```
call exit-code-trans(t,TYPE_of_t,A_i);
```

Substep 5. Postamble code for the substate A_i is given below:

```
] end loop
]
```

Step 4. The postamble code to be emitted is given below:

```
        end signal
      ]
    end trap % T'_A
  ]
end module
```

- If A is an AND-node:

(a) Generate code to emit enter and in signals for A, or for updating history, as in preamble code above.

(b) Generate code for each one of A's child nodes, A_i, and compose these in parallel.

---

[5] Note that queue is implicit in the underlying tree traversal algorithm.

[6] For two transitions out of the same state with the same priority, we assume some priority order known in advance and instead of composing code for these transitions in parallel, we use the `await case .. end await` construct of ESTEREL.

(c) Generate code for each transition that quits a child node of A and compose each in parallel with that in item 2 above. The translation for the individual transitions is exactly as for an Or-node. There are no looping transitions of type T4 for an AND-node.

• If A is an BASIC-node:
Generate code to emit enter and in signals for A, or for updating history of its parent state, just as was done for the Or-state. Also generate code to *begin*, *await* a return signal from or *end* an activity.

3. Generate code for each of the Or-nodes in the queue Q_{nodes} till no more Or-nodes remain in the queue.

Note: Algorithm 1 preserves the priority structure of transitions based on *scope* by appropriately nesting the *trap*s and using the ESTEREL semantics of nesting of *traps*.

Generation of *STEP* signal: In the above Algorithm 1, each step occurs on receipt of an artificially created external signal called $STEP$.

Clearly, this $STEP$ signal cannot be generated internally, as it will not generate a tick then. Further, $STEP$ must be given to the state machine (system) as long as there are enabled transitions (enabled on internally generated signals). In our translation, this indication is obtained from the *enter* and *exit* signals emitted.

We define a new signal "*give_step*" which is emitted whenever an *enter* or *exit* signal is emitted. Thus, whenever *give_step* is emitted, a $STEP$ signal must be emitted. Additionally, $STEP$ must be emitted on occurrence of an external input. The state machine generated by the ESTEREL compiler must interface with the environment through a *driver routine*. The driver routine executes the state machine whenever there is an input from the external environment. Thus, our problem is to execute the state machine under certain conditions(namely when *give_step* is emitted) even when there is no external input. The trick here (as in [11]) is to set a bit for every occurrence of *give_step* that is checked by the driver routine; the bit indicates that the driver routine must generate a tick (and supply a $STEP$)[7]. Thus, due to the presence of "await STEP" in the translation for transitions, although the actions are "activated" in the current step, they take effect only in the next step. This is in accordance with the STATEMATE semantics.

Our translation faithfully represents all behaviors of the STATEMATE Statecharts, in both the *Step* and *Superstep* time models. In our translation, the STEP of Statecharts is mapped to the *tick* of ESTEREL. Time instants are indicated by a separate TIME signal. In the *Superstep* time model, the STEP and TIME signals are distinct, while in the *Step* model they always occur together. As noted in [7], a Statechart using the *Superstep* time model can have possible infinite loops in the same TIME instant. This can also happen in our ESTEREL translation, and cannot be detected using the present ESTEREL tools.

[7] During simulation with the standard ESTEREL tool `xes`, we supply $STEP$ as an external input in much the same way as a tick is supplied.

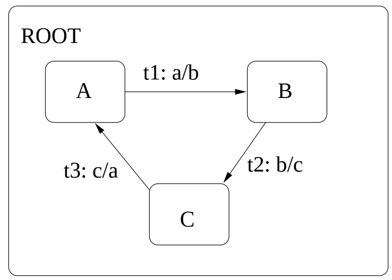


Fig. 7. Statechart with cycle

Let us consider the Statechart shown in fig. 7. Following are the steps executed when the event *a* occurs.

- STEP 1: Transition t1 is enabled because of occurrence of a and the system goes from the configuration {R,A} to {R,B} and the event *b* is generated in the system.
- STEP 2: In this step since event b is available, transition t2 is enabled and the system leaves the configuration {R,B} and goes to {R,C} and the event *c* is generated.
- STEP 3: In this step since event c is available, transition t3 is enabled

In the asynchronous time model [7], all these steps will constitute one superstep and be executed in one time instant. Each of these steps is executed when the external signal STEP is given.

It is possible to detect such loops, however, we shall not discuss it here.

3.4 History

As noted in [7], history states can occur only within Or-states. History is implemented using valued *history* signals for each Or-state having history. The value 0 corresponds to the default state, i.e. no history. The emission of the history signal for a state S, *histS* is done only by the root module ROOT, of the entire Statechart. When a new state is entered within an Or-state S, the module corresponding to that state emits a *newhistS* signal which is captured by ROOT which in turn updates *histS*. The history itself is maintained as a integer valued signal[8], the integer indicating which substate of the Or-state is the most recent one. Below, we show the code part of ROOT which updates the history values.

```
module ROOT :
...
var x in
[ % the below block exists for each Or-state with history
every immediate newhistS
```

[8] However, if we use a shared variable for keeping track of the history, there will be no need to *sustain* the integer valued signal used for that purpose.

```
   x := ?newhistS ;
   sustain histS(x) ;
end
|| ...
] ...
end module
```

3.5 Illustrative Examples

Here, we shall discuss two examples developed and verified using the above system.

Example 1. Figure 8 shows an example of the Priority Inversion problem arising due to nondeterministic behavior of the Statechart. Processes P1, P2 and P3 have priorities 1,2 and 3 respectively, and P1 and P3 share a common resource, access through which is controlled by a mutex.

It can be shown (by automata reduction) that the configuration (Root, Sys, P1blocked, P2run, P3premp) is a case of priority inversion and the system is deadlocked because of the following sequence : P3 enters critical region, P1 blocks on mutex, P2 pre-empts P3 with P1 and P3 now blocked, and thus priority of P1 and P2 has been inverted. It has been verified that this will always lead to the configuration (Root,Error). To overcome deadlock, we can add one transition between the states Error and Sys, which will again bring the system to default configuration and normal operation can resume.

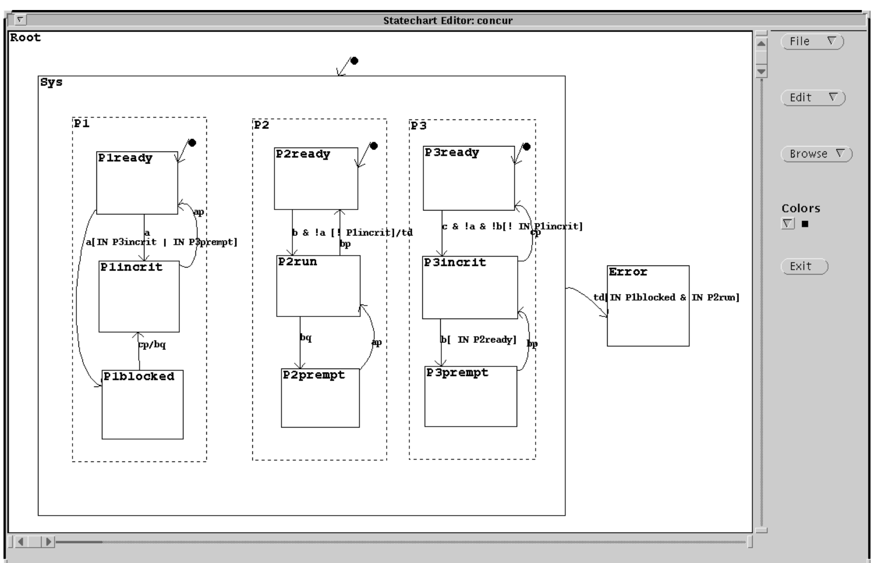


Fig. 8. Priority Inversion Example

A sample snippet of the ESTEREL code generated by our system is given in the Appendix. Note that the actual code generator slightly deviates from the abstract algorithms as it uses some implementation optimizations.

Example 2. This is the example of switching interrupts described in section 1 depicted by the Statechart shown in Figure 1.

Our translation described earlier has been applied to the Statechart shown in Figure 1 and the ESTEREL code obtained, tested, simulated and verified (using the Auto/Autograph tools). Some of specific properties that have been verified are: Event `wdt_int` occurs on system failure and when `wdt_isr` is entered, the system will toggle between states `wdt_isr` and `nmi_isr`, which is the intended behaviour. The actual code is not given for brevity.

4 Conclusions and Related Work

In this paper, we have proposed a translation from Statecharts to ESTEREL, and have applied this translation to successfully model and analyze some systems that might occur in real world problems. The translation is syntax-directed so that the translation of the entire Statecharts specification can be put together from the translation of its individual components. We have only sketched some of the algorithms for lack of space.

4.1 Related Work

An early attempt towards a graphical formalism avoiding the anomalies of Statecharts was the definition of *Argos* (see [8]). Very recently efforts have also been reported in combining *Argos* and the synchronous languages ESTERELdescribed in [5]. Another effort of translating STATEMATE Statecharts to *Signal* has been reported in [3] where the aim has been to use *Signal* (another synchronous language) and its environment for formal verification purposes. *Signal* and ESTEREL are quite different considered from the point of view of verification basis and flexibility. Our approach provides the possibility of using various automata-based/temporal logic based tools for verification in a fairly natural way.

A recent approach is that of Mikk et al.[9], in which the authors discuss the translation of Statecharts into another graphical formalism called *Extended Hierarchical Automata*(EHA). In this formulation, the inter-level transitions are eliminated, by extending the labels to include *source_restriction* and *target_determinator* sets. Our translation does something similar to the one that is resorted to for EHAs, in that we use dummy signals to make interlevel transitions, one for each level transcended. It must be noted that the translation in [9] is from one graphical language to another, ours is from a graphical language to a textual language. In a subsequent work [10], which is of a similar flavour as ours, Mikk et al. have translated EHAs into Promela for use with the model checker SPIN. This enables them to do LTL model checking on Statecharts. With our translation, we are able to use ESTEREL tools such as *FC2Tools* to do equivalence checking, checking for deadlock, livelock and divergent states; and *Hurricane*, which does verification of LTL safety properties. We also support special Statechart features such as timing primitives and history.

Another approach taken with the spirit of providing an integration of *Argos* and ESTEREL has been the work on *SyncCharts* reported in [1] and [2]. SyncCharts have a precise semantics, and is translatable to ESTEREL. It does not allow for inter-level transitions, and history, which are very useful features of Statecharts, and which are part of STATEMATE Statecharts (which we have worked with). SyncCharts however has explicit syntactic constructs for preemption such as suspension, weak abortion, and strong abortion, much like ESTEREL. The semantics of these constructs is the same as that of corresponding constructs in ESTEREL. Unlike such an approach of having another language, our aim has been to translate the existing Statecharts that is pragmatically very attractive and used widely, into an existing framework that permits formal verification. We have illustrated how the behaviours of a large class of Statecharts can be captured through the use of the unifying threads that run through the semantics of synchronous languages, both textual and graphical. Also, our aim has not been to define yet another semantics of Statecharts. Our goal has been to show how a class of Statecharts which have constructs like inter-level transitions and global communication of events, and which is used in the industrial strength tool STATEMATE, can be translated to a textual synchronous language and formally verified.

4.2 Summary of Work

We have translated Statecharts to ESTEREL version 5.10 described in [4] and a prototype system is in use. We have been using the tools of ESTEREL verification such as *FC2tools* based on bisimulation and *Hurricane* (from INRIA/CMA); we are also working on integrating the system with a tool being developed here by Paritosh Pandya on generating synchronous observers from DC specification of properties. A spectrum of industrial scale examples have been verified using ESTEREL and our translation will help combine ease of specification with this advantage of verification. The system implemented has shown that it is possible to integrate the advantages of Statecharts and ESTEREL in the design of reactive systems. While it is true that STATEMATE Statecharts and ESTEREL have different semantics, our translation works for a subset of deterministic Statecharts, and using a subset of ESTEREL constructs in a limited manner. We have thus maintained STATEMATE semantics while restricting the class of Statecharts we translate. The current translation also considers only simple events and actions; work is in progress to extend this to more general events and actions.

To sum up, this work is interesting from many standpoints. Considered from view of Statecharts, we have found it useful as a way to incorporate formal verification and as a diagnostic tool for detecting nondeterminism. From the point of view of ESTEREL, it provides an integration of textual and graphical formalisms. From a practical perspective, it is possible to use heterogeneous systems such as Statecharts and ESTEREL together in the development of reactive systems and the use of the industrial strength STATEMATE shows this work has potential worth in industrial system verification. There has been a large effort in integrating synchronous languages such as ESTEREL, Lustre and Signal. This work has attempted to bring Statecharts in a restricted way under this umbrella. The prototype has been built and found to be effective in the design of small-scale reactive systems. Experiments are going on in the development of large complex systems using the system.

Acknowledgments

We thank the anonymous referees for their valuable suggestions and comments. The work was initiated while Sanjit Seshia was with I.I.T., Bombay working on a summer project with R.K. Shyamasundar at TIFR, Bombay. He thanks I.I.T. Bombay, TIFR and CMU for their support. R.K. Shyamasundar thanks IFCPAR, New Delhi for the partial support under the project 2202-1.

References

1. André, C. SyncCharts: A Visual Representation of Reactive Behaviors. Tech. Rep. RR 95-52, I3S, Sophia-Antipolis, France, 1995.
2. André, C. Representation and Analysis of Reactive Behaviors: A Synchronous Approach. Tech. Rep. 96-28, Université de Nice, Sophia-Antipolis, France, 1996.
3. Beauvais, J.-R. et al. A Translation of Statecharts to Signal/DC+. Tech. rep., IRISA, 1997.
4. Berry, G. A Quick Guide to Esterel Version 5.10, release 1.0. Tech. rep., Ecole des Mines and INRIA, February 1997.
5. Berry, G., Halbwachs, N., and Maraninchi, F. Unpublished note on Esterel and Argos. 1995.
6. Harel, D. Statecharts: A Visual Formalism for Complex Systems. *Science of Computer Programming 8* (1987), 231–274.
7. Harel, D., and Naamad, A. The STATEMATE Semantics of StateCharts. *ACM Transactions on Software Engineering and Methodology 5*, 4 (October 1996).
8. LeGuernic, P. et al. Programming Real-time applications with Signal. *Proceedings of the IEEE 79*, 9 (September 1991), 1321–1336.
9. Mikk, E., Lakhnech, Y., and Siegel, M. Hierarchical automata as model for statecharts. In *LNCS* (Dec. 1997), 1345, pp. 181–197.
10. Mikk, E., Lakhnech, Y., Siegel, M., and Holzmann, G. Implementing Statecharts in Promela/SPIN. In *Proc. of the 2nd IEEE Workshop on Industrial-Strength Formal Specification Techniques* (1999), IEEE Computer Society.
11. Puchol, C. et al. A Formal Approach to Reactive Systems Software: A Telecommunications Application in Esterel. In *Proc. of the Workshop on Industrial Strength Formal Specification Techniques* (April 1995).

Appendix A: Code Generation for Transitions

This procedure gives the translation for a transition t of type T, with source state A and target state B. curr_node is the node which we are generating code for. As mentioned before, the algorithm details are in Roman font while the emitted code is boxed and in `typewriter` font.

procedure **translate-transition (t,T,curr_node)**
begin
A := source(t); B := target(t);
e_t := event(t); a_t := action(t);
C_t := condition(t);
/* Let hideS be signal corresponding to transition t which hides other transitions of scope less than that of t. let t be hidden by signals $hide_1$, $hide_2$, ..., $hide_n$.*/
if (A = curr_node) then
begin
EMIT :-

```
loop
  await (immediate) e_t;
  (await STEP;)
  if C_t then [
    present hide_1 else
     present hide_2 else
      ...
      present hide_n else
       emit hideS;
       emit a_t; emit exit_A;
```

if (T = T1) then
begin
EMIT :-

```
        emit goB; exit T_A;
```

end
if (T = T2 OR T = T4) then
begin
/* Let S_1, S_2, ..., S_n be intermediate states between A and B. */
EMIT :-

```
        emit sig_AtoS_1;
        exit T_A; % exit trap
```

end
EMIT :-
/* Complete all `present` statements */

```
      end present
    ...
   end present
 ] end if ]
 end loop
```

end /* if A==curr_node */
else /* if A $\neq$ curr_node, i.e., t of type T2 or T4 */
begin
EMIT :-
/* Let A_1 and A_2 be the two immediate child nodes of A */

```
present sig_AtoA_1 then
  emit sig_A_1toA_2;
end present;
```

end
end procedure

Note: Above, we have assumed that the condition is a boolean expression, as in Esterel syntax. If the condition is the test of presence of a signal it must be replaced by a

```
present SIG then ...
  else ...
end present
```

translation. If the condition involves testing values of shared valued signals, which could possibly change value “simultaneously”, then we need to ensure that the value tested is the one at the time of occurrence of the triggering event. This code is omitted for brevity.

Further, for transitions of type *T3* and *T4*, on exiting a state, code must be emitted for continuing the chain of transitions. This code generates signals that trigger transitions in child states. The code generation routine for this is referred to in Algorithm 1 as *procedure* `exit-code-trans (trans, Type-of-transition, States)`. We omit the detailed description of this routine in this paper.

Appendix B: Esterel Code : Priority Inversion Problem

Below we attach code snippets for the states P1 and P1ready.

```
module P1ready:
% Signal Declarations
output EnterP1ready, InP1ready,
ExitP1ready;

% Program Body --------------
emit EnterP1ready;
do sustain InP1ready watching
ExitP1ready;
end module
%-----------------------------
module P1:

% Declarations deleted for brevity

signal goP1ready, goP1incrit,
goP1blocked, goP1, goP2ready,
goP2run, goP2prempt, goP2,
goP3ready, goP3incrit, goP3prempt,
goP3,goSys in [

% Program Body --------------
emit goP1ready;

emit EnterP1;
do sustain InP1 watching ExitP1;
||
loop
await immediate goP1ready;
 trap outP1ready in
    run P1ready;
 ||
 loop
   await % Exit
    case immediate [a] do
      present InP3incrit or InP3prempt
      then [
      % Testing condition
        present HideSys then
         await STEP
        else [
         await STEP;emit ExitP1ready;
         emit goP1blocked;
         exit outP1ready;]
         end ]
       else [
         present HideSys then await STEP
         else [
           await STEP; emit ExitP1ready;
           emit goP1incrit;
           exit outP1ready ]
         end % end present
         ]
       end % end present
    end % end await
  end % end loop
end % end trap
end % end loop
||
loop
await immediate goP1incrit;
trap outP1incrit in
run P1incrit;
||
loop
 await % Exit
 case immediate [ap] do
  present HideSys then await STEP else [
    await STEP; emit ExitP1incrit;
    emit goP1ready; exit outP1incrit ]
  end % end present
 end % end await
end % end loop
end % end trap
end % end loop
||
loop
await immediate goP1blocked;
trap outP1blocked in
run P1blocked;
||
loop
 await % Exit
 case immediate [cp] do
  present HideSys then await STEP else [
    await STEP; emit bq; emit ExitP1blocked;
    emit goP1incrit; exit outP1blocked; ]
  end % end present
 end % end await
end % end loop
end % end trap
end % end loop
] end % end signal
end module
```

Appendix C
Intuitive Semantics of ESTEREL

At each instant, each interface or local signal is consistently seen as present or absent by all statement, ensuring determinism. By default, signals are absent; a signal is present if and only if it is an input signal emitted by the environment or a signal internally broadcast by executing an `emit` statement.

To explain how control propagates, consider first examples using the simplest derived statement that takes time: the waiting statement "`await S`", whose kernel expansion "`do halt watching S`" will be explained later. When it starts executing, this statement simply retains the control up to the first future instant where `S` is present. If such an instant exists, the `await` statement terminates immediately; that is the control is released instantaneously; If no such instant exists, then the `await` statements waits forever and never terminates. If two `await` statements are put in sequence, as in "`await S1; await S2`", one just waits for `S1` and `S2` in sequence: control transmission by the sequencing operator ';' takes no time by itself. In the parallel construct "`await S1 || await S2`", both await statements are started simultaneously right away when the parallel construct is started. The parallel statement terminates exactly when its two branches are terminated, i.e. when the last of `S1` and `S2` occurs. Again, the "||" operator takes no time by itself.

Instantaneous control transmission appears everywhere. The nothing statement is purely transparent: it terminates immediately when started. An " `emit S` " statement is instantaneous: it broadcasts `S` and terminates right away, making the emission of `S` transient. In "`emit S1; emit S2`", the signals `S1` and `S2` are emitted simultaneously. In a signal-presence test such as "`present S ...`", the presence of `S` is tested for right away and the `then` or `else` branch is immediately started accordingly. In a "`loop stat end`" statement, the body stat starts immediately when the loop statement starts, and whenever *stat* terminates it is instantaneously restarted afresh (to avoid infinite instantaneous looping, the body of a loop is required not to terminate instantaneously when started).

The `watching` and `trap-exit` statements deal with behavior preemption, which is the most important feature of Esterel. In the watchdog statement "`do state watching S`", the statement stat is executed normally up to proper termination or up to future occurrence of the signal `S`, which is called the guard. If *stat* terminates strictly before `S` occurs, so does the whole `watching` statement; then the guard has no action. Otherwise, the occurrence of `S` provokes immediate preemption of the body *stat* and immediate termination of the whole watching statement. Consider for example the statement

```
do
    do
       await I1; emit O1
    watching I2;
    emit O2
watching I3
```

If `I1` occurs strictly before `I2` and `I3`, then the internal `await` statement terminates normally; `O1` is emitted, the internal watching terminates since its body terminates, `O2` is emitted, and the external watching also terminates since its body does. If `I2` occurs before `I1` or at the same time as it, but strictly before `I3`, then the internal watching preempts the await statement that should otherwise terminate, `O1` is not emitted, `O2` is emitted, and the external watching instantaneously terminates. If `I3` occurs before `I1` and `I2` or at the same time as then, then the external watching preempts its body and terminates instantaneously, no signal being emitted. Notice how nesting watching statements provides for priorities.

Now the translation of "`await S`" as "`do halt watching S`" will be clear. The semantics of halt is simple: it keeps the control forever and never terminates. When `S` occurs, `halt` is preempted and the whole construct terminates. Note that `halt` is the only kernel statement that takes time by itself.

The trap-exit construct is similar to an exception handling mechanism, but with purely static scoping and concurrency handling. In `trap T in stat end`, the body `stat` is run normally until it executes an `exit T` statement. Then execution of `stat` is preempted and the whole construct terminates. The body of a `trap` statement can contain parallel components; the `trap` is exited as soon as one of the components executes an `exit T` statement, the other components being preempted. However, `exit` preemption is weaker than `watching` preemption, in the sense that concurrent components execute for a last time when `exit` occurs. Consider for example the statement

```
trap T in
   await I1; emit O1
        ||
   await I2; exit T
end
```

If `I1` occurs before `I2`, then `O1` is emitted and one waits for `I2` to terminate. If `I2` occurs before `I1`, then the first branch is preempted, the whole statement terminates instantaneously, and `O1` will never be emitted. If `I1` and `I2` occur simultaneously, then both branches do execute and `O1` is emitted. Preemption occurs only after execution at the concerned instant: by exiting a `trap`, a statement can preempt a concurrent statement, but it does leave it its "`last wills`". The rule for exiting from nested traps is simple: *only the outermost trap matters, the other ones being discarded.* For example, in

```
trap T1 in
        trap T2 in
             exit T1
            ||
             exit T2
        end;
        emit O
end
```

traps `T1` and `T2` are exited simultaneously, the internal trap `T2` is discarded and `O` is not emitted. Traps also provide a way of breaking loops, which would otherwise never terminate as reflected by:

```
trap T in
   loop ... exit T ... end
end
```

One can declare local variables by the statement

```
var X in stat end
```

Variables deeply differ from signals by the fact that they cannot be shared by concurrent statements. Variables are updated by instantaneous assignments "`X`:=*exp*" or by instantaneous side effect procedure calls "`call P(...)`", where a procedure `P` is an external host language piece of code that receives both value and reference arguments.