A Graphical Environment for the Specification and Verification of Reactive Systems

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

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

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

Abstract. In this paper, we describe the design and implementation of an environment for the specification, analysis and verification of reactive systems. The environment allows the user to develop specification in the graphical formalism of Statecharts [1] and a built-in translator tool translates the specification into ESTEREL [3] program. Through such an approach, we have been able to integrate the powerful graphical formalism of Statecharts, which is very appealing to engineers, and the power of formal verification environments for ESTEREL. Since we translate Statecharts, which can be nondeterministic, to ESTEREL programs which are fully deterministic, the system overcomes the nondeterminism in the specifications by enforcing priority. The behaviour of ESTEREL programs generated by the translator follows the Statechart *step* semantics [2]. In the paper, we describe the main components of the environment, the principles underlying the translation and illustrate the use of the system for the specification and verification using an example.

1 Introduction

Significant amount of research has been done in the last decade in the methods for design and development of reactive systems. The class of synchronous languages and the class of visual formalisms are two approaches that have been widely accepted and used for the design, analysis and simulation of reactive systems both in academia and industry. The family of synchronous languages are based on the *perfect synchrony hypothesis* which can be interpreted to mean that the program reacts rapidly enough to perceive all the external events in order and produces all the output reactions before reacting to a new input event. Typical examples of embedded control systems can be abstracted in this manner. Some of the prominent languages of this family that have found wide usage in the industry include ESTEREL , Lustre, Signal etc. In this work we have selected

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

ESTEREL due to range of features available which include the availability of rich primitives for concurrency, communication and preemption, a clean and rigorous semantics, a powerful programming environment with the capability of formal verification. The advantages ESTEREL 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 state automata and supporting features such as hierarchy, orthogonality (concurrency) and broadcast communication between system components. Being a formalism rather than a language, there have been a plethora of semantics corresponding to various plausible implementations. The standard semantics that is normally used (we also use the same) is that of Harel & Naamad [2]. Even though there are powerful programming environments for Statecharts such as STATEMATE [1] including simulators, code generators etc, in general they lack formal verification tools.

Textual and graphical formalisms have their own intrinsic merits and demerits as explained in the following. First, let us consider a reactive system described below, a Statechart representation for which is shown in Fig. 1.:

> The controller required specifies control flow (switching of tasks) among various computing tasks and interrupt service tasks. The computing states (`hpt, dt1, dt2` etc.) switch from one to another in a cyclic fashion and are shown as substates of `compute_proc`. The interrupt service states (`rti_isr, wdt_isr, net_isr, nmi_isr`) are entered as a result of the occurrence of interrupt events. The history notation has been used to indicate that on return from interrupt states, the system returns to last executing compute state (except when event `100ms` occurs, the control returns to computing state `hpt`). The event `wdt_int` occurs on system failure and it has to be verified that when `wdt_isr` is entered, the system will toggle between states `wdt_isr` and `nmi_isr`, which is the intended behavior.

Intuitively, one can understand the Statechart in Fig. 1 and its correspondence with the English description above. The above system can be realized in ESTEREL as well. However, writing the program in ESTEREL is somewhat cumbersome than drawing the Statechart shown in Fig. 1. In our experience and opinion, some classes of programs (such as the above) can be modelled/programmed naturally using visual formalisms. Further, visual formalisms are appealing to practicing software designers. Arguing the formal correctness from programs as shown in Fig. 1 however, is quite complex particularly when the number of states are large and hence, the need verification support. In terms of the verification environment, ESTEREL scores over other formalisms.

In this paper, we describe the design and implementation of a programming environment PERTS, that has been designed and implemented in order to leverage the advantages of both visual formalism of Statecharts and simulation &

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

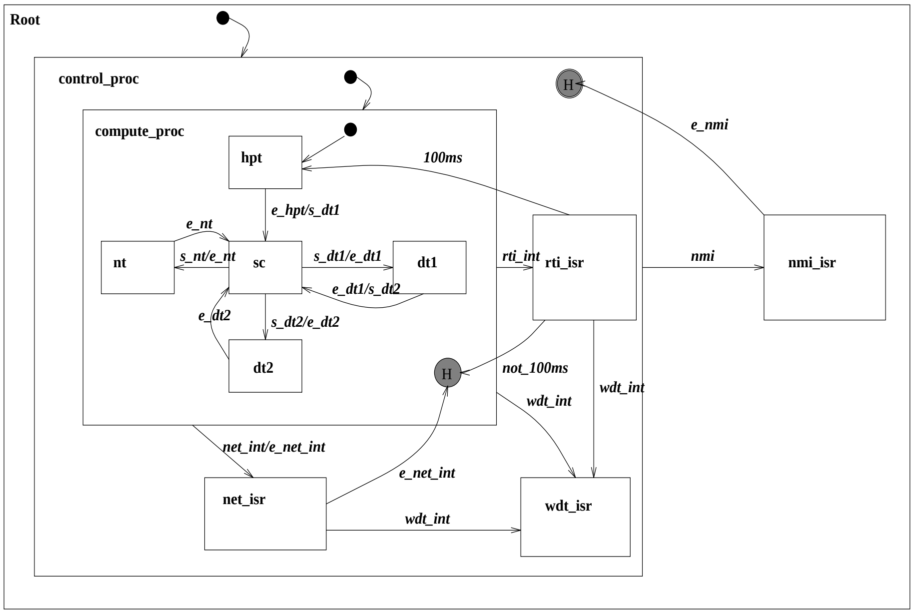

Fig. 1. Example of Switching Interrupts

verification support offered by ESTEREL environment. PERTS environment allows the user to create a model of the software using the user-friendly formalism of Statecharts. The specification is then translated into ESTEREL which then can be subjected to simulation and verification steps. This is possible due to a Statecharts-to-ESTEREL translator tool, STATEST incorporated in PERTS. The translation preserves the correspondence between the Statechart states and the ESTEREL program components for aiding debugging of the programs and verification of the properties at different levels. The primary motivation incorporating the Statecharts-to-Esterel translator tool in PERTS was to open up possibilities of formal verification which seemed easily possible via ESTEREL route, the code generation/optimization capability coming as a bonus. In fact, the implementation of the tool has found to have the important advantage of using Statecharts or ESTEREL for the specifications/modeling of different components of the same system.

In the following, we describe the components of the PERTS environment with a special emphasis on the translation tool i.e., STATEST. The rest of the paper is organized as follows. Section 2 gives an overview of PERTS environment and its main components. In section 3, we describe the translation from Statechart programs to ESTEREL programs as implemented in STATEST. Verification is discussed in section 4 followed by a discussion of related work in section 5.

2 Overview of PERTS

Figure 2 shows the components of the PERTS environment. Presently it consists of a Statechart Editor (SCE) and a translator tool (STATEST) to translate Statecharts into Esterel. The translator forms the link between PERTS and the Esterel environment containing the simulation and verification tools.

The user is required to capture the behaviour of the reactive component using Statecharts notation supported by the SCE. Once the Statecharts specification is ready and is syntactically correct, it can be translated into the ESTEREL using STATEST. The ESTEREL code generated can be used for simulation by using the tool *Xes*. The verification steps can be carried out using tools like *FC2TOOLS* [6], *AUTO/AUTOGRAPH* [7] or Esterel model checker *TEMPEST*[2]. The verification support provides the designer an added level of confidence. The third component SIMSTAT (currently under development) will support simulation of Statecharts based on the same *Step* semantics.

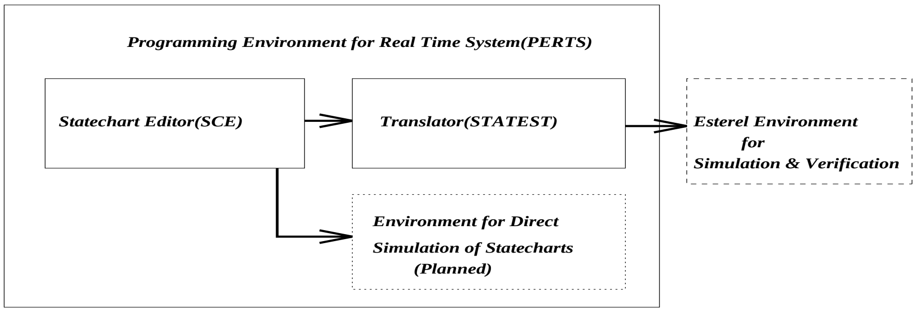

Fig. 2. Components of PERTS

2.1 SCE: Statechart Editor

Figure 3 shows the main window of the Statechart editor SCE along with the expanded *Edit* option menu, that gives the list of the edit functions available to the user. The drawing panel shows a Statechart drawn using the editor. The Statechart Editor SCE provides other useful facilities for storing and retrieving diagrams, loading and browsing previously developed diagrams and for choosing colours for drawing new diagrams. A zoom-in and zoom-out capability to view refinements at various levels is also provided.

Statechart Editor (SCE) allows user to construct Statecharts using the basic building blocks constituting the Statecharts notation. These basic building blocks are *basic states* and *super-states* (OR and AND states) denoted by labeled rectangles, *states with memory* denoted by symbol H, *default transitions* denoted

[2] ESTEREL model checker from University of Texas at Austin

by unlabeled arrows with a dot, *transitions* between states denoted by labeled arrows and separate Statecharts showing *refinements* of a given state. When the system is in an OR-state e.g. state S in Fig. 3, it would only be in one of it's immediate sub-states *S1, S2 or S3*. The AND-state e.g. state S2, has sub-states that are related by "and", meaning that when the system is in S2, it would be in all of its immediate sub-states i.e. S21 & S22. AND-state is the mechanism by which concurrent states can be represented. BASIC-states are those at the bottom of the state hierarchy, i.e., those that have no sub-states. The transitions in Statecharts are shown by arrows and they take the system from one set of occupied states to another. The general syntax of an expression labeling a transition in a Statecharts is "e[C]/a", where e is the event that triggers the transition, C is condition that guards the transition from being taken unless it is true when the event e occurs, and a is an action that is carried out if and when the transition is taken. Only the OR-states can have memory and when such states are entered the transition ends on a substate that was last occupied. Figures 1, 3 & 5 show typical Statechart specifications illustrating use of various constructs.

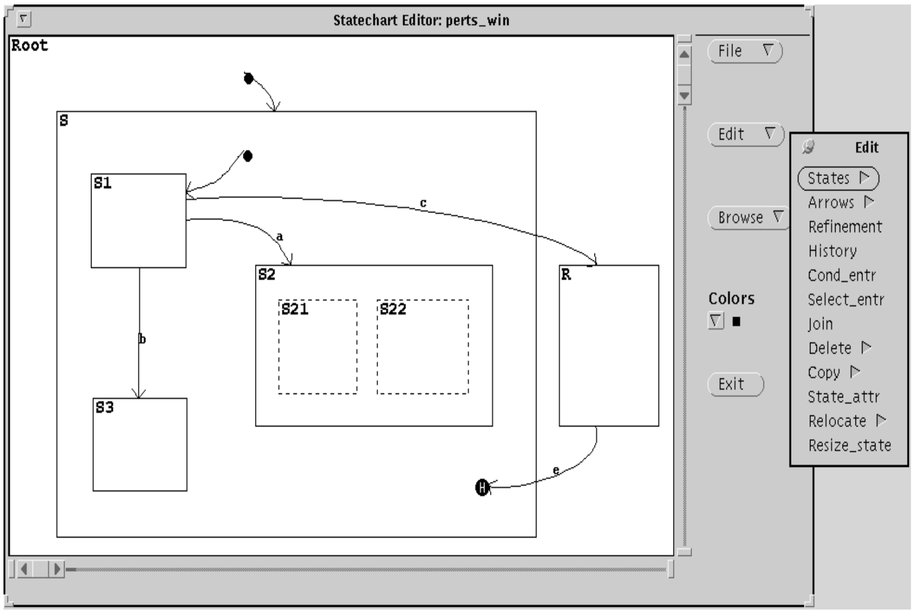

Fig. 3. Main window of Statechart Editor (SCE)

Statechart Editor builds a database, in the background, for use by other tools. In this database a Statechart diagram is represented in the form of a tree and a set of transitions. The Statechart editor is described in details in [5].

3 STATEST: Translator from Statecharts to Esterel

The internal stages of the translator are schematically shown in Fig.4. The first stage is the preprocessing tool **stpp**. The **stpp** module converts syntactical constructs like join points, conditional points etc. into a simpler but equivalent notation. These constructs are designed to provide a crisper notation while drawing Statecharts but can be simplified to simple transitions. This helps in simplifying the implementation of translator which can then expect fewer constructs in the input. It also checks for violation of Statechart syntax and gives warning messages.

The second stage of the translator is the functional form generator **stffg**. The output of **stpp** is fed into the functional form generator **stffg**, which converts the graphical notation into a textual form for further translation. This textual form is described by a context free grammar and hence amenable to rigorous syntax checking and translation. The third stage **stgen** is the ESTEREL code generator. It takes the functional form as input and emits ESTEREL code.

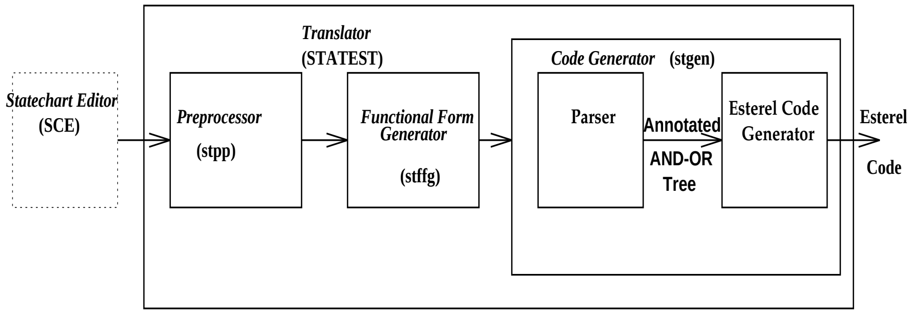

Fig. 4. Schematic Diagram of the Translator

3.1 ESTEREL Code Generator

The ESTEREL code generator is the most complex stage of the translator and its internal structure is shown in the Figure 4. Here we present a brief description but the details along with all translation algorithms can be found in [8, 15]. It consists of a parser which parses the input Statechart (in the functional form) and constructs an annotated AND-OR tree by a syntax directed translation scheme. This AND-OR tree is the input to the code constructor.

Translating a "state" in Statechart into ESTEREL means generating a code segment in ESTEREL which will represent the occupied state in Statechart. Such a code segment in ESTEREL , representing an occupied state, has to repeatedly perform ("sustain") the actions expected be carried out in that state, while waiting for events which will cause that code segment to be terminated and other code segment (a new state) to be entered. Such code segments linked together by ESTEREL statements which watch for the events in the system and pass on control from one code segment to other (as per the specified transitions in Statechart) constitute the full translation of Statechart program to ESTEREL program. In ESTEREL code, the transitions are implemented by *await* statements. In principle it is possible to construct a single module in ESTEREL representing the entire Statechart behaviour. However such a monolithic ESTEREL code is difficult to understand and also does not permit visualization of occupied/unoccupied states when simulated using ESTEREL simulator *Xes.* In the code generation strategy described here, one ESTEREL module is generated for each state in the input Statechart. This results in the generation of modular ESTEREL code capturing the behaviour of the Statechart. The only disadvantage of this scheme is that the code generated contains large number of global signals which are seen on the simulator panel.

Figure 5 shows a Statechart and its AND-OR tree alongside. Each node of the AND-OR tree represents the corresponding state in the Statechart and is annotated with the information like name, type, set of all transitions which enter and exit this state, history flag indicating whether history exists for this state etc.

The annotated And-Or tree is the central data structure for the code generation phase. The code generation algorithm traverses the tree in reverse postorder visiting all the children nodes from left-to-right before visiting the parent node. While visiting each node, the ESTEREL module for each node (State in Statechart) is emitted and the signals required to interface various modules are collected. This list is pushed up the tree for the purpose of global signal declarations in each related module. In the following, we explain the structure of ESTEREL code generated for different types of states in Statechart.

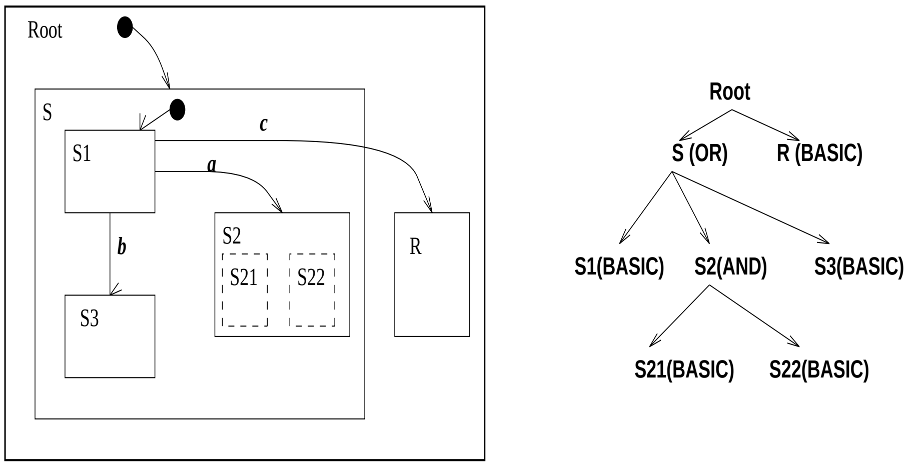

Fig. 5. Statechart and its AND-OR tree

Types of Transitions:

Transitions are important in generating ESTEREL code of each module and are divided into four types as follows:

- Type 1: Transitions between immediate children of same Or-state
- Type 2: Transitions between child to either it's parent or any ancestor
- Type 3: Transitions between ancestor to any of the child state
- Type 4: Transitions between states not governed by the above three rules (inter-level transitions).

In addition to these types of transitions, there is another special type called Loop-type. A transition w.r.t a superstate falls into this type if that transition takes place between two child states (not immediate) of that superstate and the last exited state is one of the immediate children of that superstate. With reference to the state Root in Fig. 5, the transition labeled with event c is of Loop-type.

Restrictions on Transitions

The current implementation of the translator restricts the syntax associated with the transition label *e[C]/a* as follows.

- Label *e* can be a single event or a boolean combination of events
- Guard *C* can be checking for an internally generated event like IN(S) (which is true only if the current configuration includes state S) or their boolean combinations.
- Action a can only be generation of events .

Priority of Transitions and their Implementation:

Due to nondeterminism permitted in Statecharts multiple transitions can be enabled due to the simultaneous happening of different events.Two transitions are said to be in conflict if there is some common state that would be exited, if any one them were to be taken. The decision as to which transition to take is resolved by priority of transitions. To decide the priority of transitions we use the scope rule [2]. The priority can be determined statically by traversing the AND-OR tree for each transition. We have implemented the priority by introducing a "hide" signal, emitted by a higher priority transition, which is tested while taking all lower priority transitions. Whenever "hide" signal is present all enabled lower priority transitions are blocked. In another type of nondeterminism, two transitions of same priority can be in conflict. Resolution of this type of nondeterminism is explained later.

Structure of ESTEREL Code:

We illustrate the structure of the generated ESTEREL code for each type of states from Statechart. Each state when translated is a module in ESTEREL .

Translation of a Basic state: If A is Basic-state, then the ESTEREL code generated for state A has the form

```
module A :
          emit EnterA;
          do sustain InA watching ExitA;
end module
```

Both "EnterA" and "InA" are global signals. The first statement broadcasts an internal event *EnterA* of entering the state A. The *do sustain InA watching* statement simulates the effect of being in state A by ensuring that while the state A is occupied the *InA* signal is emitted every instant and is available for other states for testing if required. The *do .. watching ExitA* construct ensures that as soon as the signal *ExitA* is emitted, the statement *sustain InA* , terminates in the same instant (strong preemption)

Translation of a Super-state: While translating a superstate of a Statechart, the following has to be implemented.

1. Entry to this superstate
2. Emission of signals to cause entry to substates. The substates may be entered as direct destinations or as a history or as defaults, if the first two are absent.
3. All transitions exiting the substates. Transitions of type 4 are broken into multiple transitions of type 2(child to parent) and type 3(parent to child), happening together in one step.
4. Transitions of loop-type w.r.t this superstate.

Or-State Translation: If S is an OR-state with substates, say, S1, S2,S3 ... Sn, then the structure of the ESTEREL module has the following structure. The || symbol represents parallel execution of three code blocks in ESTEREL syntax.

```
module S :
        Block_1 || Block_2 || Block_3
end module
```

`Block_1:`This block in the beginning contains statements broadcasting two signals *EnterS* and *InS* signifying entry to superstate S. A local signal ("go" prefixed to the name of the immediate substate to be entered) is then emitted to cause entry to one of the substates. The signal is captured in `Block_2.`

`Block_2:` This block of ESTEREL code contains n parallel segments, n being equal to the number of immediate substates, one segment corresponding to each of the substates. This is of this form

```
Code_for_S1 || Code_for_S2 || .. || Code_for_Sn
```

Each of the segments contain ESTEREL statements for entering the module of each substate as well as code for all transitions exiting that substate.

`Block_3:` This block of the module handles all inter-level transitions which are of Loop-type w.r.t. this state. This transition is similar to type 1 transition (sibling to sibling) except that the transition originates from deep within one sibling and may terminate deep within another sibling. During a transition this superstate

which is the common ancestor of the source and destination state of the transition receives a valued signal from immediate child state exited due to the transition and emits a signal causing entry to the destination state. The emitted signal is simple "go" if the destination is immediate child. Otherwise entry is caused to the ancestor of the destination state, by emitting a valued signal prefixed with "sig_D", the value being the numerical identifier of the destination state. All of the interposing superstates will relay this signal to their substates till the final destination state is reached.
And-state Translation: If S is an And-state with substates S1, S2, ...Sn , the structure of the ESTEREL code has the following form:

```
module S :
         Block_1 || Block_2
end module
```

In And-state the code segment `Block_1` is different from the corresponding `Block_1` code segment of an Or-state, in that, there is no emission of local *go* signal which causes a substate to be entered as in Or-state. The reason is when an And-state is entered all of its substates are entered at the same instant by definition. There is no `Block_3` because there cannot be Loop-type transitions for And-states. The `Block_2` code has n parallel segments (n= no of substates) but there is no *await immediate go..* construct as there is no corresponding emission of a go signal in `Block_1`.

Handling Nondeterminism Some enabled but conflicting transitions cannot be resolved by scope rule [2]. For example, there can be two enabled transitions leaving the same state having the same scope. This type of nondeterminism is permitted in Statechart and is expected to be handled by simulation environment either by randomly taking one transition or allowing the user to decide about which transition to take. Since ESTEREL is a purely deterministic language, the translation scheme has to prioritize the transitions. We resolve this by generating the *await case .. end* construct[3], whereby whichever "case" is positioned first is executed when more than two conflicting events occur. However in case of occurrence of a single event, the corresponding *case* statement will be executed and it will respond to that event only. All possible cases of nondeterminism are reported to the user during the translation process.

The *STEP* signal The translation scheme implements the *step* semantics of STATEMATE [2]. In Statecharts, several *steps* may be executed simultaneously due to the internal generation of events, which enable further transitions before a basic configuration is reached where no more transitions are enabled. This is termed *superstep*. Implementation of such a *superstep* causes semantic problem in ESTEREL as one *superstep* consisting of several steps (transitions) is to be executed in one ESTEREL instant. To overcome this problem, we introduce a

[3] Each case statement corresponds to an event triggering the transition

STEP signal and one *step* in Statechart(a transition, including multilevel) is mapped to one ESTEREL instant. The simulation stops for external *STEP* signal to be given, so that next enabled transition can be taken. The *STEP* signal cannot be internally generated in ESTEREL code as it would not create a new instant

In the generated ESTEREL code, when one transition emits an internal signal enabling another transition as in *superstep* semantics, that signal is actually consumed in the same instant, but the execution is broken by *await STEP* statement initiating a new instant. This is operationally equivalent to *step* semantics of Statechart.

3.2 Correspondence between Statechart and ESTEREL Programs

In the structure of the generated ESTEREL code, there is correspondence between *states* in Statechart and *modules* in the ESTEREL code and also a one to one correspondence between an *event* (triggering a transition) in Statechart and a *signal* in ESTEREL . This fact is used to intuitively link the input Statechart with the translated ESTEREL code during process of verification of properties as well as during simulation by ESTEREL tools. The characteristics of the translation is captured below. Consider an arbitrary Statechart program S and its corresponding translated ESTEREL program E. Then, we can establish the following properties from the nature of the translation process:

- For every state in S there is a corresponding code segment in E
- For every transition in S, labeled as *e[C]/A* there is a unique corresponding code segment in E that awaits for the event e, its guard C and emits the signals corresponding to action A.
- If the original program is deterministic, then the ESTEREL program generated is causality error free.
- The translation preserves the STATEMATE semantics.

4 Illustration: Verification

We illustrate how the ESTEREL code generated can be used for verification in the ESTEREL environment. Figure 6 shows the specification of a simple lift controller [9]. It is required to verify that the lift cannot move while the door is open. The only relevant events (signals in ESTEREL) are the events *lift_stop* and *door_closed* and the output events are *open_command* and *motor*. To verify the above property, we translate the Statecharts in ESTEREL and obtain the reduced automata by *FC2TOOLS* using the notion of *bisimulation* [14] . The reduced automata is shown in fig 7. The input and output events are prefixed with "?" and "!" respectively. The states are shown by concentric circles and the initial state is the one on top left corner of the figure. If we assume that the door is initially closed(implied by the initial state s0) then the door can remain open only between an emission of *open_command* and the next reception of *door_closed*, and the lift can be moving only between an emission of *motor* and the next reception of *lift_stop*. Since the two states showing the possibility of door remaining open and lift moving are exclusive of each other, the property is true.

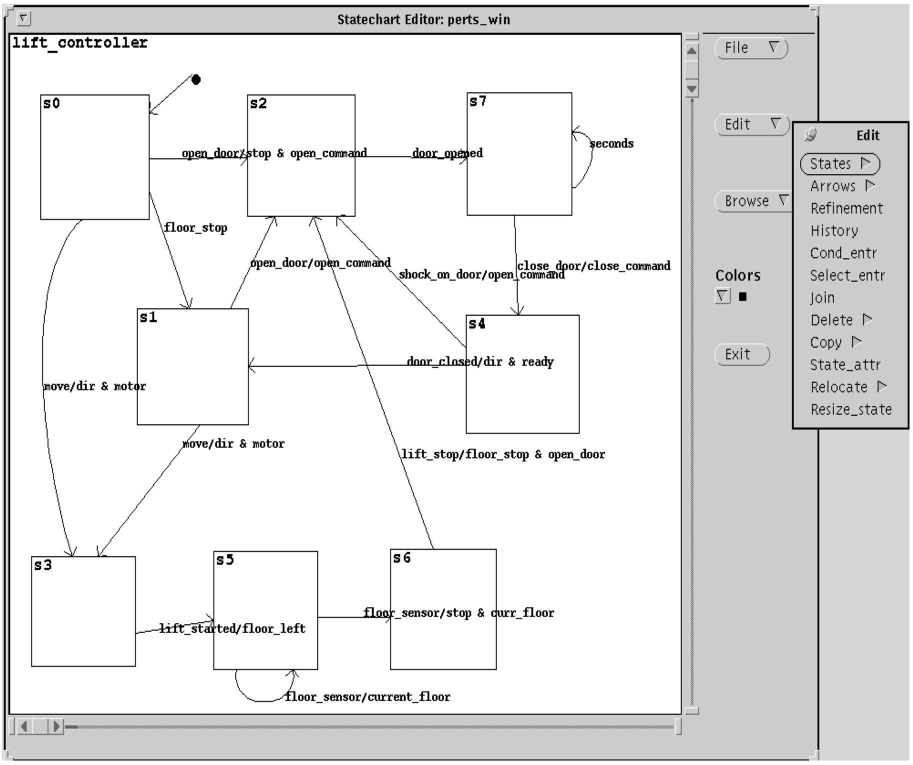

Fig. 6. Specification of a Lift Controller in Statechart

The translator described here has also been applied to the Statechart shown in Figure 1 and the Esterel code obtained, tested, simulated and verified. It is also possible to verify temporal properties of the specification using model checking tools like *TEMPEST* on the translated Esterel program.

5 Related Work and Conclusion

Recently efforts have been reported in combining Argos and Esterel [10]. The translation of Statecharts to Signal has been reported in [11] where the aim has been to use Signal for formal verification purposes. Another effort is translating Statecharts into Promela and has been reported in [12] but it is not clear whether they have taken into account of constructs like *history* of Statecharts. A recent

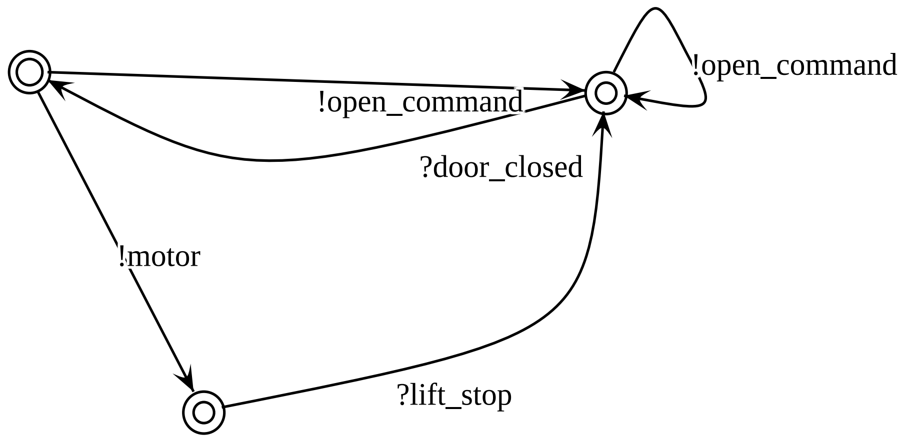

Fig. 7. Reduced Automata obtained using FC2TOOLS

approach based on the work on SyncCharts [13] was aimed at integrating Argos and ESTEREL . However SyncCharts do not allow inter-level transitions and history, which are key features of the version of Statecharts we have considered.

In this paper, we have described an environment for the specification and verification of reactive components. We have been motivated by and concerned with exploiting the advantages of the visual and textual formalisms in the development of reactive systems. The Statecharts follow STATEMATE semantics and our translation of Statechart description into ESTEREL preserves this semantics. In other words the translated ESTEREL programs, when executed would behave exactly like the Statecharts specification from which they were obtained. The prototype system has been tested on a variety of examples and is under use particularly for the design and realization of controllers. We are exploring the use of the system for large scale real-life system developments. It is planned to make this tool available through Internet.

References

1. David Harel: *Statecharts: A Visual Approach to Complex Systems*, Science of Computer Programming, 8:231-274 1987
2. David Harel, Amnon Naamad : *The* STATEMATE *Semantics of Statecharts*, ACM Transactions on Software Engineering and Methodology, Vol. 5, No. 4, Oct. 1996
3. G. Berry, G. Gonthier: *The* ESTEREL *synchronous programming languages: Design, semantics, implementation.* Science of Computer Programming, 19(2):87-152, 1992
4. G. Berry : *The Semantics of Pure* ESTEREL . Survey Paper, INRIA, Sophia-Antipolis, France.
5. A. Rai, S.D. Dhodapkar: Statechart Editor (SCE), BARC Technical Report, BARC/1996/E/012, 1996
6. Annie Ressouche et.al. *FC2TOOLS for* ESTEREL *: Verification by reduction of Synchronous Reactive Programs*, INRIA, Sophia-Antipolis, France.

7. V. Roy, R. de Simone *An Autograph Primer*, INRIA Technical Report, Sophia-Antipolis, France.
8. A.K. Bhattacharjee, S.D. Dhodapkar, S. Seshia and R.K. Shyamasundar: *STAT-EST: A Tool to Translate Statecharts to* ESTEREL . BARC Technical Report BARC/1998/E/014, 1998.
9. N. Halbwachs:*Synchronous Programming of Reactive Systems*, Lecture Notes, 21st AFCET International School of Computer Science, 1991
10. Berry et.al. : *Unpublished note on* ESTEREL *and Argos*, 1995
11. Beauvais. J.R et. al.: *A translation of Statecharts to Signal/DC+.* Tech Rep. IRISA, 1997.
12. E. Mikk et.al.*Implementing Statecharts in Promela/SPIN*, Technical Report, 1997.
13. C. Andre':*A visual Representation of Reactive Behaviours* Tech. Rep. RR 95-52 I3S, Sophia-Antipolis, France, 1995
14. R. Milner:*Communication and Concurrency*, Series in Computer Science, Prentice Hall, 1989.
15. A.K. Bhattacharjee, S.D. Dhodapkar, S. Seshia, R.K. Shyamasundar:*A Translation of Statecharts to Esterel* Accepted for publication in the proceedings of FM'99(Technical Symposium), Toulouse, France, 20-24 Sept., 1999