Synchronous Thread Management in a Distributed Operating System's Micro Kernel

Olivier Potonniée[1], Jean-Bernard Stefani[2]

[1] Alcatel Alsthom Recherche, Route de Nozay, 91460 Marcoussis, France,
[2] France Telecom/CNET, 38-40 av. du Gal Leclerc, 92794 Issy Moulineaux Cedex 9, France

Abstract. This paper describes an experiment in programming part of an operating system kernel using the Esterel synchronous programming language. Using a synchronous programming language allows the construction of provable, deterministic reactive systems. The paper describes and analyzes the small executive realized and the formal verification of some of its properties. It also presents how multiple interconnected instances of this executive can be synchronized, yielding a distributed real-time platform operating under a sparse-time model.

Key Words : Synchronous programming, distributed systems, thread management, real-time systems, deterministic systems

1 Introduction

Synchronous languages [2] such as Signal [3], Lustre [4], Esterel [5] or Argos [6], provide a means to program real time systems having a formal description of their (deterministic) behavior. One potential interesting area of application is the construction of a real time operating system kernel.

Current operating systems are built using standard languages, which make it difficult to formerly verify their behavior. In this paper, for the first time to best of our knowledge, we use a synchronous programming language, Esterel, to develop a small real time executive corresponding to the thread management function of the Chorus micro-kernel. We used the Mauto tool [7] to verify properties of our implementation through bisimulation reductions.

Based on our synchronous executive, we further implemented a ditributed computational model, originally proposed in the Saturn project [8]. The resulting distributed platform realizes a sparse time execution model as proposed by H. Kopetz [9], where events can only occur in specified intervals of time, thus simplifying the synchronization of distributed applications.

2 Overview of the prototype

We did not aim to reinvent thread management, but only to investigate a new way to realize it. We thus reproduced that of the Chorus [10] micro-kernel.

Thread management in Chorus comes with a specific scheduler which is preemptive and piority driven : threads having a priority higher than a given threshold can only be preempted by higher priority threads, while the others may share time with equal priority threads.

The analysis of an existing thread management is not an easy work : the documentation presents isolated possible states of a thread, but it misses information on the combination of those states. The kernel sources are usually the only way to get precise information on specific behavior. This is due to the large numbers of combined states that a thread can take (hundreds), that can't be documented and do not directly appear in classical programming languages. Esterel, being state oriented, provides a much simpler description of those states.

We decomposed Chorus thread management into four different components, corresponding to the four basic functions involved in thread management, namely interrupt management, handling of time, thread management proper, and scheduling. Each of these components corresponds in our prototype to one or more Esterel module(s). The structure of the prototype, together with the flow of communication between the different components, is shown in figure 1.

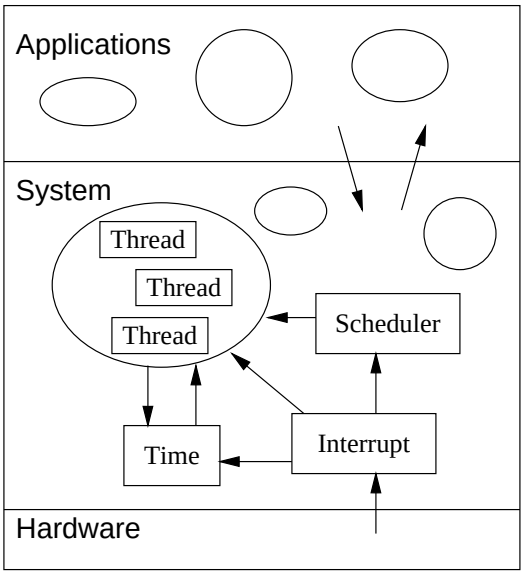

Fig. 1. Architecture of the synchronous process management

- **Interrupt**
  This component receives interrupt signals from the processor, and redirects them to the involved components. It contains two Esterel modules : one launches handlers to treat hardware interrupts, the other redirects software interrupts (traps) and exceptions to the concerned threads.

- **Time**
  The Time component handles the physical clock interrupt, and timers.
- **Scheduler**
  This component allocates the processor to the threads. It is divided in a generic module, and an interchangeable specific one, that implements the Chorus scheduling policy.
- **Thread**
  This is the largest component. It manages all possible states that a thread can take. Whereas each of the previous components was unique in the system, there is one instance of Thread per thread in the system.
  The number of states managed by this component being very large, it is decomposed in five modules : three of them handle states concerning different kinds of blocked states, and communicate a sub-state to a fourth module, that deduces if the thread is ready or not to execute. This information is transmitted to the scheduler. The fifth module handles transitions to and from system mode.

3 Implementation and verification

The compilation of an Esterel module produces an automaton coded in a set of C++ classes [12]. We call an instance of the automaton a *reactive object*. Ideally, all modules should be compiled together, providing a single object. However, this is not possible for two reasons :

- The resulting automaton would be so large that it would not even be possible to generate it due to a combinatorial explosion of states [3].
- Esterel does not allow dynamicity (which would require dynamic reconstruction of the automaton !). Thus the Thread component has to be compiled separately, so it can be instantiated at run time when new threads are created.

The interface of a reactive object provides methods to set input signals awaited by the automaton, and get output signals that it produces. An output signal can be connected to the input signal of another reactive object, ensuring synchronous communications between separately compiled modules. This enables to chain reactive objects, provided that they only have unidirectional communication, so that an order can be decided for the execution of the reactions [4]. We call this composition method *synchronous sequencing*. Our prototype has five reactive objects, linked by synchronous sequencing.

To execute those objects, it is necessary to have an execution machine, that will activate each object in the sequencing order, ensuring the synchronous semantic. This leads to the definition of a *synchronous execution machine*, which

[3] However, the new version of the Esterel compiler which was not available to us at the time, resolves this problem, by using a new coding structure for the automaton.

[4] Sequencing allows to have loop in the communication graph by introducing a *delay* object [12].

performs signal communications inside and at the interface of the thread management, and guarantees atomicity of the reactions. This execution machine being written in classical programming language, it has to be minimum, so that it does not contain unproven behavior. We adopted the simplest policy to start objects' activation : a reaction sequence is executed as soon as there is one input signal, unless there is already one running. There are two concurrent sources of events in an operating system : threads, invoking system operations or raising exceptions, and hardware interrupts. A lock mechanism is used to ensure atomicity of the reaction sequence when concurrent events occur simultaneously.

To certify a program, a formal verification, must be used. This is done with the Mauto tool [11], that uses a mathematical representation of the automaton, as a set of labeled transitions. To verify that a situation cannot occur, we define the abstract actions specifying it. Those actions are described by a sequence of combinations of input and output signals. Each combination can contain AND, OR and NOT boolean operators, allowing complex situations to be expressed. Given the automaton and one abstract action, Mauto certifies, using bisimulation reductions, that the transition system does not hold the undesired action.

Example 1. The property *"the Run signal cannot be emitted by the scheduler while it is in interrupted mode"* is verified by proving that the following abstract action is unreachable by the scheduler automaton :

true : /EnterInterrupt? : (not /ExitInterrupt?)* : (not /ExitInterrupt?) and /Run!*

For each Esterel module of our executive, its important properties are checked with this tool. However, the use of the sequencing composition instead of the synchronous parallel operator prohibits verifications of behavior implicating several modules.

4 A sparse time distributed platform

Based on our executive, we have implemented a distributed execution platform which realizes the sparse time execution model proposed by Kopetz [9]. We detail it in this section.

4.1 Different distributed execution models for reactive objects

The execution of multiple reactive objects in a distributed environment may follow different schemes :

- **Asynchronous** ; no guarantees on module execution time and communication delays.
  Two executions of the same system may lead to different results, depending on arbitrary parameters. It is an indeterministic system.

- **Weak synchronous**
  A logical time may be defined, that is shared by all the objects of the system. If the reaction of each object occurs at specified instants of that logical time, and if the communications between objects have a fixed length in that logical time, then the system is deterministic. This can be accomplished, for example, by a master clock, broadcasting ticks to all reactive objects, and waiting for their acknowledgment to advance to the next instant. Each object starts a reaction when it receives a tick, and emits its acknowledgment when it is sure to have received all external signals for the next instant.
  This is called the weak synchronous model in the literature, as first introduced by Milner in [13], and implemented in the Saturn project [8].
- **Timed weak synchronous**
  The logical time is anchored in the physical time. That is each instant of the logical time corresponds to a defined granularity of the physical time. The system is then time deterministic, i.e. its result is deterministic in value and time. A distributed system providing this determinism considerably eases the development of real-time distributed applications, that necessitate time guarantees. We present in the next subsection an efficient implementation of such a synchronization.

4.2 Offering Sparse Time to distributed applications

H. Kopetz, in [9], presented an execution model restricting event occurrences in specific points of a distributed synchronized time. The set of those points constitutes a sparse timebase. This model, when synchronized points are properly chosen, provides temporal order on events : an event E_1 is either earlier, simultaneous or later than an event E_2.

Our reactive process management, timely synchronized, constructs such a sparse timebase for the applications, realizing temporal order determinism of their events.

To determine the synchronization points, we rely on a classical clock synchronization mechanism, with a known precision p.

$$p = MAX((\forall i, (\forall k, (\forall l, \quad \|z(k_i) - z(l_i)\|) \tag{1}$$

where $z(e)$ is the time-stamp, on a reference clock, of the event e, and k_i and l_i are the i^{th} ticks of local clocks k and l.

We build a global timebase, with a granularity g_g superior to the precision p. Between two ticks of this global time, there is an interval K within which all clocks have the same value.

$$K = g_g - p \qquad (g_g > p) \tag{2}$$

Those K intervals are the synchronization points, defining a sparse timebase. If all events of our system occur during one of the K intervals, then they will be time-stamped with the same value on every machine.

We suppose the existence of a maximum event communication delay d_{max}. If an event e is emitted from k at $z(k_i)$, we are sure that all recipients have received it at $z(k_i) + d_{max}$.

A receiver time-stamps an event, with the global timebase, as soon as it receives it. For this value to be identical on all receivers, the emission must respect two constraints (represented in figure 2) :

- An event must not be emitted before $g(i) + p$, so that its receiver won't get it too soon (imprecision constraint)
- An event must not be emitted after $g(i+1) - p - d_{max}$, so that its receiver won't get it too late (transit delay constraint)

Those two constraints define the interval K :

$$K = g_g - 2.p - d_{max} \tag{3}$$

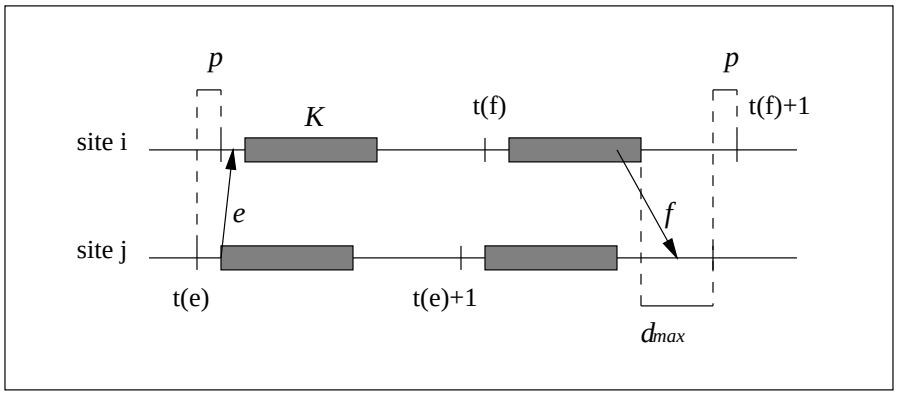

Fig. 2. Restricting emissions to K ensures uniform time-stamps

p and d_{max} values are fixed by the environment. g_g has thus to be determined to offer a sufficient K length to allow event processing and emissions.

In our implementation, all kernel instances start their reactions at the beginning of g_g, and must complete it and emit its resulting signals at least at time $g_g + p + K$. K value has hence to be superior or equal to the worst case execution time of this treatment, minus p. After their reaction, kernels receive signals emitted from other kernels. They will be taken into account in the next reaction, starting at next g_g interval. All kernels will then have the same set of message signals.

To prototype this model, we isolated the triggering mechanism of the synchronous kernel, making it replaceable. We implemented three triggers, realizing respectively the asynchronous, weak synchronous, and timed weak synchronous models. In the first one, kernels react as soon as they detect an input signal.

In the second, an external sequencer broadcast stimulation to start reactions, and collects acknowledgment of their completion. The last one implements the model presented above by relying on a physical clock synchronisation protocol.

5 Results and conclusion

Our implementation revealed that it was indeed possible to use synchronous programming to realize real time operating system kernels, gaining the inherent benefits : determinism and formal descriptions.

Our distributed implementation also showed how we could realize a distributed timed synchronous model, preserving the benefit of determinism and time determinacy in a distributed setting.

However, the following difficulties must be noted :

1. We had difficulties in using Esterel synchronous parallel composition, essentially due to the inability to dynamically create reactive objects. To circumvent Esterel limitations, we had to rely on an ad-hoc synchronous sequencing operator, implementing a weaker form of composition. Note that the recent reactive objects model proposed in [15] alleviates the problem by providing a new synchronous parallel operator which allows dynamic creation of reactive objects, at the expense of a slightly weaker semantic for interrupts (trap constructs in Esterel).
2. The implementation exhibited poor performances, compared to the original Chorus kernel. The sources of these inefficiencies lie mainly in the way events are implemented in the C++ library. The document [14] provides a detailed analysis of the sources of inefficiencies, and suggests ways to remove them.
3. Verification tools at our disposal did not allow us to verify global properties of the executive, because they did not take into account the weak parallel operator used in the implementation. Note however that these tools could still be used if we introduced bounds on the maximum number of threads allowed.

These difficulties, in turn, suggest different avenues for further research.

References

1. Ferrari, D.: Client requirements for real time communication services. Research Reports ICSI TR-90-007, Berkeley, California, USA, 1990
2. Benveniste, A., Berry, G.: The Synchronous Approach to reactive and real-time systems. IEEE, 1991
3. le Guernic, P., Gautier, T., le Borgne, M., le Maire, C.: Programming Real-Time Applications with Signal. Proceedings of the IEEE, 1991
4. Halbwachs, N., Caspi, P., Raymond, P., Pilaud, D.: The Synchronous Data Flow Programming Language Lustre. Proceedings of the IEEE, 1991

5. Berry, G., Gonthier, G.: The Esterel synchronous language : Design, Semantic, Implementation. Journal of Science Of Computer Programming, Vol 19, Num 2., pp87-152, 1992
6. Jourdan, M., Maraninchim, F., Olivero, A.: Verifying qualitative real-time properties of synchronous programs. International Conference on Computer Aided Verification, Elounda, 1993, LNCS697
7. Lecompte, V.: Vérification automatique de programmes Esterel. Ph.D thesis from Paris VII University, 1989
8. Boniol, F., Adelantado, M.: Programming distributed reactive communicating distributed reactive automata : the weak synchronous paradigm. International Conference on Decentralized and Distributed Systems, Spain, 1993
9. Kopetz, H.: Sparse time versus Dense time in distributed real time systems. IEEE Symp. On Distributed Systems, 1992
10. Rozier, M., Abrossimov, V., Armand, F., Boule, I., Gien, M., Guillemont, M., Herrmann, F., Kaiser, C., Langlois, S., Lanard, P., Neuhauser, W.: CHORUS distributed operating systems. Computing Systems 1(4), pp 305-367, 1988
11. Vergamini, D.: Auto/Mauto User Manual, version 2-3. INRIA CERICS, 1992
12. Boulanger, F.: Intégration de Modules Synchrones dans la Progammation par Objets. Ph.D thesis from Orsay University, 1994
13. Milner, R.: Calculi for synchrony and asynchrony. Theoretical Computer Science, 25(3), 1983
14. Potonniée, O.: Etude et prototypage en Esterel de la gestion de processus d'un micro-noyau de système d'exploitation réparti avec garantie de service. Ph.D thesis from Paris VI University, 1996
15. F. Boussinot, G. Doumenc, J.B. Stefani: Reactive Objects. Research Report 2664, INRIA, France, October 1995.