

# Modeling and implementing critical real-time systems with SyncCharts/Esterel

This article presents SyncCharts / Esterel, a product developed by Simulog, to be used for the specification and the development of critical real time systems.

SyncCharts is a graphical notation used to represent hierarchical state diagrams designed by Charles André at the University of Nice Sophia Antipolis.

Esterel is the name of a synchronous programming language and its compiler, both designed at Ecole des Mines de Paris and INRIA under the guidance of G. Berry. Esterel is also the name of famous hills on the French Riviera, nearby Sophia Antipolis where the language was designed.

The full SyncCharts/Esterel product, which is described in this article, is now available from Simulog in its version 1.2

SyncCharts / Esterel provides an easy way to go from a state transition diagram to a code implementing it. Figure 1 illustrates the typical steps you need to follow.

Let's now answer some fundamental questions about this approach:

- What is a reactive system?
- What are synchronous languages and why use them?

- What are SyncCharts main features and benefits?

## REACTIVE SYSTEMS

Reactive systems include real-time systems (often embedded), communication protocols, man-machine interfaces, etc. The usual definition of a reactive system is that « it maintains permanent interaction with its environment ». In this context, interacting means receiving input information, called events, and reacting

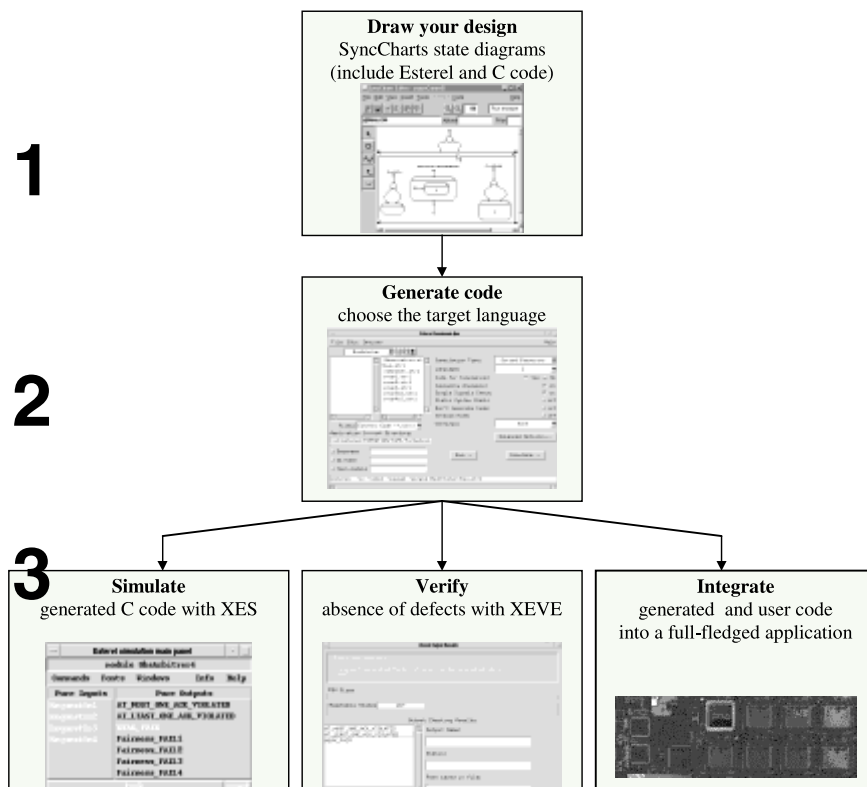


Figure 1. Development of a critical real-time system with SyncCharts/Esterel

# **AD FORCE COMPUTERS**

# DEV. ENVIRONMENT

by producing output information, also called events. Permanently means this process is on going, whether at regular time intervals or not. Hence, reactive systems and their environment work in tight cooperation, as shown on the Figure 2 below. The outputs produced by reactive system are interleaved with the inputs they receive, as time elapses. Indeed the response it should produce actually depends on the dates and sequences of received events. For instance, a reactive system may well receive the same event twice but produce two different answers. In other words, reactive systems have memory, needed to keep internally an image of the state of their environment.

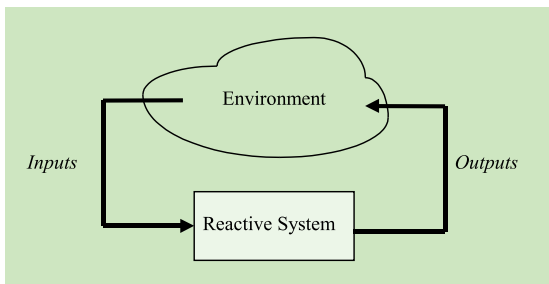


Figure 2: A reactive system

The term « reactive » was invented by D. Harel and A. Pnueli to distinguish such systems from others of a more algorithmic nature that they called transformational systems: transformational systems simply read their inputs, compute, and finally produce their outputs and terminate. Reactive systems must always react to new input; most of them never terminate (a controller in a nuclear plant is expected to run forever). The following Figure 3 illustrates that a reactive system maintains a state depending of the history of received events, while a transformational system has no memory and just computes the outputs as a function of its inputs.

Another important characteristic of reactive systems is that they should always be ready to re-pond to the evolutions of their environment, which imposes its speed. This implies a reactive system must acquire

new events as they are produced by its environment, and respond in due time so that it has useful influence in return.

In this respect, reactive systems are different from the interactive ones that respond to events at their own speed and thus may cause their environment to wait, like an operating system.

Reactive systems are very common in everyday life: they are found in cars, washing machines, portable telephones, signal processing systems, on-board electronics in defense and aerospace systems, device controllers etc.

## REAL-TIME SYSTEMS SPECIFIC REQUIREMENTS

Real-time systems induce specific needs, not only because of their specific timing constraints, but also because of the very nature of the applications using them.

The modeled system will be embedded in a real application, either software or hardware. Therefore, the produced code must be easily interfaced with a software environment such as a real time operating system (RTOS), a micro controller or onto specific hardware (ASIC, FPGA). Usually, the target is constrained in memory size, power consumption etc.

The modeled system must process inputs and return outputs at a speed compatible with the environment requirements. More importantly, it should be possible to measure response times, and especially bound worst case response time in order to be sure of correct operation according to the application needs. A correct value delivered too late is an incorrect behavior. Hardware clock regulated designs usually exhibit good predictability (critical path analysis), but software programs can prove more difficult to analyze. The expanding role of software in embedded systems calls for specific methods and tools to address this requirement.

Usually a reactive and real-time system is composed of several subsystems that behave independently to some extent: parallelism should be used to express

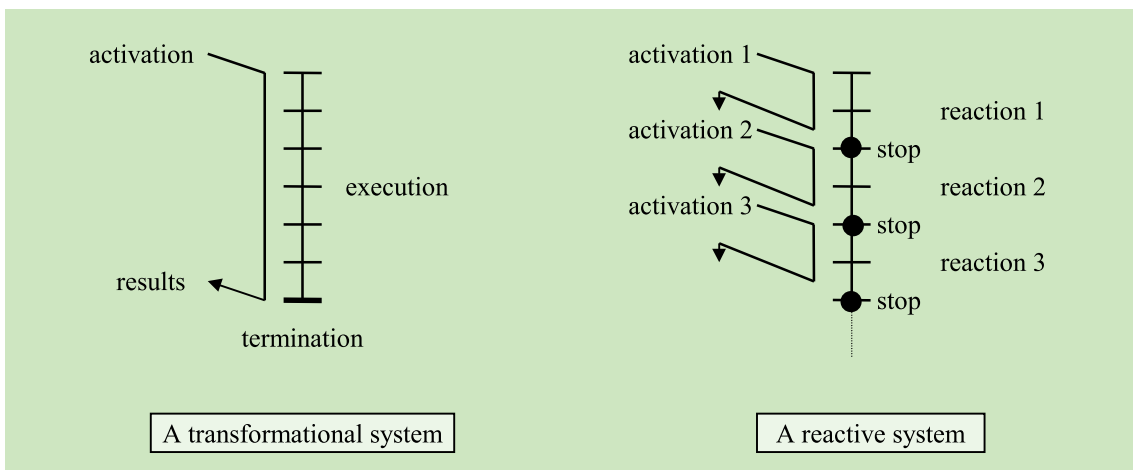


Figure 3: Transformational vs. reactive systems

the global behavior as a composition of the behaviors of the parts. One should also remember that the system as a whole and its environment itself are two concurrent processes. However, subsystems are not completely independent: they need synchronizing and communicating to ensure some degree of interaction. Finally, because of the operational system requirements, it may be needed to partition the control over several physically distinct hardware or software components. For all these reasons, behaviors need to be expressed as compositions of simpler behaviors, using parallelism and communication.

Critical systems must be deterministic, for safety reasons. This implies that the model should fully and unambiguously define all possible behaviors of a modeled system. The behavior should not depend on some particular implementation or compilation strategy. An obviously expected - but rarely satisfied - property is that simulation and embedded code should behave exactly the same. Computer design tools should not interfere with this property: one wants the insurance that two similar model parts drawn in a different order are indeed equivalent and that the result never depends on some ordering or prioritization not visible in the model.

The embedded system must be robust (no bug leading to a crash or a never terminating process). The embedded system must be permanently able to answer to the environment events, so it should never be out of order or at least it should be fault tolerant. For the same reason, any performance degradation should occur gracefully, predictably and in an as localized fashion as possible.

### **THE SYNCHRONOUS APPROACH OF THE SYNCCHARTS/ESTEREL PRODUCT**

Reactive systems modeling and programming has proven difficult and error prone using traditional software programming methods. This is why synchronous languages have been designed.

Esterel is a synchronous textual programming language, while SyncCharts is a synchronous graphical formalism. Other formalisms and tools can be used to design and program reactive systems. These do not address all the requirements of critical real time systems as well as synchronous languages do.

It should be noted that synchronous programming languages share the synchronous model with sequential digital circuits that cyclically «read» their input pins, «compute» and «write» on their output pins, at a given clock speed. The SyncCharts/Esterel compilers and associated tools (verification, optimization) do actually use this analogy by reusing several powerful algorithms used by EDA (Electronic Design Automation) tools. Using this technology enables SyncCharts/Esterel to cope with large designs.

### **THE ESTEREL PROGRAMMING LANGUAGE**

The Esterel programming language was especially tai-

lored to model complex reactive behaviors. Reactive systems specific characteristics, such as parallelism, communication and abortions are well addressed in Esterel since the language has specific high level constructs for each of them. For instance, parallelism is a first class concept and a simple operator in the language that composes well and at any level with other operators like communication and abortion.

Esterel is based on a rigorous mathematical semantics. This is very important in order to apply formal verification methods to Esterel programs. It is possible to track and eliminate defects in critical systems by applying formal verification tools, until satisfactory safety level has been proven.

Esterel programs are compiled in an intermediate electronic circuit representation, onto which powerful hardware circuit analysis and optimization can be applied. Thanks to compact representation of such systems (using Binary Decision Diagrams), the Esterel compiler is now able to deal with large programs (from version 5 in 1995). A system of over 1014 states has been modeled, analyzed and compiled onto a simple 68000 family processor. Code can also be generated for FPGAs or other hardware targets in VHDL form for instance.

The generated C code is compact and fast; hence compiled code can fit onto embedded targets. The produced code is simple, does not have dangerous constructs like recursion, while loops or run-time memory allocation and tasks switches. This makes the produced code reliable, easy to analyze and to measure. Moreover, careful static timing analysis (worst case for instance) of the reactive kernel code produced by Esterel can be carried out with a good confidence level thanks to the simplicity and regularity of the produced code. Indeed, critical path analysis can be done much in the same way as for sequential circuits, except that software and cache policies are less predictable. Apart from graphical instrumentation, the generated reactive kernel is exactly the same for simulation and final target (since it is small). This ensures that what you specify is what you execute.

### **TIME IN THE SYNCHRONOUS APPROACH**

The main novelty brought by the synchronous approach for programming real time and reactive systems is the discrete model of time they adopt: the system repeatedly receives inputs and produces outputs. Such a sequence -- read inputs, computes, produces outputs -- is called a reaction, or an instant. The system internally has no notion of time, is merely captures activation. The system can count the inputs it has received and how often it has been activated, but it cannot capture the physical time elapsed between activations, nor the physical time elapsed during internal computation of a reaction. Reactions are numbered, as well as external events. The system keeps track of ordering, sequences, after, and before. In other words a synchronous program is a sampled system.

In order to model physical time in synchronous languages, one has to devise a clock mechanism that will

## DEV. ENVIRONMENT

activate the system at regular time intervals and thus provide it with a notion of sampled time. For instance, say a synchronous program is activated every 50 microseconds: 50 microseconds is then its finest approximation for appreciating real time. However, assuming this clock is precise, the system can synthesize internally the millisecond and use it as another clock, simply by counting 20 clock ticks.

A consequence of this logical view of time is that any event can be used to measure time. A synchronous program can trigger a timeout after a number of microsecond events have been received or after two mouse clicks have been received in the same manner. This property of treating any external event as a potential clock is very useful when modeling controllers reacting to a-periodic event sources, like human commands for instance. The system can be activated only on demand and kept idle otherwise.

Synchronous languages assume that reactions should not overlap, that is each reaction should be the atomic calculation of the output response to an event (a set of input signals), that has been sampled, collected and treated. Of course, this assumption should be checked, that is one should ensure reactions do not take too long so that no input is lost and the system operates at a speed matching the requirements of the application. But synchronous languages insist on keeping these two concerns separate, and to clearly model the desired logical behavior of the system prior to addressing performance constraints. This is common practice in hardware design of sequential circuits where one addresses the logical description of the system and then ensures that the external clock is compatible with the circuit good operation. Because of this model, powerful description and analysis can be carried out to ensure a design respects correct logical behavior of the system.

### THE ADVANTAGES OF SYNCHRONOUS PROGRAMMING LANGUAGES

The synchronous programming method consists in separating the logic of the dynamic behavior from sampling and real time constraints, in order to gain clarity, safety, programming modularity and productivity as well as efficiency and compactness of compiled programs.

### THE GRAPHICAL METHODS OF SYNCCHARTS

Textual synchronous languages are mainly used by programmers, but control engineers and system designers will often prefer graphical description of their system, especially in the early phases of specification and development. Several graphical methods exist (Graphcet, SA/RT, Statecharts, UML/RT...), but they are not so well suited to model reactive systems. Either because of some ambiguities, lack of rigorous semantics, it is almost impossible to apply formal verification techniques when using these graphical formalisms and their supporting tools. For the same reasons, it is often impossible to generate efficient target code. The SyncCharts formalism was then created to recover all

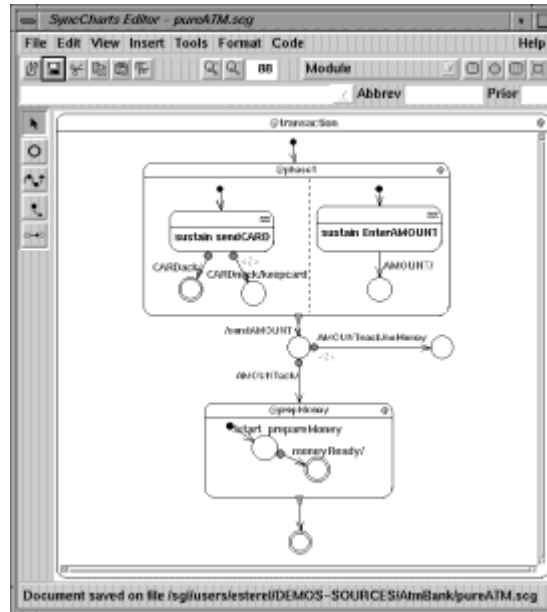


Figure 4. The SyncCharts description of an Automatic Teller Machine (ATM)

these possibilities, by adopting an unambiguous notation with strong semantics, and by following the synchronous model, which permits to benefit from the strengths of the Esterel language and compiler.

The SyncCharts formalism relies on state diagrams a very commonly used visual method to specify a state machine. A state machine describes what a system should be doing under certain conditions and how the system should evolve according to the previous state it was in. Finite State Machines (FSMs) are well suited to specify and design reactive (sub)-systems, such as controllers, especially when safety and determinism are wanted properties. In this respect, SyncCharts resemble to the Statecharts formalism, and other notations widely used in object oriented methods (OMT/UML).

Figure 4 exhibits a model of an Automatic Teller Machine, a banking terminal application. Figure 5 is an

```
Breakpoints Font Father Tree MainPanel Close
module phase1:
  input CARDack;
  input CARDnack;
  input AMOUNT;
  output keepcard;
  output sendCARD;
  output EnterAMOUNT;
[
  abort
  [sustain sendCARD];
  halt;
  when
  case [CARDack] do
  nothing;
  case [CARDnack] do
  emit keepcard ;
  halt;
  end abort ;
  |
  weak abort
  [sustain EnterAMOUNT];
  halt;
  when [AMOUNT] ;
  halt;
];
end module
```

Figure 5. The instrumented ESTEREL generated code for the ATM

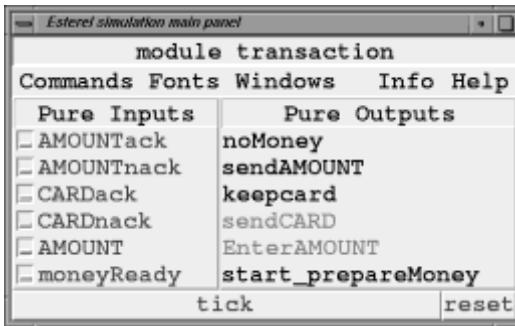


Figure 6. The simulation main panel for the ATM

instrumented display of the generated Esterel code. Figure 6 shows the simulation main panel, which can be used to launch simulations of the model and to instrument it after having used the formal verification tool XEVE in order to understand why some desirable properties of the model (and therefore of the application) are not satisfied.

The SyncCharts notation has been restricted to be fully compatible with the synchronous approach of Esterel and because of this, fully analyzable, so that all possible behaviors of the systems are, in principle, explorable. The advantage of synchronous models is that they permit the analysis of large state spaces, thanks to powerful algorithms.

### SOME WORDS OF CONCLUSION

As a conclusion to this article, we have shown that the SyncCharts / Esterel product provides a strong and unique synchronous framework that brings to the developer of critical real-time systems a set of highly desirable possibilities:

- shortened specification and development times
- deterministic behavior of the resulting system
- possibility to formally verify properties of the system
- possibility to automatically generate embedded code

Current users of the method and tools include Dassault Aviation in the aerospace sector, Thomson CSF Communication for the development of telecommunication protocols and Texas Instruments for the simulation and verification of DSPs hardware designs. ■

---

*Bernard Dion joined Simulog in 1994 and is currently the Executive Vice-President of Simulog, in charge of Software Products. He has received a Ph. D. degree from the University of Wisconsin at Madison in 1978. He has formally worked with Bull in the Ada development team and he has been the founder of the CERICS software engineering school in Sophia Antipolis. Sylvain Dissoubrey joined Simulog in 1987 and is currently the SyncCharts/Esterel Product Manager. He has received Mastère degree from CERICS in 1987 and a Ph. D. degree in Computer Sciences in 1995 at The University of Nice Sophia Antipolis.*

## AD EMBEDDED SYSTEMS GERMANY 2