A Synchronous Approach to Reactive System Design[1]

Charles André

I3S Laboratory – UNSA/CNRS
BP 121, F-06903 Sophia Antipolis cédex
T.(33) 492 942 740; F.(33) 492 942 896; E. andre@unice.fr.

Abstract

Reactive systems are systems that maintain permanent interactions with their environment. In this paper, we present our experience teaching discrete-event reactive systems to Electrical Engineering students. The course ranges from logic circuits to software components, and covers models, analysis, design of various systems (from logical systems to embedded systems). Since our students are not especially proficient in software engineering, we have preferred synchronous languages to general purpose languages for reactive system programming. In the paper, we discuss the rationale for choosing the synchronous paradigm.

1. Introduction

Embedded computer systems are more and more present in everyday life. Many of them perform control or monitoring, often under severe constraints, and their failure may have dramatic consequences. Another feature of these systems is their increasing complexity.

Students in Electrical Engineering must be taught how to model, design, implement, and test such systems. This paper presents our educational experience in this field, and a solution we propose to cope with realistic, and therefore often complex, modern reactive systems. We restrict our teaching to control-dominated systems, that is discrete-event systems with little data processing, but with complex behavior. Of course, there exist other controllers involving heavy real-time data processing like telecommunication or image processing applications. They are beyond the scope of our teaching. In what follows, a “controller” will refer to a control-dominated system.

Since controllers must react to all stimuli in a predictable and timely way, their analysis and their design must be rigorous and based on well-founded methods. So, our teaching has to address the following points:

- *Demand for precise and powerful models*. The models should capture both sequential and concurrent evolutions. To cope with complex systems, they should support hierarchical descriptions. Moreover, since “exception” is often the rule in reactive systems, they should offer facilities for exception handling specification.
- *Mathematical foundation* that makes it possible to formally analyze the models and guarantee critical properties.
- Efficient and safe techniques for implementation.
- *Reusability* (notions of components and architecture).

The paper is organized as follows: We first set forth our educational objectives. We

[1] This paper was presented at the 12th EAEEIE Annual Conf., 14-16 May 2001, Nancy (France).

then give the outline of our course. The fourth section presents the rationale for introducing synchronous languages. Finally, we mention what software engineering and especially the UML can bring to our teaching.

2. Educational objectives

Through the teaching of reactive systems, we aim at three goals for students: 1) to enrich their *theoretical background*, 2) to bring out new *skills*, 3) to open them to *innovation*.

Theoretical background

We teach fundamentals like Boolean Algebra, Finite State Machines, Concurrency (Petri nets), …

Skills

We encourage students to experiment with labs, in order to develop their skill in design, programming, and testing.

Innovations

Innovations are introductions to new concepts and methods, beyond the classical knowledge in E.E.: synchrony, modern design, objects, model-checking.

3. Contents of the course

The course starts with logical circuits and evolves to software solutions. Below, we mention the main chapters and their contents.

3.1. Classical design

We first study the logical design of combinational and sequential circuits. Expected behaviors are expressed either by state graphs or by Boolean recurrence equations.

We then tackle complex synchronous systems. Instead of a state-based model, we adopt an activity-oriented specification of the behavior. The system is seen has a sequencer (control part) that controls functional units (operative part).

Finally, we consider programmable controllers (PLCs, DSPs, micro-controllers, microprocessors). They generalize the previous class, making programming a central activity.

3.2. From circuits to programs

Thus, due to the increasing complexity of the systems to be controlled, and the technological evolution, we observe a shift in models, methods, and techniques. Electrical Engineering students of today might think that designing a controller is only a special application of programming, and that logic design is outdated.

Clearly, modern design, including software/hardware co-design, heavily relies on software environments. Unfortunately, most Electrical Engineering students lack the knowledge and the skills in software engineering that are now necessary.

To address these deficiencies, we have introduced the *synchronous languages*. The rationale for this choice is developed in the next section.

4. Rationale for teaching the synchronous approach

Synchronous languages [7] are dedicated to control, they are specialized languages. This section analyzes why we have preferred these languages to general purpose languages (GPL).

4.1. Simple languages

Being dedicated to specific applications, specialized languages are usually simpler than GPLs. A synchronous language has a restricted set of constructs and operations. For instance, Lustre [5], a declarative synchronous language, has only 4 specific primitive operators dealing with flows. The syntax of the Esterel language [4] can be kept in one A4 sheet of paper.

Another cause of simplicity is that synchronous languages deal with static structures and control: there is neither dynamic allocation, nor arbitrary recursion.

4.2. Languages tailored to control

Patching a classical language with missing constructs leads to unsatisfactory solutions (*e.g.*, dealing with concurrency, communication and synchronization in C programs, with the help of a RTOS). Esterel has a sequence and a parallel operator, and special constructs for pre-emption. Key words like `await` or `abort` … `when` … are well-suited to express event-driven behaviors. Declarative languages like Signal [10] and Lustre offer facilities for sampling, over-sampling, down-sampling.

4.3. Execution: Reactions

The way a synchronous program reacts is familiar to E.E. students because it is close to the clocked sequential circuit running mode. A synchronous program is executed as a sequence of reactions. Each reaction is associated with an instant. A reaction is atomic: it runs to its completion, and the incoming information is kept invariant during the whole reaction (one instant). Moreover, a reaction is usually “fast”, more precisely, sufficiently fast with respect to the time constants of the system to be controlled. This is an essential assumption to use synchronous controllers (be it hardware or software) in real-time control.

Note that, contrary to E.E. students, Software Engineering students are often disturbed by this mode of execution.

4.4. Mathematical semantics

Synchronous languages rely on mathematical semantics: the semantics can be “computed”. Underlying models are simple and powerful mathematic models: finite state machines, systems of Boolean equations, recurrence equations …, all concepts studied in classical E.E. courses.

The semantics of the Esterel language is an especially interesting example. Each Esterel construct can be translated into an equivalent “circuit” representation. A circuit is composed of gates, registers, and wires. What an Esterel program does is easily interpreted in terms of what the associated circuit does. For details, refer to papers of G. Berry on the semantics of the language ([3], for instance). Subtle issues, like causality cycles, are closely related to propagation delays and electrical stability of circuits. These arguments are, by far, more easily understood by E.E. students than rewriting rules and fix-point theory.

4.5. Formal verification

Thanks to their formal foundation, synchronous programs can be formally analysed. This gives us a great opportunity to introduce or to revisit reachability, safety, liveness … properties and the associated analysis methods.

4.6. Software development environment

There exist software environments free of charge for educational uses [9]. Besides compilers, a synchronous platform provides simulation facilities and verification tools.

A synchronous program has a fully deterministic behavior. As a consequence, bug finding and testing are easier with synchronous programs than with classical asynchronous programs. The interactive simulation allows the student to test typical scenarios with source code debugging. This is an invaluable tool for understanding concurrent evolutions, instantaneous communication, and exception handling. Reachability and safety properties are established by (symbolic) model checking: a model-checker makes an exhaustive simulation of the program. The idea is to write an “observer”, a small reactive module, that is run in parallel with the controller to be tested. An observer is dedicated to a property; It emits a violation signal whenever the propriety is violated. The property holds if during the symbolic execution the violation signal is never emitted. If this signal can be emitted, the model-checker generates a sequence leading to the violating situation. This sequence is then replayed with the interactive simulator.

Synchronous languages are textual, but they also have graphical interfaces. Lustre and Signal allow control-block diagram descriptions. Esterel has now a graphical notation, known as SyncCharts [1].

4.7. Labs

For labs, students use 2 synchronous languages: Lustre and Esterel.

Lustre adopts a declarative style, convenient for data-flow processing. A Lustre program is a network of operators driven by data-flow. The behavior of a node (*i.e.,* a user-defined operator) is defined by equations. This is similar to using recurrence equations, but with additional facilities.

Esterel is an imperative synchronous language, dedicated to control-dominated discrete systems. Esterel statements are very convenient for expressing reactions triggered by event occurrences.

Students use also synchronous graphical models in their design. *Sequential Function Charts* (Grafcet) are widely-spread in industrial applications, but they have limited programming capabilities, and semantic weakness. *Statecharts* [8] are more interesting. They are a state-based representation of behavior supporting sequencing, concurrency and hierarchy. Students have to be familiar with this model that tends to become a standard, through the UML. Finally, students use *SyncCharts*, the Esterel graphical companion model. This model is similar to statecharts but with additional capabilities and a full semantic compatibility with the Esterel language, so that all methods and tools available in the Esterel environment, apply to SyncCharts, as well.

In order to test their controllers, students use a series of software simulators we have developed [2]. A controller, not necessarily written in a synchronous language, is connected to the software simulator via a socket. Simulators cover various applications: electronic interfaces, washing machine …, (simplified) flexible manufacturing system. Of course, these animations are good for checking interactive behaviors, not for hard real-time constraints.

5. Borrowing from Software Engineering

For long, Embedded System programming had not benefited from the best practices of software engineering. Many applications had been written in assembling languages. Most of today's implementations are still C programs, calling services of some real-time operating system. Fortunately, object-oriented modeling is now entering the world of embedded and real-time system (see, for example, "Doing Hard Time" of B.P Douglass [6]).

Even if we are reluctant to adopt an object-oriented language for E.E. students[2], we believe that the Unified Modeling Language (UML) is worth being introduced. The UML is general enough to be used even though the implementation language is not an object-based language. UML is about modeling, not about programming. Recently, the Object Management Group (OMG) has shown its interest in Embedded and Real-Time systems by issuing a "Request For Proposal" related to real-time systems [11].

[2] We are not at all opposed to OO Languages, but we think that in an E.E. curriculum, we do not have enough time to teach an O.O language in a correct and efficient way.

We use some of the various points of view offered by the UML. We especially focus on:

- *Use case* modeling (functional view),
- *Dynamic* modeling (statecharts, scenarios),
- *Object and Class* modeling (static structure),
- Collaboration,
- Deployment.

Since controllers are our main concern, “active objects” (i.e., objects having independent processing resources) are especially interesting. The present UML definition of an active object is not sufficiently precise. Forthcoming real-time UML profiles should make this concept more consistent. For the time being, components (be it hardware or software) such as those used in some Architectural Description Languages may be candidate to cope with the complexity of modern control applications. After all, architecture and components are concepts familiar to E.E. students.

5. Conclusion

In this paper, we have presented our experience teaching discrete-event reactive systems to Electrical Engineering students.

The course still contains *fundamentals* on logic circuits, binary algebra, finite state machines, Petri nets, …

With the emergence of more and more complex systems, including *Embedded* and *Real-Time* applications, our teaching has evolved *from circuits to programs*.

General-Purpose languages are not well-adapted to control-dominated system programming. To overcome this inadequacy, we have chosen to introduce *synchronous languages and formalisms* in our Electrical Engineering course. We have presented the rational for this choice. Briefly, it appears that these specialized languages are expressive, efficient, and easier to learn than general-purpose languages for E.E. students.

Besides these new programming languages, we have also introduced the Unified Modeling Language. The UML contains several concepts useful to deal with complex applications and tends to become a standard, even in Embedded and Real-Time systems.

References

[1] C. André, "Representation and Analysis of Reactive Behaviors: A Synchronous Approach”, *IEEE-SMC*, Computational Engineering in Systems Applications (CESA), 1996, pp 19-29.

[2] C. André, M-A. Peraldi-Frati, D. Gaffé, **"**Plateforme pour l’étude et la conception de systèmes automatisés**",** *Communication dans les Enseigne-ments d’ingénieurs et dans l’industrie (TICE’2000)*, 18-20 Octobre, 2000, Troyes (F), pp 121-126.

[3] G. Berry, “The Foundations of Esterel”, in “Proof, Language, and Interaction: Essay in Honor of Robin Milner”, *Editors: G Plotkin, C. Stirling, and M. Tofte*, MIT Press, 2000.

[4] F. Boussinot, R. De Simone, " The Esterel Language”, *Proceeding of the IEEE*, 79(9), 1991, pp 1293-1304.

[5] P. Caspi, N. Halbwachs; D. Pilaud, P. Raymond, "The Synchronous Data Flow Programming Language Lustre”, *Proceeding of the IEEE*, 79(9), 1991, pp 1305-1320.

[6] B.P Douglass, “Doing Hard Time: Developing Real-Time Systems with UML, Objects, Frameworks, and Patterns”, Addison-Wesley, 1999.

[7] N. Halbwachs, “*Synchronous Programming of Reactive Systems*”, Kluwer Academic Publishers, Amsterdam, 1993.

[8] D. Harel, "Statecharts: A visual formalism for complex systems”, *Science of computer programming*, 8, 1987, pp 231-274.

[9] http://www.esterel.org

[10] P. Le Guernic, T. Gautier, M. Le Borgne, C. Le Maire, “Programming Real-Time Applications with Signal”, *Proceeding of the IEEE*, 79(9), 1991, pp 1321-1336.

[11] OMG, *“UML profile for Scheduling Performance and Time. Request for proposal (ADTF RFP-9)”*, OMG document number : ad/99-03-13, April 1999.