Real-Time Systems Formal Modeling based on Statecharts

Vinícius Valente Maciel
Pós-Graduação em Ciência da Computação
Universidade Federal Fluminense
vmaciel@infax.com.br

Maria Luiza d'Almeida Sanchez
Pós-Graduação em Ciência da Computação
Depto de Engenharia de Telecomunicações
Universidade Federal Fluminense
mluiza@ic.uff.br

Abstract

The goal of this research is the specification and implementation of an environment that employs traditional software engineering tools (in compliance with object-oriented concepts) to elaborate verifiable models (in compliance with formal techniques), allowing consistence analysis and system behaviour simulation - important features to real-time systems development. Statecharts are excellent for representing real-time systems and the statechart model can be combined with an object model. The statechart system model can be converted to SCSL (Statechart Specification Language) [16] and ASM [17] (to specify the actions presents at the statecharts). The SCSL-ASM system model is the input of an automatic tool developed to enables model analysis by a Model Checker. In this way, we demonstrated the possibility to develop software with high confiability, using a formal method mixed with an object-oriented method.

1. Introduction

Nowadays software engineers normally uses semi-formal techniques for software development, especially they constructs object-oriented models with UML [12]. These techniques emphasize the employing graphical modeling tools in order to facilitate the
model comprehension and to minimize the communication gap between users and developers.

Formal modelling techniques allows consistence, correctness analysis and system behaviour simulation, through well-defined rules. But they present difficulties [5]:

- The model verification process is expensive;
- Software engineers have to be trained since they are not adapted to the use of formal techniques;
- Formal techniques are not simple and do not provide an immediate gain;
- The model comprehension are not so attractive and we have to remember that systems models are the basis for teams members communication;
- We demand more time to elaborate formal models than object oriented models. The automatic code generation is the prize from formal model, but it isn’t always possible.

However, real-time embedded systems presents single characteristics to incentive formal methods employment, in despite of negative points mentioned above. For example, the use of formal methods for cellular telephone software development is a lucrative investment since we compare this with the cost of a recall for thousands of users. If the system being modelled is safe critical, the importance of formal methods increases [7] since detection and removing errors may occur at the initial development phases, through model verification.

The goal of this research is the specification and implementation of an environment that employs traditional software engineering tools (in compliance with object-oriented concepts) to elaborate verifiable models (in compliance with formal techniques). In this
way, we intend to structure a development formal technique, with graphical presentation, easy learning, and high communication power.

A system is a composition of concurrent objects, i.e., the state of each concurrent object can change independently. The system state is the composition of each component object state. The specification of all objects’ states and consequently the specification of all system’s states may employ one or more finite state machine [14] and this is the modeling approach of this work.

Real-time systems are reactive systems, i.e., highly dependent of these internal states. A system reaction depends from: (i) an external event occurrence and (ii) its actual system internal state. Statecharts is a visual formalism developed for the reactive systems modeling. It extends finite state machines incorporating hierarchy, concurrency and broadcast. These extensions allow the specification of systems with high level of complexity [iState]. Statecharts are excellent for representing real-time systems and the Statechart model can be combined with an object model.

The statechart system model can be converted to SCSL, a language introduced in [16] with a one-to-one mapping for graphical statecharts elements. In order to complete the formal modeling, we proposed the use of ASM (Abstract State Machine) [17] to specify the actions presents at the statecharts. ASM is a formalism with a syntax and semantics near conventional programming languages. The SCSL-ASM system model is the input of an automatic tool developed to enables model analysis by a Model Checker. Model Checking is a formal verification technique that explores, in an extensive way, all the possible states of a state machine in order to verify the correctness of a formula in a temporal logic. Model Checking is the formal verification technique used in this work to validate a Statechart System Model.
This paper is organized as follows: in the section 2 we show the most important concepts of statechart; in section 3 we describe the model verifying process and a case study model with its correctness analysis. Finally, in section 4 we conclude this paper presenting its actual stage and future perspective of this research.

2. Statecharts

Statechart is an extension of traditional states-transitions diagrams. A complete definition of the topic in question is presented in [1]. The most important concepts incorporated are state hierarchy (depth), orthogonality, (representation of parallel activities) and interdependence between states (communication mechanisms).

The basic statecharts elements are: States, Transitions, Events, and Actions. Each basic element can be described as follows:

- **States** used to describe components (and its possible situations) of a given system. They represent the variable’s values of the system at a specific instant. The statechart states are classified in two groups: basics and not basics;
  - **Basic States** are states that do not have substates;
  - **Not Basic States** are states that must be decomposed in substates. There are two types of decomposition process: OR or AND. OR decomposition means that the system is at a single substate at a given instant. AND decomposition, however, means that the system state is due by more than one substate simultaneously;

- **Transitions** represent a possible state change inside the system;

- **Events** are entities that change actual system behavior. Its occurrence can move the system to another state. If desired, a condition can be attached to an event, also called
guard condition. When the event occurs, its associated transitions only occurs if condition is true;

- **Actions** are defined by one of the following options: (i) a procedure execution; (ii) a variable attribution and (iii) events that trigger to another parallel components. Actions can be associated with transition or states (entry/exit actions).

The Figure 1 shows a real problem statechart model – a crossroad traffic light control system, used at two one-way roads. This example was first presented at the STATEMATE\(^1\) reference manual [4].

This model represents two concurrent devices – the traffic light of the north-south road (at the top of the Figure 1) and the traffic light of the east-west road (at the bottom of the Figure 1). These are traditional three stages traffic lights: red, green and yellow, with states represented at Table 1.

<table>
<thead>
<tr>
<th>Road</th>
<th>State</th>
<th>Green</th>
<th>Yellow</th>
<th>Red</th>
</tr>
</thead>
<tbody>
<tr>
<td>North-South</td>
<td>Road-NS</td>
<td>GREENNS</td>
<td>YELLOWNS</td>
<td>REDNS</td>
</tr>
<tr>
<td>East-West</td>
<td>Road – EW</td>
<td>GREENEW</td>
<td>YELLOWEW</td>
<td>REDEW</td>
</tr>
</tbody>
</table>

**Table 1 – Traffic Light States**

The system must respect the following property: it is not possible that the state of the two traffic lights to be red or green simultaneously. In our model, one traffic light moves from the state green to state yellow and from state yellow to state red by the occurrence of a clock event. However, the transition from state red to state green (in compliance to safe requirements) is triggered by an internal event – the occurrence of state transition to red by the other traffic light.

---

\(^1\) Statemate is a CASE tool with statechart graphical notation for the specification of real-time systems.
In this example the super-state SEM describes the entire problem: the crossroad control with two traffic lights. The super-state SEM is a “not basic state” – a AND states decomposition. At the SEM state we can see two concurrent statecharts (the dotted line partitioning the diagram is the graphical representation of the concurrence), modeling the independence of traffic lights operation. The cooperation between those objects is expressed by internal events IN/OUT. Those events are triggered by the model execution whenever a statechart enters/exit a state.

The events with \( tm \) prefix at the above example are clock events. They model the passing of a time interval that starts when the statechart moves to the origin transition state.

A transition changes the actual state in a statechart. Events cause transitions. Events can be classified as:

a) \textit{External events}: clock events or those triggered by external environment;
b) *Internal events*: triggered by the model execution whenever a statechart enters/exit an state or fired inside actions associated to transitions or states.

In the example of Figure 1, if the EW traffic light is at YELLOWEW state, the occurrence of external event *tmlnyellow* moves the system to the state REDEW. This triggers the internal event IN (REDEW), relevant to the NS statechart. If, at this instant, the NS traffic light is at REDNS state, another transition will be fired, in this diagram, to the GREENNS state. This last event may be certainly treated before any other external event.

A *step* is defined as a transition set and internal events triggered by an external event. Internal events are consumed before any other external event. The set of transitions in consequence of an internal event is named *micro-step*. A step processing may move the system to a stable configuration; i.e., all the triggered internal events are consumed and the system are ready to process another external event.

A statechart *configuration* is its current state, which means the set of current state of each component statechart, the value of each variable and all the triggered internal events not yet consumed.

Some works restricted some transitions in consequence of internal events firing, in order to guarantee that a statechart reach a stable state. Those restrictions can results in some conflicts - the occurrence of the same external event can move the system to different states as a consequence of the transition set construction order [11].

In this work, we adopted another approach, since we are interested at deterministic systems, we considered that the system must to respect the following properties:

a) A external event processing must move the system to an stable configuration;

b) The order of trigger and process internal events don’t change the statechart configuration at the end of a step.
2.1. Proposed Semantics

The Figure 2 shows the operational semantics mechanism for statecharts proposed in this work. This semantics is defined with a statechart execution machine, composed by:

- An statechart representation;
- A dispatcher;
- A events channel.

The events channel stores all the events to be processed by the entire system. It maintains two FIFO lists, one for external events and other to internal events (triggered internally to the machine).

The dispatcher is the engine of our machine. Inside it a structure maintains the statechart configuration. The dispatcher removes events from the channel to be processed, respecting its priority:

- Internal events have higher priority and are removed first – this means a micro-step cycle starting;
- External event is removed just when the list of internal events is empty- this means a step cycle starting.

The event channel and the dispatcher work together with a statechart representation in order to execute it.

Each processing cycle starts with the dispatcher removing an event from the channel and sends a broadcast message with this event wrapped. This message is delivered to each state of the statechart representation configuration.
A state receives the message from the dispatcher, and prepares a message reply with its behavior caused by the event occurrence. The dispatcher processes the reply message updating the statechart representation configuration.

The options of state behavior in consequence to an event are:

- Ignore the event;
- Fire a transition to another state;
- Fire a transition to the same state.

The last two alternatives can be related with actions execution defined at the statechart representation. Besides, firing a transition triggers internal events and these are new events in the channel event. The internal events are IN/OUT state or explicitly triggered inside the executed actions. After all replies are received by the dispatcher a new cycle is started.
2.2. A Statechart Execution

At Figure 3 we can see a statechart model of a client-server problem. The client, for instance, can be a browser and the server, a web server. It presents an AND state with two concurrent statecharts to represent the client and server threads. The client requests a service to the server and the latter will reply after processing the request.

In the thread C model, the client’s initial state is the WAITUSER state (waiting a request from the user). When the external event get occurs, the client moves to the PROC state (processing). At this state, the client executes an entry action, firing a request event at the end of the execution. This event changes the server state and, consequently, the triggered internal event moves the client to WAITRESP state (waiting response).

In the thread S model, the server’s initial state is the WAIT state. A request from the client moves the server to the SERVICE state. At the SERVICE state, the entry action executes the request from the client, firing a response event at the end of the execution, which finally moves the client toward the WAITUSER state. This triggers an internal event responsible for returning the server to the WAIT state.

Initially, only the event get is in the events channel when the machine starts. The dispatcher removes get from the events channel, wraps it in a message delivering it to the known states of the configuration - (WAITUSER, WAIT). The WAIT state ignores the event. The WAITUSER state notifies a state change to the PROC state and fire the request event (triggered at the end of the entry action execution). Internal events OUT (WAITUSER), IN(PROC) and request are inserted at the events channel, in this order. The dispatcher maintains the new configuration (PROC, WAIT).
2.3. **SCSL – Statechart Specification Language**

Since statechart is a graphical formalism, we propose a simpler language – SCSL – to describe statechart in a textual format. SCSL has a syntax and semantics that enables an easy learning by the programming teams.

Table 2 shows the mapping of each statechart symbol (statechart column) to an SCSL element (SCSL column).

<table>
<thead>
<tr>
<th>Statechart</th>
<th>SCSL</th>
</tr>
</thead>
<tbody>
<tr>
<td>State OR</td>
<td>State</td>
</tr>
<tr>
<td>State AND</td>
<td>Thread</td>
</tr>
<tr>
<td>Event</td>
<td>Event</td>
</tr>
<tr>
<td>Condition</td>
<td>ASM expression resulting a boolean</td>
</tr>
<tr>
<td>Action</td>
<td>ASM block</td>
</tr>
</tbody>
</table>

Table 2 – Statechart to SCSL dictionary
<table>
<thead>
<tr>
<th><strong>Graphical Representation</strong></th>
<th><strong>SCSL</strong></th>
</tr>
</thead>
</table>
| ![Statechart 1](image1) | state S1 (){
| | ... |
| | ]; |
| ![Statechart 2](image2) | state S1 (){
| | state A (){
| | ... |
| | state B (){
| | ... |
| | }; |
| ![Statechart 3](image3) | state S1 (){
| | thread A (){
| | ... |
| | thread B (){
| | ... |
| | }; |
| ![Statechart 4](image4) | state A (){
| | on off go |
| | ]; |
| ![Statechart 5](image5) | state B (){
| | }; |
| ![Statechart 6](image6) | State A {
| | on off go B if [i < |
| | ]; |
| ![Statechart 7](image7) | state B (){
| | }; |
| ![Statechart 8](image8) | state A (){
| | on off go B if [i < 10]
| | do [ i := i + 1 ]; |
| | ]; |
| ![Statechart 9](image9) | state B (){
| | }; |

Table 3 – Translating statecharts to SCSL
Table 3 shows the most important statechart constructions and its translation schema to SCSL.

A Classes diagram (Figure 4) is used to guide the SCSL mappings:

- the program creates a thread to manage events occurrence (Event Channel)
- each SCSL program creates a thread dispatcher to process the Event Channel
- each thread creates an operational system thread
- the Statechart inside each thread is implemented with a memory resident transition table and generates a dispatcher for local use

![Figure 4 –Classes Diagram of SCSL mapping](image)

3. Model execution and analysis

In the first place, we used Maude tool as the basis to model execution and validation. Maude [8][9][13] is a declarative language and a high performance interpreter well-founded in rewrite logics [10]. A Maude program is a logic theory, and a Maude
computation is a logic deduction using axioms from theory and program. Maude was chosen since it is an algebraic specification language, with a broad application spectrum, enabling an object-oriented implementation of our semantics and a high performance model checker on the fly[15] available, comparable to the most used model checkers.

At first, we translated SCSL into Maude making this approach efficient enough to verify a restrict class of statecharts [16]. However, this approach made it difficult to elaborate formal definitions for actions associated to the transitions/states. The use of Maude to specify actions was not a good alternative, since algebraic specifications are not often used for traditional programming pattern, and consequently making it hard to adopt.

In order to provide a complete modeling tool, it is imperative to incorporate formal specifications for actions. To supply this requirement, we selected ASM (Abstract State Machine) - a formalism, with syntax and semantics near traditional programming languages that enables simultaneously computational description and model execution. In [3] Tirelo summarized the ASM concepts as follows:

ASM are abstract machines, where a state is a set of functions. This functions have names and are defined into a set named “state super-universe”. The set of functions names is a state vocabulary. The meaning of a function name is a mapping of vocabulary names into its functions. A machine state transition is due by a transition rule. A transition rule modifies the interpretation of some functions names in the state vocabulary. A transition rule ASM has a program format like using a normal imperative language. The main difference is the absence of iteration, since this concept is implicit inside execution machine. The execution consists in run the transition rule so many times, as necessary, changing the actual state at each time. In this way, we construct the sequence of states of
the same vocabulary, composed of different functions. We say that the interpretation of names in the vocabulary changes while state changes during the execution.

The model elaborated in this proposed environment employs SCSL (mapping the statechart model) and ASM (for actions specification), resulting a complete formal model. The semantics of statechart proposed is now written using ASM. We specify an ASM to Maude mapping in order to translate the entire system model to Maude. This mapping uses the Maude reflexive features and allows references to Maude core library. This library has, for example, the NAT type and its operations, like summation. The mapping ASM to Maude is applied, originating a Maude model. Hence, this enables one to execute and verify this semantics employing the same tools developed at the first phase of this research.

3.1. Processing the model

Figure 5 illustrates the process that translates the model into a verifiable one. This process is segmented in the following steps:

- **Step 1**: the statechart system model is manually translated into an SCSL model.

- **Step 2**: the SCSL model, increased by actions specifications using ASM, is translated to an ASM specification, using the scsl2asm tool and a SCSL compiler developed in this research context.

- **Step 3**: the ASM specification is translated to Maude, using asm2maude tool, developed in this research context employing the mapping ASM–Maude proposed. This code is used to, afterwards, verify the model.

- **Step 4**: the Model Checker Maude is used with the following inputs, both generated at step 3: the Maude model and a set of LTL (Linear Temporal Logic) predicates –
properties that we want to verify at the model. The use of LTL is due to the fact that Maude Model Checker employs LTL to expresses the verified predicates.

To verify the model, the model checker needs to do a complete search in the entire space of states, therefore, the problem must have a finite space of states as a requirement. Hence, this restriction inhibits generalizations. For example, if many instances of an active object is being considered, the verification of a model case with 3 instances and another with 4 instances does not validate the model with n instances.

The model checking problem is as follows: to a system with transitions α and a temporal logic formula π, it must verify if π is satisfied with α. If π is not satisfied, the model checker returns a trace in order to explicit a case where it detected a false value to the formula.

![Figure 5- Steps during model verification](image)

### 3.2. Case Studied

The statechart model for the crossroad traffic light control system, presented at Figure 1, was translated from graphical notation to SCSL (see Figure 6).

The #include<scsl.sm> imports the proposed semantics for statecharts execution. A submachine is declared with a SCSL block. This block directly translates the statechart.
Figure 6 – SCSL source for Crossroad Traffic Light Control System

This SCSL model was an input to SCSLC tool (this tool incorporates steps 2 and 3 described at the verifying process, and execute in sequence scsl2asm and asm2maude), originating a Maude equivalent model. Besides model Maude, it generates another file containing the necessary code to apply model checking to our mode, i.e., this file contains the general properties definitions and the initial state.

In the case studied, the following properties were observed:
a) If the system initiates at a stable configuration, an external event processed may modify it, leaving the system at another stable configuration.

b) The two traffic lights can’t stay at a red state at the same time.

c) The two traffic lights can’t stay at a green state at the same time.

The Figure 7 shows the verification result of property (b). This property was written with LTL as follows:

\[ \langle \text{REDNS} \rangle \neg (\text{REDEW} \land \text{PREDConfStable}) \]

This means: henceforth, the system never reaches, in a stable configuration, at the same time, the both states REDEW and REDNS.

--- Welcome to Maude ---
Maude 2.0.1 built: Aug 1 2003 17:25:59
Copyright 1997-2003 SRI International
Tue Apr 13 11:45:46 2004
Maude> red modelCheck(initial, \[ \neg (\text{PREDInState(REDNS)} \lor \text{PREDInState(REDEW)} \lor \text{PREDConfStable}) \] ) .
reduce in CHECK-CS : modelCheck(initial, \[ (\text{PREDConfStable} \lor (\text{REDNS} \lor \text{PREDInState(REDEW)})) \] ) .
rewrites: 750281 in 55530ms cpu (55730ms real) (13511 rewrites/second)
result Bool: true

Figure 7 – Model Checker Result

4. Conclusion

The formal model system approach presented in this paper resulted in an efficient tool, in the sense that it becomes easy to model and to understand it. Besides, the resulting model is precise and can be validated with little effort.

The SCSL language associated with ASM formalism is in compliance with traditional modeling/programming paradigms. The inclusion of ASM gives extra benefits:
the language that describes statecharts (SCSL+ASM) is a complete computational language, which inherits the ASM simplicity, types and operations from Maude library.

Besides, the use of ASM permits model elaboration, for the same system, with different specification levels, from elaboration of partial models with simple statecharts and actions defined as pre and post conditions lists, up to complete system models with complex procedures specification.

In this way, we demonstrated the possibility to develop software with high confiability, using formal method mixed with object-oriented method.

The correction proof of ASM to Maude mapping and the employment of these tools in a real and complex system model are being done in order to obtain a full notion of the proposed tools applicability. Afterwards, we intend to investigate new approaches for model verification, like the use of theorem proovers strategy based, to supply classes of systems where model checking is not applicable, especially in systems where generalizations are of fundamental importance.

Acknowledgments: The authors would like to acknowledge the support of Brazilian Agency CNPq.
5. References


