Synchronous Estelle: A Language to Specify Distributed Control Systems

Miguel Riesco, Javier Tuya, Oliverio González
Computing Department, University of Oviedo.
E-mail: albizu@lsi.uniovi.es

Abstract

Distributed systems are getting more and more popular in every field of computing. This is also true for reactive systems. Traditional techniques for developing reactive systems are not very suitable for developing distributed reactive systems, because they are based on synchrony hypothesis, and it can not be hold in a distributed system. It is necessary to develop new techniques to deal with distributed reactive systems.

In this paper we propose to join the features of two techniques used to specify systems: Estelle and StateCharts. The first one is very suitable for distributed systems, meanwhile the second one is broadly used for specifying reactive systems.

Introduction

A reactive system is defined as a system that has a permanent interaction with the environment, when this has not synchronicity capabilities [1] [3], i.e. the environment does not stop to wait for the system reaction. The evolution of the system is conditioned by the stimuli arrived from the environment and its internal state, reacting to them generating new stimuli and/or changing its internal state.

Traditional techniques used to develop concurrent systems are not satisfactory for developing reactive systems [1]. Instead of this, several specific languages have been developed. These languages are usually called "reactive synchronous languages": reactive because they are used to specify reactive systems; synchronous because their semantics are based in the "synchrony hypothesis" of Gérard Berry [2]. We can mention languages as StateCharts [4], RSML [6], Lustre [3], Signal [5] and Esterel [2], all based on this hypothesis but with different programming styles.

The assumption of the synchrony hypothesis (system reacts in a zero time) makes easier the development of the system, because the developer does not have to worry about the time the system takes to react to the events from the environment, so he does not have to synchronise anything. Moreover, the system can be decomposed into several simpler concurrent
subsystems without affecting its observable behaviour \((0 + 0 = 0)\). This can seem artificial (every computation takes some time), but the actual meaning of this hypothesis is that the system reacts to an external event as fast as possible, and before the environment can emit the following event.

**Why Synchronous Estelle?**

Nowadays reactive systems get more and more complex, and the usual way to make their implementation is to decompose them into simpler systems, more or less autonomous, connected one another by means of a communication network, building a distributed reactive system.

To specify this kind of reactive systems one problem arises: usual reactive languages are based in the synchrony hypothesis. This hypothesis can be considered realistic when the reaction of the system depends only of the execution of one computer, but when we distributed the control over a set of computers connected through (probably low speed) communication channels, this hypothesis can not be hold [6].

Estelle is a Formal Description Technique, standardised by ISO, very suitable to specify distributed systems. So, we can try to use it to specify distributed reactive systems. However, some features of reactive synchronous languages (which Estelle does not have) make Estelle not suitable to be used for this propose. We can mention, among others, the following reasons:

- Operational semantics of synchronous reactive languages is (from an external point of view) simpler than Estelle semantics. Synchrony hypothesis make simpler system specification to the developer, because he does not have to worry about the duration of the reactions. Estelle transitions take no time to react too, but the total reaction time of the system is defined.

- The specification of the system is simpler in synchronous reactive languages based on finite state machines (like StateCharts). They model the system like one machine, instead of the set of machines that use Estelle to do the same.

- Events not treated in a step are ignored in the following step. In Estelle all messages must be treated. This feature is annoying when the environment (or the system itself) is generating events in a continuous way (like a sensor does, for instance).

Summarising, we have two languages with interesting features, that could be integrate in only a new language: we will adopt the way StateCharts specify independent systems, while we will use Estelle to specify the communication among different subsystems. This new language
will be called *Synchronous Estelle*.

**The language of the StateCharts**

Among all the languages designed to develop reactive systems we have chosen StateCharts because is a visual language, that allow us to develop systems in a very natural way.

StateCharts is a visual formalism for specifying reactive systems. It uses state-diagrams (traditional state machines) with some additional features, like concurrent states and modular state decomposition, to avoid the explosion of the number of states.

The system evolution is driven by the events that come from the environment (or from the system itself). All the events are instantaneously broadcast to the whole system.

A statechart represent a system like a state machine, using rounded rectangles for the states and arrows for the transitions among them. Transitions are labelled with a expression of the form:

$$t_1: i (c) / o$$

where \(t_1\) is a tag for this transition, \(i\) is an expression of the events that will shoot the transition, \(c\) is a condition that guard the transition, and, finally, \(o\) is the set of events that will be generated when the transition is shot. Any of these parts of the transition expression can be omitted.

A state can be divided in several substates, building a hierarchy of states. Some of the states can be concurrent (orthogonal).

In this example we have one statechart component, named St1. We have omitted transition tags to simplify it. The component is decomposed into three states: Cmd, Norm and Disp. The first one is a single state; the second one is decomposed into two states; the third one is a state
compound by two parallel states.
There are some additional features, like history connectors, condition and selection entrances, delays and time-outs, ... A more detailed description of the Statechart formalism can be found in [4].

**The Synchronous Estelle Language**

We propose to add to a Estelle specification a new kind of module, which we call *systemsynchronity*. Modules of this class will run in parallel with the rest of modules of the Estelle specification, but inside of them syntax and semantics are different of Estelle modules and more similar to a component of a Statechart or RSML component.

One *systemsynchronity* module stands for one finite state machine with hierarchical and parallel states. One module of this kind can be divided again in several modules, each one of them represent one "extended state" of the hierarchical machine. The structure of this state machine and its operational semantics is similar to an Statechart.

Apart of this kind of modules, the rest of the Synchronous Estelle specification is similar to Standard Estelle.

To specify the *systemsynchronity* modules we will use a very similar syntax to standard modules, in order to make it familiar to Estelle developers. Moreover, this syntax has a direct relation with a graphical specification, making easy to develop tools that can obtain a textual Synchronous Estelle specification from a graphical one.

**Synchronous Estelle syntax**

A systemsynchronity module is declared in two parts (like every Estelle module): first the header is declared, and then the body of it.

In the header declaration appear the name, the interaction points with the rest of the specification and the interaction points with the environment. Interaction points with other modules are kept to use the usual Estelle message passing mechanism. The environment interaction points are an easy way to connect the system with the rest of the world.

```plaintext
module name systemsynchronity;
ip
  Interaction points dec.;
  Environment Int. Points dec.;
end;

body body_name for name;
  Local usual declarations;
  Local Events declaration;
  Module declarations;
  Estelle transition declaration;
  Transition between modules declaration;
initialize to initial module
begin
  Initialization sentences;
end;

Module header declaration
First level modules declaration
```
The body declaration is made apart to allow different behaviours to the same modules (as usual in Estelle). Here we can declare constants, types, variables, procedures and functions in the same way than in standard Estelle. We can also declare new modules inside, that will represent compound states in the global state machine. Transitions between modules stand for transitions between states, but they have a similar syntax than in Estelle. Finally, we can specify initialization sentences, that will be executed at the beginning of the execution of the system.

Declaration of internal modules is made in a very similar way. In this case we declare directly the body of the module (without any header declaration), because we don’t have any interaction points to declare. In the case of internal modules they can be tagged as parallel to represent a state that can be active in parallel with others states. Finally, we can give an alias to the module to facilitate the naming of the module from another module of the system.

<table>
<thead>
<tr>
<th>module body_name [parallel] [alias name];</th>
<th>Trans from module1 to module2 [, module2, ...] when event expression provided condition begin sentences; end;</th>
</tr>
</thead>
<tbody>
<tr>
<td>Local usual declarations;</td>
<td>General module declaration</td>
</tr>
<tr>
<td>Local Events declaration;</td>
<td>Module declaration;</td>
</tr>
<tr>
<td>Transition between modules declaration;</td>
<td>Transition declaration</td>
</tr>
<tr>
<td>initialize to initial_module;</td>
<td></td>
</tr>
<tr>
<td>begin</td>
<td></td>
</tr>
<tr>
<td>Initialization sentences;</td>
<td></td>
</tr>
<tr>
<td>end;</td>
<td></td>
</tr>
</tbody>
</table>

Transition declarations have some differences with Estelle transitions:

- After the “when” clause we have to specify an event expression (similar to a boolean expression, changing boolean variables with event occurrence).
- One transition can have different target modules (states) at the same time, if they are declared as parallel, of course.

Inside a systemsynchrony module (in a transition declaration) we can include the command “output”, similar to the Estelle one. This instruction generates an event, that is immediately broadcast to all the modules inside the ambit of that event. The rules of an event are the same that any variable in Estelle. This is different to StateCharts, where events are global to the whole system.

**Synchronous Estelle Semantics**

A Synchronous Estelle specification execution makes Estelle normal modules run in parallel (with their own semantics) with systemsynchrony modules, that have an operational semantics very similar to the Statecharts one.
The system execution (for this kind of modules) is divided in a set of computational steps, which are divided in micro-steps.

A step begins with the arrival of a set of messages from other modules and events from the environment. These messages are treated as declared in ‘Estelle transition declaration’ of the body of a first level module. The aim of these first phase is to transform Estelle messages into events, that is the communication mechanism used inside a systemsynchrony module. So, the result of this first part of the step will be a set of generated events.

We have to note that the treated messages are those that have arrived before the beginning of the step; all messages that arrived after the beginning of the step will be treated on the following step.

The set of generated events, in conjunction with environmental events, is the input for the first micro-step. These events will enable many transitions; from which the system will choose one to be shot. To do that the system will sort transitions by its level on the system: “higher” transitions (closer to the top of the system) have more priority than “lower” transitions. Transitions of the same level have the same priority, so the system will choose them in a non-deterministically way.

The shooting of the transition makes the system to change its state, and, probably, to generate more events. This events are added to a second set of events to be treated. Another transition is chosen and this process is repeated until the first set of events is empty (or there is not any transition enabled).

Now the events to be considered are the second set of events. This is the input of the following micro-step, and it begins. This process continues until a micro-step can not begin because there is not any enabled transition.

Now external signals are sent to the environment and messages are sent to other modules, finishing the step.

The interaction to the rest of the system and with the environment is done only at the beginning and at the end of each computational step. At the beginning of the step all messages are treated, as seen before, and at the end of the step all messages generated within it are sent to the rest of the system (and to the environment).

**Developing Synchronous Estelle Specifications**

Any language or technique used to develop systems has to be released with a set of tools that help developers to do their job.

We are working in the development of the following environment:
All Synchronous Estelle specifications will be transformed into a SIF (Synchronous-Estelle Intermediate Form), an intermediate notation used to simplify the manipulation of the specification.

With this toolset we will be able to specify the system in a graphical way (using the Synchronous Estelle Graphical Editor) or in a textual way (using the Synchronous Estelle Textual Editor). In both cases, the specification will be stored in SIF notation. Of course, we can work in a graphical way with a specification wrote in a textual way before.

In order to execute a specification developed in Synchronous Estelle it is necessary to translate it into something understandable by the computer.

We propose two choices: the first one is to translate it to Standard Estelle (by means of Synchronous Estelle Translator), so a developer can use the existing tools to work with this FDT. With this option the semantics of the synchronous modules will be interpreted by a Estelle specification.

Another possibility is to compile all the systemsynchronity modules into a global automata (like it is done in Esterel), coded in some host language (using the Synchronous Estelle Compiler). This option will generate one Standard Estelle specification and one executable module that is called from inside the Estelle Specification. This executable code will contain all the code necessary to execute the systemsynchronity modules of the original specification. This option will make a faster (but bigger) executable code.
References


