

Correct and Efficient Implementations of Synchronous Models on Asynchronous Execution Platforms*

Stavros Tripakis, University of California, Berkeley, stavros@eecs.berkeley.edu
Albert Benveniste, IRISA/INRIA, albert.benveniste@irisa.fr
Paul Caspi, VERIMAG/CNRS, paul.caspi@imag.fr
Claudio Pinello, UTRC, pinello@claudiopinello.net
Alberto Sangiovanni-Vincentelli, UC Berkeley, alberto@eecs.berkeley.edu
Christos Sofronis, PARADES, christos.sofronis@parades.rm.cnr.it

April 1, 2009

1 Synchronous vs. Asynchronous Concurrency

Concurrency is enjoying a second (or third? fourth?) youth. This is triggered by an anticipated revolution in the evolution of computer architecture, from faster, single-processor chips, to multi-processor chips with more and more processors [6]. This change brings hopes that the limits of Moore's law will be overcome, but also fears that programming will become even harder than it already is. Indeed, parallel computing has been one of the holy grails of computer science.

Concurrent programming today is predominantly *thread-based*. Thread-based concurrency is fundamentally *asynchronous*, in the sense that most thread-based models have an *interleaving* semantics, which stems from making few or no assumptions on the relative speeds of the threads. Thread communication is often based on some type of *shared-memory* model. This type of asynchronous concurrency results in non-deterministic behavior, which is hard to understand and debug. Indeed, a number of researchers claim that thread-based parallel programming is a bad idea [12, 9].

Other concurrency models, such as Kahn Process Networks [7], ensure deterministic results despite process interleaving. Unfortunately, most multi-processor architectures today do not follow such models, and use threads instead.

Synchronous concurrency is based on a model that avoids interleaving. Instead, all processes execute in *lock-step*. Many theoretical models but also widely-used practical systems exist that follow the synchronous paradigm. For instance, Milner's SCCS [11], the synchronous languages [2], languages such as VHDL and Verilog, which are very popular in the design and implementation of synchronous circuits, and tools such as Simulink and SCADE, which are widely used in the design and implementation of synchronous controllers.

There is a long debate over the pros and cons of synchronous vs. asynchronous concurrency. One argument that often comes up in the debate in favor of synchrony is determinism. The semantics of most¹ synchronous models are deterministic: given the same (sequence of inputs) the same (sequence of) outputs

*We would like to thank our co-authors, Norman Scaife from Verimag, and Marco di Natale from Scuola Superiore S. Anna. This work is supported by the Center for Hybrid and Embedded Software Systems (CHESS) at UC Berkeley, which receives support from the National Science Foundation (NSF awards #0720882 (CSR-EHS: PRET) and #0720841 (CSR-CPS)), the U.S. Army Research Office (ARO #W911NF-07-2-0019), the U.S. Air Force Office of Scientific Research (MURI #FA9550-06-0312), the Air Force Research Lab (AFRL), the State of California Micro Program, and the following companies: Agilent, Bosch, Lockheed-Martin, National Instruments, and Toyota.

¹ The synchronous language Signal allows non-determinism for modeling, but for code generation, programs must be deterministic. The compiler checks this.

will be produced. This has a number of benefits: systems are easier to understand and debug; they are also easier to verify: because interleavings are avoided, problems such as *state explosion* are less severe; etc.

Proponents of asynchrony often counter-argue that these benefits are lost during implementation. It is hard, if not impossible, to implement synchrony, they say, especially when it comes to distributed systems. Indeed, practice seems to confirm this argument. Most implementations of synchronous systems are either circuits (which can be viewed as centralized systems), or single-processor, single-task software implementations of synchronous control loops of the form: `while (true) do: await trigger; read inputs; compute; write outputs; update state; end while.`

2 The Implementation Problem

The main difficulty in implementing synchronous semantics on distributed systems is how to synchronize the distributed agents, which share no common clock. One solution is to use a clock-synchronization mechanism. Commercial components providing clock-synchronization services are available today, from systems such as TTA (see <http://www.tttech.com/>) and Flexray (see <http://www.flexray.com>), to more economic solutions such as implementations of the IEEE 1588 standard [1]. Such components can achieve clock synchronization sometimes as good as tens of nano-seconds, which is sufficient for a large class of applications. Still, despite their successes, these solutions are not universal today.

But clock synchronization is not the only way to implement synchronous semantics on asynchronous execution platforms. Examples of other solutions, not relying on clock synchronization, can be found in [4] and [14]. These works have two common themes. First, *correctness*, in the sense of *semantical preservation*: the methods proposed guarantee *by construction* that the implementation preserves the semantics of the original synchronous model. Second, *efficiency*: care is taken to develop methods that are guaranteed to be *optimal* with respect to some metric of interest (e.g., buffer sizes), or techniques that allow to compute such metrics (e.g., throughput or end-to-end latency). In what follows, we provide a summary of these works.

3 Implementation of synchronous models on single-processor, multitasking architectures with preemptive scheduling

This section gives a summary of the work described in [4]. The problem solved in that paper is how to implement a synchronous model on a single-processor computer that runs an operating system that allows *multitasking*, that is, where multiple processes (or *tasks*) share the computer.² Since there is only one CPU, *preemptive scheduling* is employed, where one task is interrupted (*preempted*) by another, and later resumes execution. Different scheduling policies can be used, such as *fixed-priority* or *earliest-deadline first* [3]. Because of preemptions, that can happen at arbitrary points in time, depending on the arrival and execution times of the tasks, this execution platform can be considered asynchronous. In fact, its semantics is very much that of concurrent threads with shared memory.

Multitasking creates problems, however, when tasks communicate. As shown in [4], straightforward communication schemes (e.g., shared memory protected by some semaphore or other mechanism, to ensure atomicity) do not generally preserve the semantics. This means that the *streams* (i.e., sequences of values exchanged between tasks) in the implementation are different from the streams defined by the synchronous model. For this reason, a new set of inter-task communication protocols are developed in [4], that are semantics preserving. These protocols result generally in higher memory usage than the straightforward scheme mentioned above. However, it is shown in [4] that this is necessary in order to preserve semantics. Indeed, these protocols are shown to be *memory-optimal*.

² A natural question is: “why use multitasking, when synchronous programs can be implemented using the single-task control-loop scheme?” The answer is: due to lack of computational resources. Synchronous programs are often *multirate*, meaning that different parts of the program execute at different frequencies. In these cases, executing all the parts within the smallest period (determined by the highest frequency) may not be feasible (the worst-case execution time may exceed the period). In this case, multitasking can be used to make the system schedulable. See [4] for details.

4 Implementation of synchronous models on distributed execution platforms

This section gives a summary of the work described in [14]. The problem solved in that paper is how to implement a synchronous model on a distributed execution platform called LTTA, consisting of a set of computers, each having its local clock, not synchronized with the other clocks. Computers communicate using point-to-point *communication by sampling* (CbS) channels, which can be conceptually viewed as shared buffers on which a producer writes (and overwrites) and a consumer reads.

To make the problem easier, an intermediate layer, called FFP, is introduced in [14]. FFP is a distributed and asynchronous execution platform, like LTTA. But nodes in FFP communicate via finite FIFO queues, instead of CbS channels. It is shown how FFP queues can be implemented on top of CbS channels, thus reducing the problem to implementing synchronous concurrency on top of FFP. It is shown that this is possible to do using principles similar to those used in Kahn Process Networks. The difference in FFP is that queues are finite, so care must be taken to avoid deadlocks: this can be done by having sufficiently large queues. Semantics is preserved in the sense that every stream (sequence of values) produced by the distributed implementation is equal to the corresponding stream defined by the synchronous model.

This takes care of correctness. However, issues of efficiency arise. In particular, because nodes in FFP communicate with FIFOs, and because clocks are asynchronous, a node need not do useful work at every tick of its clock: if some of its input queues are empty, or some of its output queues are full, the node will *skip* this tick, and wait for the next one. This motivates the study of *throughput*: how often are results produced in an FFP network? A notion of *real-time* throughput (RTT) is defined in [14]. RTT is hard to use, however, because it depends on the exact times when distributed clocks tick.³

To avoid this difficulty, in [14] introduces the notion of worst-case *logical-time* throughput (WCLTT). WCLTT is a unitless number in the interval $[0, 1]$, that only depends on network topology, queue sizes, and initial condition. Moreover, WCLTT is computable. Finally, WCLTT is related to RTT as follows: if all clocks in the system have a maximum inter-arrival time of Δ , then

$$\text{RTT} \geq \text{WCLTT} \cdot \frac{1}{\Delta}$$

WCLTT is an important metric characterizing how “well-balanced” the FFP network is, in particular, in terms of queue sizes. Increasing queue sizes cannot decrease WCLTT, in fact it may increase it. For throughput purposes, WCLTT should ideally equal 1. On the other hand, there is a trade-off between WCLTT and memory (queue sizes). The algorithm to compute WCLTT can serve as a basis for a design-space exploration process to explore this trade-off, by determining what the size of each queue should be.

Another metric studied in [14] is *latency*: the time it takes for a message to travel along a given path in the network, from a producer node to a consumer node. As with throughput, two notions of latency are introduced, real-time latency (RTL) and (worst-case) logical-time latency (WCLTL). WCLTL is computable and provides upper bounds to RTL.

5 Conclusions and Perspectives

As observed in [9], asynchronous concurrency models such as threads start with non-deterministic semantics, and then struggle to tame this non-determinism using various mechanisms (locks, semaphores, ...). An alternative is to start with a deterministic semantics, such as the one offered by synchronous models, and try to preserve it in a non-deterministic execution environment. Indeed, the latter can be distributed, asynchronous, and even thread-based! The research examples described above show that it is possible to do this both correctly and efficiently.

Clearly, this line of research is far from complete. Some open problems are discussed here, others can be found in the conclusions of [4] and [14].

³ It also depends on execution times and communication times, however, for simplicity, these are considered negligible in [14].

First, we only scratched the surface of the issue of semantical preservation. Exactly *which* semantics (or properties) of the model needs to be preserved by the implementation is largely dependent on the application at hand. Stream preservation is used in [4] and [14], but other notions may be more suitable in other contexts. In particular, for embedded system applications, the domain in which synchronous models perhaps excel, other properties of interest include reliability, power efficiency, stability, or even comfort. These notions lack sufficient formalization, and often require techniques that go beyond standard computer science.

A second question is, to what extent methods such as those presented above can be extended to other classes of execution platforms, or applications. Regarding execution platforms, of particular interest today are *multicores* [6]: one interesting problem is to study implementation of synchronous models on such platforms. Regarding applications, synchronous models have traditionally been strong in embedded control, with de-facto standards such as Simulink or SCADE, in the automotive and aerospace domains. Purely synchronous models are probably less suitable in more general *streaming applications* (e.g., video encoding/decoding), where the notion of dataflow needs to be generalized, as is done, for instance in models such as SDF [8] or StreamIt [13]. Recent work attempts to tackle this problem by translating dataflow diagrams to synchronous models [10] or by relaxing the synchrony requirements [5].

References

- [1] IEEE 1588 Standard for a Precision Clock Synchronization Protocol for Networked Measurement and Control Systems, 2008.
- [2] A. Benveniste, P. Caspi, S.A. Edwards, N. Halbwachs, P. Le Guernic, and R. de Simone. The synchronous languages 12 years later. *Proceedings of the IEEE*, 91(1):64–83, January 2003.
- [3] G. Buttazzo. *Hard Real-time computing systems*. Kluwer Academic Publishers, 1997.
- [4] P. Caspi, N. Scaife, C. Sofronis, and S. Tripakis. Semantics-Preserving Multitask Implementation of Synchronous Programs. *ACM Transactions on Embedded Computing Systems*, 7(2):1–40, February 2008.
- [5] A. Cohen, L. Mandel, F. Plateau, and M. Pouzet. Abstraction of Clocks in Synchronous Data-flow Systems. In *APLAS'08*, December 2008.
- [6] K. Asanovic et al. The Parallel Computing Laboratory at U.C. Berkeley: A Research Agenda Based on the Berkeley View. Technical Report UCB/EECS-2008-23, EECS Department, University of California, Berkeley, Mar 2008.
- [7] G. Kahn. The semantics of a simple language for parallel programming. In *Information Processing 74, Proceedings of IFIP Congress 74*. North-Holland, 1974.
- [8] E. Lee and D. Messerschmitt. Synchronous data flow. *Proceedings of the IEEE*, 75(9):1235–1245, 1987.
- [9] E.A. Lee. The problem with threads. *IEEE Computer*, 39(5):33–42, 2006.
- [10] R. Lubliner and S. Tripakis. Translating Data Flow to Synchronous Block Diagrams. In *6th IEEE Workshop on Embedded Systems for Real-time Multimedia (ESTIMedia'08)*, 2008.
- [11] R. Milner. *Communication and Concurrency*. Prentice Hall, 1989.
- [12] J. Ousterhout. Why threads are a bad idea (for most purposes). Invited Talk at the 1996 USENIX Technical Conference. Available online.
- [13] W. Thies, M. Karczmarek, and S. Amarasinghe. StreamIt: A Language for Streaming Applications. In *Intl. Conf. on Compiler Construction*, 2002.
- [14] S. Tripakis, C. Pinello, A. Benveniste, A. Sangiovanni-Vincentelli, P. Caspi, and M. Di Natale. Implementing Synchronous Models on Loosely Time-Triggered Architectures. *IEEE Transactions on Computers*, 57(10):1300–1314, October 2008.