Electronic Notes in Theoretical Computer Science 88 (2003)
URL: http://www.elsevier.nl/locate/entcs/volume88.html 15 pages

Synchronous Estelle: Just Another Synchronous Language?

Miguel Riesco [1] Javier Tuya [2]

Computer Science Department
University of Oviedo
Oviedo, Spain

Abstract

Synchronous Estelle is a new language designed to specify distributed reactive systems. We have developed this new language, that tries to merge both asynchronous and synchronous paradigms, as an answer to some problems not solved with existing techniques. We will show its syntactic and semantic features, the way to obtain executable programs according to a specification, and some hints to perform the validation of the programs written in this language.

The Steam Boiler Control Problem, a referent study case in the field of reactive systems, has been used to study the applicability of Synchronous Estelle to distributed reactive systems.

Key words: Synchronous Estelle, Reactive Systems, Distributed Systems, Estelle, Reactive Languages, Steam Boiler

1 Introduction

The term *reactive system* was introduced to characterize a kind of systems different from the *transformational* ones. Many languages and techniques have been used to try specify them. The first techniques used to do this (concurrent programs synchronized and communicated by OS primitives, finite- states machines, Petri nets, classical concurrent programming languages, etc.) present several problems that made them not suitable for this kind of systems [4]. Consequently several specific designed languages were developed to specify, program and verify reactive systems. There are several different approaches: graphical languages based on automata (Statecharts [14], RSML [22], SyncCharts [2], Argos [25]), data- flow based languages (Signal [21], Lustre [13]), imperative languages (Esterel [5], Reactive C [8]), ..., but all of them are based

[1] Email:albizu@lsi.uniovi.es
[2] Email:tuya@lsi.uniovi.es

on the *synchrony hypothesis*. The adoption of this hypothesis (which says that *the reaction time of the system is zero*), helps the developer to specify reactive systems.

Synchronous Estelle [28] is a new member of this family of languages, usually called *synchronous languages*. It was created to specify distributed reactive systems, because we found that existing languages were not completely suitable for this kind of systems, where the features of reactivity and distribution appear together.

Most existing languages and tools to specify reactive systems are based on the synchrony hypothesis, but this hypothesis is not applicable in a distributed environment. As we can read in [4] *"... real-time systems are often implemented on distributed architecture, that is on sets of processors connected by asynchronous means. Synchronous models as introduced before can hardly be considered as realistic for such target architectures"*.

As the same need was felt by some other working teams some tools were designed to specify distributed reactive systems (Distributed Reactive Machines [9], Meta [26], CRP [6], Corea [7]), but they are only useful for a few aspects of distribution:

- Distributed Reactive Machines tried to extend the synchrony hypothesis through the network, using a centralized synchronizer. Due to network delays, system reaction can not be considered *instantaneous*.
- Meta detaches the interface from the reactive kernel, allowing each part to be implemented in different machines.
- CRP allows the communication of reactive nodes (implemented in Esterel) using a *rendezvous* mechanism. We consider this technique as a good approach, but we think that message passing is a more intuitive communication mechanism than *rendezvous*, and it does not introduce unnecessary delays due to the synchronization.
- Corea makes different reactive systems share the main clock, and assumes that the communication among reactive nodes has an exact and constant length of time.

So, we found that only CRP can be considered a valid option to specify distributed reactive systems, but it has the problem mentioned above.

We have studied two additional languages, but they present also some drawbacks:

- The use of Signal in the design of distributed reactive systems was rejected because data-flow declarative languages are not easy to understand in some environments. A graphical language can be better accepted.
- The GALS (Globally Asynchronous Locally Synchronous). We will deal with this model later.

Next we tried the use of Formal Description Techniques (FDTs) (Lotos [20], SDL [12], Estelle [19]), but we found that they are not adequate to specify

reactive systems either, although they are very suitable to describe distributed systems. We found that their semantics depart a lot from the semantics of synchronous languages, which we consider easier to understand.

A complete comparison of all these techniques and the problems we found when dealing with distributed and reactive systems can be found in [27]. The conclusion of this study is that no existing language has all the features needed to deal with a distributed reactive system.

This paper is structured as follows. First we will introduce Synchronous Estelle main features, including its syntax and semantics. The way of obtaining the executable specification from the original one and how the verification of the system can be done is also presented. Next we compare our language with GALS system and OcRep tool. Then, Section 4 will introduce the Steam Boiler Control Problem, which will be specified using our new language. Finally, we will show the conclusions of this work.

2 Synchronous Estelle

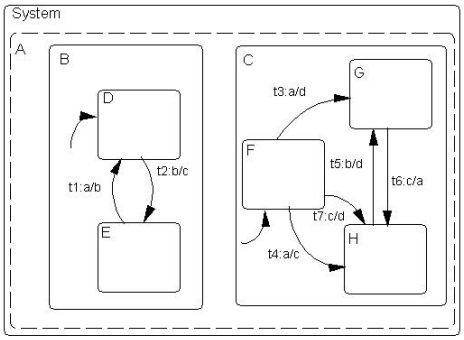
To solve the problem of specifying distributed reactive systems, we have designed Synchronous Estelle. The main goal of Synchronous Estelle is to join in one specification technique two different paradigms: first, a synchronous one, to specify reactive systems, and second an asynchronous one, to specify communication protocols.

We use the term *distributed reactive system* to define a set of reactive nodes connected one another by means of a communication network. Each node have a specific function to perform autonomously, but they have to share information to do it. So distribution has to be given at specification level.

Synchronous Estelle is an extension of the ISO Standard Estelle. Estelle is a Formal Description Technique standardized by ISO, designed to specify distributed systems. An Estelle specification consists of a set of modules and a set of channels to communicate one another by means of exchanging messages. One module can be divided into several modules in order to develop the system in a structured way. The behaviour of each module is represented by a finite state machine, where the transitions between states are triggered by the arriving of messages through interaction points.

The only difference between Synchronous and Standard Estelle is that our language adds a new kind of modules (called *systemsynchrony*), to specify reactive behaviours. So, a Synchronous Estelle specification consists of a set of modules of three types: standard systemactivity and systemprocess and our new systemsynchrony modules. The structure and semantics of systemsynchrony modules are based on the Statechart formalism: one systemsynchrony module represents one finite state machine with hierarchical and parallel states. Each module of this kind can be divided again in two or more modules, each one representing one "extended state" of the hierarchical machine.

Table 1
Graphical representation of an example system and its textual code




```
module System SystemSynchrony;
body body1 for System;
 module A; Parallel;
   module B;
     module D; module E;
     trans name t1 from E to D;
     trans name t2 from D to E;
     initialize to D;
   end module (* B *);
   module C;
     module F; module G; module H;
     trans name t3 from F to G;
     trans name t4 from F to H;
     trans name t5 from H to G;
     trans name t6 from G to H;
     trans name t7 from F to H;
     initialize to F;
   end module (* C *);
 end module (* A *)
end; (* Body1 *)
```

It is important to note the difference between Standard Estelle modules and systemsynchrony modules: both can be divided into a set of modules, but they represent a set of state machines in the former, while each module represents one state of a hierarchical state machine in the latter. The semantics of the new kind of modules is completely different from the semantics of standard modules, as will be shown.

Synchronous Estelle supports both textual and graphical specification. The textual specification of this kind of modules has a very similar syntax to Standard Estelle, to make it familiar to Estelle developers. This syntax has a direct relationship with the graphical specification. This allows us to use a graphical representation in early stages of development, and a textual language to complete implementation details.

Using Synchronous Estelle systemsynchrony modules we can specify reactive subsystems, while the communication between them is specified using Standard Estelle modules and message passing mechanism.

2.1 Synchronous Estelle syntax

A systemsynchrony module is declared in two parts (like every Estelle module): the header and the body. This division stresses the separation between the interface of the module and its internal structure and behaviour. It allows the definition of different bodies for the same module, making it easy to test different behaviours, in order to choose the most suitable for each situation.

The header declaration should include the name of the module, the interaction points with the rest of the specification and the interaction points with the environment. Interaction points with other modules are maintained to use the usual message passing mechanism to communicate the module with the rest of the system (other systemsynchrony or Standard Estelle modules). Interaction points with the environment are an easy way to connect the system

and the rest of the world. Three types can be distinguished, depending on the kind of events that will circulate through them: continuous events, discrete events and discrete events with parameters.

The body declaration is made apart, as said before. In it we can declare constants, types, variables, procedures and functions in the same way it is done in Standard Estelle. We can also declare new modules, which will represent compound states in the global state machine [3]. Transitions between modules represent transitions between states. Inter-level transitions are allowed. Each transition have an *enabling condition* and a *guard*. The first one is a logical expression of events (where de presence of an event is treated like *true* and its absence like *false*), while the guard is a Pascal-like logical expression. When both are computed as true the transition is said *enabled*. Finally, a module can be tagged as parallel.

Table 1 shows a Statechart style graphic representation of a system. The module drawn in dotted line is a parallel module: its sons will work in parallel. This graphical specification can be drawn with the Synchronous Estelle Graphical Editor (see Section 2.5), and it is automatically transformed into a textual specification. In the same table the Synchronous Estelle textual specification for that system is shown, where all the details have been removed in order to show only the way to specify the system structure.

2.2 Synchronous Estelle semantics

A Synchronous Estelle specification execution makes normal Estelle modules (with their own semantics) run in parallel with systemsynchrony modules, which have an execution semantics very similar to the Statecharts one. There are other semantic approaches (like SyncCharts), but we choose the Statecharts approach because it deals with event paradoxes in a flexible and easy to understand way, although it can be considered "less synchronous" than other semantics.

The system execution (for this kind of modules) is divided into a set of computations steps, which are also divided into micro-steps. A step begins with the arrival of a set of messages from other modules and events from the environment; then all the messages are dealt with generating, eventually, a set of events. These, together with environmental events, are the input for the first micro-step. These events will enable some transitions. Incompatible transitions will be eliminated from this set. Every transition of the resulting set will be shot (the enable transition with higher priority will be the first one), making the system to change its state, and, eventually, to generate more events. When all transitions are been shot, the next micro-step begins. This process goes on until one micro-step can not begin because there is not any enabled transition. Now external signals are sent to the environment and

[3] Remember that a systemsynchrony module represents a hierarchical state machine, and each submodule represents one state of that machine.

messages are sent to other modules, finishing the step. Algorithm 1 shows the implementation of the computation step.

Algorithm 1 *Implementation of computation step*

1.- I={Input events}
2.- Compute triggered=f(I)
3.- I=∅
4.- While (triggered ≠ ∅)
  4.1.- microstep=compatibles(triggered);
  4.2.- while (microstep ≠ ∅)
    4.2.1.- ToShot = highest priority transition ∈ microstep;
    4.2.2.- microstep = microstep - ToShot;
    4.2.3.- shoot ToShot
    4.2.4.- I=I ∪{ Generated events }
  4.3.- Compute triggered=f(I)
5.- Send generated Estelle messages and environment events.

Each synchronous module runs asynchronously in parallel with the rest of the modules of the system. The interaction between synchronous modules and the rest of the system is performed only at the beginning and at the end of the computational step. At the beginning of the step all messages arrived to the synchronous module are processed, usually to generate internal events. At the end of the step all external signals generated inside it are sent through external interaction points.

A system execution consists not only on changing the actual state of the automata. Each transition has an associated code, which determines the actions to perform when the transition is shot. In addition, each state has four codes associated: initialization, which is executed when the system starts; on-entry, which is executed when the state goes into the actual state of the automata; on- first, which is executed the first time the state goes into the actual state of the automata; on-exit, which is executed when the state leaves the actual configuration of the system. All these codes are written using Pascal programming language. We use this language because it is used in Standard Estelle specifications.

System response time is supposed to be zero (due to synchrony hypothesis), but in fact it will take some time. All events received during one step will be dealt with in the next step. All events that have not been dealt with during one step are ignored. This behaviour is similar to Statecharts one, but it differs from SyncCharts semantics.

2.3 Determinism

There is an important property we have tried to preserve in our semantic approach: determinism. Our semantics is completely deterministic, because even when several transitions are enabled in the same computational step,

there is a clear criterion to choose one: the transition with highest priority will be shot. All transitions have different priorities (parent module transitions rank higher than child module transitions; transitions that share the same origin are more important depending on their declaration order). Only when the user changes "by hand" the transition priority it is possible to have non-determinism (assigning the same priority to two different transitions), but in this case it is understood that the user knows what he is doing. The compiler will warn the user though it is not an error.

2.4 Compositionality

The computation step semantics is not compositional. This is so on purpose. We have chosen this behaviour for three reasons:

- It is easier to understand this kind of semantics rather than a compositional one.
- It is easier to implement, and the execution is faster.
- Modularity can be achieved implementing independent reactive modules.

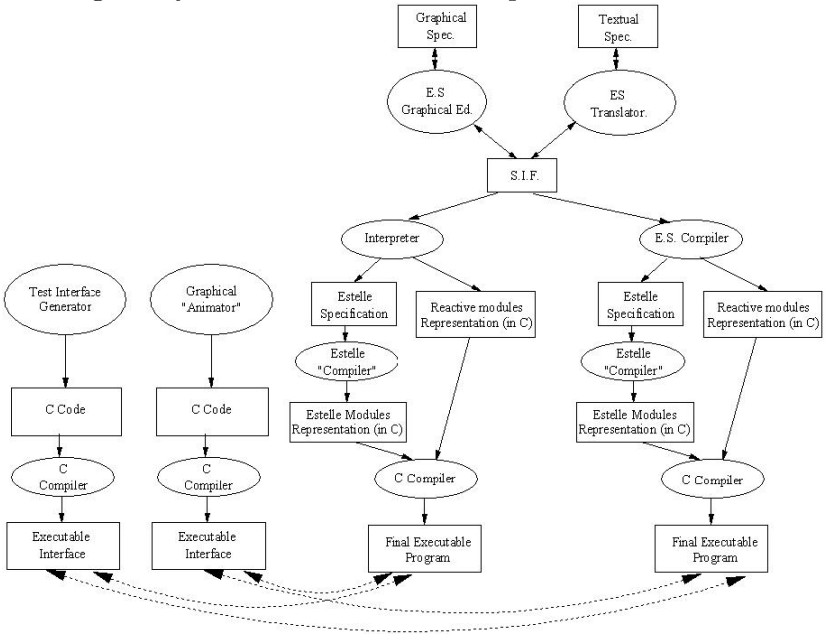
Although we know that compositionality is one of the desirable properties for a language designed to specify reactive systems, we have decided to sacrifice it to obtain responsiveness and causality, given that it is not possible to have a semantics with the three properties [17].

2.5 The Synchronous Estelle development environment

We have designed several tools to help developers use this language. The structure of this development environment can be seen in Fig. 1. We have designed the following tools:

- *Graphical Editor*: It allows us to "draw" the system in a graphical way. It generates both textual Synchronous Estelle and SIF representation of the system. In order to make the implementation of these tools easy, we use an intermediate form (*Synchronous Estelle Intermediate Form* - SIF), that contains all the information of the original specification but in a simpler format (in fact, XML is used to do this).
- *Translator*: It generates the SIF representation of the system from the original Synchronous Estelle specification.
- *Interpreter*: From a specification written in Synchronous Estelle, it generates several files: one containing the Standard Estelle modules, and a group of files, written in C, that represent the behaviour of each reactive module. Once all these files are compiled (the first one using Estelle tools, while the rest are compiled using a standard C compiler), we will obtain a set of executable programs that work together to represent the behaviour of the whole system.
- *Compiler*: It is functionally very similar to the interpreter. The difference

Fig. 1. Synchronous Estelle Development Environment



will be dealt with later.

- *Graphical "Animator"*: It allows us to examine the execution of a system, in a graphical way: the user can see the graphical representation of each reactive module (where active states have a different colour), he can perform a step by step execution, seeing graphically how a transition is shot, how a state is exited, etc.
- *Test interface generator*: It is used to generate one custom made interface with the system, to allow users to generate control events or to display the information the system sends them. It can be used to test the system or to build the final user interface.

All these tools (except the translator) have been developed and are functional, in a prototype version.

2.6 Executing Synchronous Estelle specifications

When we want to generate an executable program from a Synchronous Estelle specification we have two possibilities: either using the interpreter or the compiler. Both have a similar function: they separate each reactive subsystem and the Estelle modules, generating several Estelle and C files that can be compiled to get a set of executable programs that, running together, represent the behaviour of the system.

The first step, both in the compiler and in the interpreter, is to copy Standard Estelle modules as they are, with the addition of a new module to manage the communication with the rest of the system. The Standard Estelle Compiler, like EC [18] is used to obtain an executable program.

The interpreter represents all the information of the original specification for each reactive module within a C program. All the structural information (states, substates, transitions, etc) is included with data structures, but maintaining the original hierarchical information. An execution engine that implements the semantics of the system, as has been seen in section 2.2, is added to this program.

The compiler transforms the original hierarchical structure into a "traditional" automaton, without hierarchical nor parallel states, but with an equivalent functionally. A similar execution engine, like in the interpreter, is added too, to implement the computation step semantics.

The advantage of the second tool is a faster execution, because all operations are simpler than in the first case. In a hierarchical automaton it is not simple to choose one transition to be shot or to know which states are exited and which ones are entered. In a flat state machine these are very simple decisions. The problem is that only simple systems can be translated, because the equivalent automaton is much larger than the original one (parallel and compound states were originally designed to avoid state-explosions problems). Our studies [27] show that with 100 states, with a medium degree of parallelism [4] we can not perform a transformation due to virtual memory overflow (with the same number of states but with a low degree of parallelism the size of the program generated by the compiler is greater than 103 MB, while the size of the program generated by the interpreter is only 38 KB).

The transformation is not a simple process, because every state and every transition has to be treated, eventually, several times. Every code of each state or transition has to be modified, too. In [27] a complete description of how to perform the transformation is done.

2.7 Verifying specifications

It is important for a language like Synchronous Estelle to have some kind of tool to perform properties verification of the specification. Now we are working in verifying Synchronous Estelle specifications, using SPIN. Instead of building a new model checker from scratch, we are trying to use SPIN [16], an automated verification tool (model checker) that uses PROMELA (PROcess MEta LAnguage) as system specification language, in a similar way to what we have done before using SPIN to verify SA/RT Models [29]. Our intention is to translate the Synchronous Estelle Specification to PROMELA in order to use SPIN to verify the original system. A translator from Synchronous Estelle to PROMELA is under development for that purpose.

The question of why not using directly PROMELA to specify the system could be made. The answer is that it is easer to understand and use Synchronous Estelle (a language based on a graphical formalism) than PROMELA,

[4] The size of the equivalent automaton depends on the number of states, the structure, the number of parallel states and their position on the original system

a pure textual language.

3 Other related work

Now we can compare Synchronous Estelle with two existing techniques to specify distributed reactive systems: the GALS systems and OcRep tool. The GALS (Globally Asynchronous Locally Synchronous) systems [11] were designed to specify a reactive distributed system like a set of Finite State Machines, working in a synchronous mode. FSMs can exchange information using a single-slot buffer of signals. The global behaviour of the system is asynchronous, because communication is not instantaneous, so each FSM sees that the reaction time of the other FSMs is not zero. The POLIS [3] design environment uses a GALS model (CFSM - Codesign Finite State Machines), where each node is developed using Esterel synchronous language.

Both POLIS and Synchronous Estelle have a very similar structure: there is a set of synchronous nodes and each node communicates through non-instantaneous channels. But there are some differences between them:

- POLIS uses a "more synchronous" language, like Esterel, while Synchronous Estelle uses a structure very similar to Statecharts. The former has a stronger formal basis than the latter.
- POLIS is a complete development environment. Synchronous Estelle development environment, under construction, and all the tools are prototype versions.
- Graphical tools to specify the whole system can be used in Synchronous Estelle. In POLIS only text is used for the specification.
- Using Synchronous Estelle we can *encompass within a single framework all reactive aspects* [4]: the whole specification is included in one file. In POLIS we need the Esterel specification for each module and one additional file to indicate the interconnections of the modules.
- All modules in POLIS are synchronous; a Synchronous Estelle specification can be made up of synchronous modules and Standard Estelle modules. This kind of modules has a different structure and semantics, and have been mainly used to specify network protocols. This different approach can be interesting in some circumstances, and we can reuse existing protocol specifications when the network behaviour is important for the global system performance.

So, both approaches are very similar, but while in POLIS a lot of work has already done in Synchronous Estelle, that has some additional features, a lot of work has to be done.

OcRep [10] deals with distribution from a different point of view. With OcRep where the whole system is compiled in only one module. This module contains the code that every node of the distributed system has to execute.

The idea is to distribute this program to every location, according to distribution directives given by the user, and then remove on each node the code that is not relevant to it. Communication and synchronization will be automatically done, too. This approach offers a centralized compilation and debugging before the distribution. The advantage of this approach is that supposedly the distributed program has the same safety as the centralized one.

4 The steam boiler control problem

The steam boiler control specification problem [1] belongs to a group of benchmark problems designed to compare different techniques to specify reactive and real-time systems. Other similar problems are the Production Cell Problem [23] or the Generalized Railroad Crossing (GRC) [15].

The steam boiler is a typical example of a reactive system. There is a set of sensors (water level, steam rate, state of the pumps) and a set of actuators (four pumps, one valve). According to the values measured by the sensors, the system has to react opening or closing the pumps or the valve to keep the water level in a safe value. The controller has to respond as soon as possible to avoid the boiler damage, even when one component of the boiler fail.

From 1995, when the problem was introduced several techniques have been applied to solve the problem [1]. Timed Automata, Statecharts, Esterel, Lustre, NUT, Z, Evolving Algebra, TTL and many other languages and techniques have been used to specify and/or verify this problem.

4.1 Synchronous Estelle specification of the Steam Boiler Problem

We have used this problem to test the Synchronous Estelle language and related tools. We will specify every physical part of it and the controller itself as different synchronous modules, while the communication network will be specified with a Standard Estelle module. All the synchronous modules will be connected to the network module to exchange information. Modules representing physical parts of the system are connected with the environment too, in order to get information or to send commands. All messages emitted by the physical units are sent to the control program. The messages emitted by the control program are sent only to one physical unit.

4.2 General structure of the specification

The first step in our specification is to define the main modules of the system and the interconnection channels between them. All the channels will connect every module with the *network* module. This module has to route every message to the proper destination. The declaration of the channels is very simple: it only enumerates the messages that can travel through them.

Once defined the channels, we have to specify the header of the modules. We include in the header definition of each module the interaction points of the

module with the environment (when the module interacts with the physical units) and with the other modules.

In the main body of the specification the configuration of the system is included associating the headers of the modules with the appropriate bodies and connecting the interaction points of the different modules.

The rest of the specification consists on the definition of each reactive module and one Standard Estelle module to specify the interconnection network.

4.3 Specification of reactive modules

There are six main types of reactive modules in our specification, representing different parts of the system: the water level measurement device, the steam measurement device, the pump, the pump control device, the valve and the control program. There are four pumps and pump controllers, so there will be four instances of those kind of modules.

The behaviour of the physical units is really simple: sensors send periodically read values to the control program, while actuators send signals to the environment to change the state of some physical unit (pump and valve). The modules dealing with the physical parts of the system have to manage the errors, too. When the system control program detects a failure in some physical part of the system it will send a message to notify the failure. The controller of this part then stops the normal behaviour until the device is repaired: Any message received will be ignored, and no message will be emitted until the physical device is repaired. Then, the physical device will notify this event to its controller, which will report it to the system controller.

4.3.1 Control module

The core of the steam boiler program is the Control module. This module represents all the logic of the system: it has to accept read values of system parameters (water level, steam level, state of the pumps) and it has to decide what to do in each moment to maintain a safe water level. The controller has to deal with potential failures of the physical units too. Therefore this is the most complex module of the specification. It is constituted by four submodules that run in parallel:

- *Read_messages_ini*: used only during the initialization phase to deal with initial message exchanging.
- *Read_messages*: reads messages sent by physical units, and estimates the present value when some physical sensor does not work properly.
- *Timing*: periodically sends a signal to notify that it is the time to react to the physical units messages.
- *Operation_modes*: represents different modes of operation: *Normal*, when everything works properly; *Rescue*, *Degraded* when there are some physical failures; *Emergency_stop*, when the system has to be shut down because the

program can not guarantee the safety of the boiler; and *Initialization*, for this purpose.

All those modules (states) are decomposed in more complex state machines. To give an idea of the complexity of the control module we can say that it has 160 submodules, 154 transitions and 52 Estelle transitions. Besides states and transitions the specification should be completed with pieces of Pascal code to define the actions to perform when the transition fires or when an state is entered or exited.

In [27] the complete specification of the system, both in graphical and in textual way can be found.

4.4 Specification of network module

This module is used to represent the behaviour of the communication system between reactive modules. Communication through a network is based on message interchanging, and this is the mechanism Estelle uses instead event broadcasting using in synchronous modules. So, Standard Estelle has been used to specify this module because it is more suitable for this kind of behaviour.

A very simple protocol has been used in this first approach (messages are sent to their destination without any further considerations). More complex and real protocols can be used to study real systems, but in this paper we are only interested in studying the applicability of Synchronous Estelle to distributed reactive systems, not in a real, efficient and fault tolerant specification. Further refinements of this first work can improve and complete the specification to become more robust.

5 Conclusions and future work

This paper has introduced a new language: Synchronous Estelle. This language is not just another synchronous language, but it tries to merge in one single framework both synchronous and asynchronous paradigms to specify distributed reactive systems. We have shown its syntax, semantics and the way to get executable specifications.

The main contribution of this approach is to put together in only one specification both different kinds of behaviours: the synchronous behaviour for the reactive subsystems and the asynchronous one for the communication part of the system. The new technique to specify distributed reactive systems has been tested on the Steam Boiler Control Problem, showing that it is very suitable for this kind of systems.

The developed specification for the Steam Boiler Control Problem has been compiled with the Synchronous Estelle Compiler, obtaining a set of C files that implement a fully executable model of the system. This control program has been successfully linked to the steam boiler simulator developed in the FZI[24].

At this point of time we are working in verifying the Synchronous Estelle specifications. As we have said in Section 2.7, we are working on a translator from Synchronous Estelle to PROMELA to do that.

References

[1] Abrial, J.R., *Specifying and Programming the Steam Boiler Control*. Springer-Verlag Lecture Notes in Computer Science, **11654** (1996).

[2] André, C., *SyncCharts: A Visual Representation of Reactive Behaviours*. Research Report 95-52, Université Nice, Shopia Antipolis, 1996

[3] Ballarin, F, M. Chiodo, P. Giusto *et al*, *Hardware-Software Co-Design of Embedded Systems: The Polis Approach*, Kluwer Academic, 1997.

[4] Benveniste, A., and G. Berry, *The Synchronous Approach to Reactive and Real-Time Systems*. Proceedings of the IEEE, vol 79, n 9, pp.1270- 1282, 1991

[5] Berry G., and G. Gonthier, *The Esterel Synchronous Programming Language: Design, Semantics, Implementation*. Technical Report, 1988. Reprinted in Science of Computing Programming, **19** (1992), no.2, 83-152.

[6] Berry, G., S. Ramesh and R.K. Shyamasundar, *Communicating Reactive Processes*. Proc. 20th Symp. on Principles of Programming Languages, 1993

[7] Boniol, F., and M. Adelantado, *Programming Distributed Reactive Systems: A Strong and Weak Synchronous Coupling*. Proc. of WDAG'93, pp. 294-308. 1993

[8] Boussinot, F., *Reactive C: An extension of C to Program Reactive Systems*. Software Practice And Experience, vol.21(4), pp. 401-428, 1991

[9] Boussinot, F., J. Susini, and L. Hazard, *Distributed Reactive Machines*. Inria Research Report 3376, 1998

[10] Caspi, P., A. Girault, D. Pilaud, *Automatic Distribution of Reactive Systems for Asynchronous Networks of Processors*, IEEE Trans. on Software Engineering, 25(3), pp.416-127, 1999.

[11] Chapiro, D.M., *Globally Asynchronous Locally Synchronous Systems*, PhD Thesis, Stanford University, 1984

[12] CCITT, *Specification and Description Language*. International Consultative Committee on Telegraphy and Telephony, CCITT Z.100, Geneva.

[13] Halbwachs N., P. Caspi, P. Raymond and D. Pilaud, *The Synchronous Data Flow Programming Language LUSTRE*. Proc. of the IEEE, vol 79, no 9, 1305-1320, 1991

[14] Harel, D., *Statecharts: A visual Formalism For Complex Systems*. Science of Computer Programming, 1987.

[15] Heitmeryer, C., R. Jeffords and B. Labaw, *A benchmark for comparing different approaches for specifying and verifying real-time systems*. In Proc. 10th Int. Workshop on Real-Time Operating Systems and Software. May, 1993.

[16] Holzmann, G. J., *The Spin Model Checker*, IEEE Trans. on Software Engineering, **23** (1997), No. 5, pp. 279-295.

[17] Huizing, C., *Semantics of Reactive Systems: Comparison and Full Abstraction*. Ph.D. Thesis, Eindhoven University, 1991.

[18] Institut Nacional des Telecomunications. *EDT: Estelle Development Toolset*, Evry, France, 1996.

[19] International Organization for Standardization, *ESTELLE - A Formal Description Technique based on an Extended State Transition Model*, ISO/IEC 9074, Geneva, 1989

[20] International Organization for Standardization, *LOTOS - A Formal Description Technique based on the Temporal Ordering of Observational Behaviour*, ISO/IEC 8807, Geneva, 1989

[21] Le Gernic, P., T. Gautier, M. Le Borgne and C. Le Marie, *Programming Real Time Applications with Signal*. Proc. of the IEEE, Vol. 79, No 9, Sep. 1991

[22] Leveson, N., M. Heimdahl, H. Hildreth and J. D. Reese, *Requirements Specification for Process Control Systems*. IEEE Transactions on Software Engineering, **20** (1994), No. 9

[23] Lindner, T., *Task Description of the case Study Production Cell*. Technical Report. Forschungszentrum Informatik. Karlsruhe, 1993

[24] Lötzbeyer, Annette, *Simulation of a Steam-Boiler*. Technical Report. Forschungszentrum Informatik. Karlsruhe, 1994.

[25] Maraninchi, F., and Y. Rémond, *Argos: an automaton-based synchronous language*. Computer languages 27, pp. 61-92, 2001

[26] Marzullo K., and M. D. Wood, *Tools for Constructing Distributed Reactive Systems*. Technical Report. Cornell University, Department of Computer Science, Ithaca, New York, 1991

[27] Riesco, M., *Specification of Distributed Reactive Systems Using Synchronous Estelle*. Ph.D. thesis, University of Oviedo, Oviedo (Spain), 2002. Available via `http://www.di.uniovi.es/~albizu/tesis`.

[28] Riesco, M., J. Tuya and O. Alonso, *Synchronous Estelle: A language for Specifying Distributed Control System*. In International Workshop on the Formal Description Technique ESTELLE, pp. 237-244. Evry, France, 1998

[29] Tuya, J., J. R. De Diego, C. de la Riva, J. A. Corrales. *Dynamic Analysis of SA/RT Models using SPIN and Modular Verification*. The Spin Verification System. J-C Grégoire,G.J. Holzmann, D.A. Peled, Eds., American Mathematical Society, Vol. DIMACS/32, pp. 165-183, 1997