The Specification and Execution of Heterogeneous Synchronous Reactive Systems

A Proposal for Ph.D Research

Stephen A. Edwards

December 5, 1995

Abstract

Electronic systems are becoming more complex. Using subproblem-specific languages simplifies their design, but presents the problem of connecting the parts. I propose a system description scheme for reactive systems (systems that maintain a dialog with their environment) that supports such heterogeneity.

I expect to contribute the system description scheme, a mathematical framework for it, a set of efficient algorithms for simulating these systems, and a practical implementation of the scheme. My prototype compiler suggests this scheme can be made practical, and the mathematical framework is nearly complete.

I expect this work to make designing complex, heterogeneous reactive systems fast and simple.

1 Introduction

Electronic systems are growing more complex. Describing these with a diverse set of languages, each suited to a particular subtask, can greatly simplify designing these systems, but the problem of connecting the subtasks arises. For example, a convenient description of the digital answering machine depicted in Figure 1 might use a digital signal processing language for the dialtone and DTMF detectors, a traditional sequential programming language for the memory, and a state-machine centered language for the controllers. The research proposed here addresses describing and simulating a large class of such heterogeneous systems.

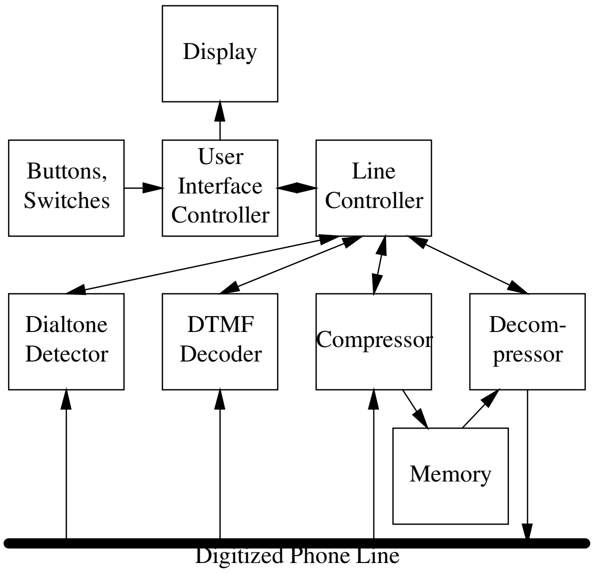

Figure 1: A digital answering machine—a typical heterogeneous system

I propose a specification scheme that supports heterogeneity through abstraction—it ignores the details of large, complex parts of a system. The level of abstraction I propose is enough to greatly simplify certain analyses, but not so high as to preclude all.

The inter-task communication scheme I adopt was developed by others for specifying *reactive* systems[1]—systems that maintain an ongoing dialog with their envi-

[1] Harel and Pnueli [14] coined the term.

ronment. In these systems, when things happen is as important as what happens. Many important systems fit this mold, including those with complex user interfaces (e.g., digital watches, CD players, and most consumer electronics) and those doing real-time control (e.g., anti-lock braking systems, industrial process controllers).

The contributions of this research will be a specification scheme for heterogeneous reactive systems, a mathematical framework for it, a set of scheduling algorithms for quickly simulating systems described in it, and a practical implementation of the scheme in the existing multi-language environment Ptolemy [6].

In the following sections, I discuss my specification scheme (Section 2), its mathematical framework (Section 3), and issues in its implementation (Section 4). Section 5 concludes with a description of the current state of the research and proposed future work.

2 The Specification Scheme

In this section, I describe the approach I take to heterogeneity and show how it fits into my specification scheme. My specification scheme is new, although it is similar to many existing ones. The heterogeneous approach presented here is not new, but its application to reactive systems is.

2.1 Heterogeneity

To support heterogeneity, I adopt the technique in Figure 2. This is a "black box" scheme where the module interface style is dictated, but the contents of these modules may be anything. The inter-module communication style is also imposed. The objective of this technique is to maintain a high level of abstraction while allowing a reasonable amount of analysis. Chang, Kalavade, and Lee discuss this approach at greater length [9].

This approach is especially useful for describing large systems. For many whole-system analyses, complex module contents may be ignored, greatly simplifying the problem. Moreover, new module specification schemes can easily be added to such a framework.

Such a heterogeneous approach is employed in many large system development environments. One

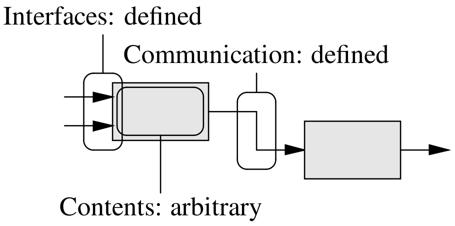

Figure 2: A heterogeneous specification scheme.

well-known example is the UNIX programming environment [16]. Here, a system under development is broken into independently-compiled source files, which are connected later using little knowledge of their contents. Kahn's network of communicating concurrent processes [15] are deterministic because of a simple constraint on process interfaces. The Ptolemy system [6] also takes a heterogeneous approach to connect systems specified using different computational paradigms.

2.2 The Scheme

My proposed specification scheme is based on a network of communicating modules, such as those shown in Figure 3. The module interfaces are restricted to compute monotonic functions on finite complete partial orders (see Section 3.2). Modules communicate through single-driver wires: each wire takes exactly one value in an instant—there is no buffering, no production or consumption of data, and no possibility of deadlock.

That each wire must be driven by exactly one module is not a significant restriction. The effect of driving a wire from multiple sources can be achieved by adding a module that combines the output of each source.

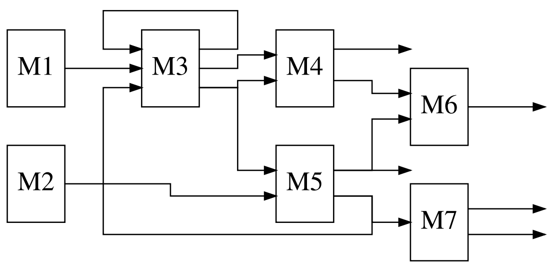

Figure 3: An example of my specification scheme: A network of communicating modules

Technically, the input and output domains of the module functions are all the wires in the circuit. However, a module only examines its input wires and only modifies its output wires, as shown in Figure 4.

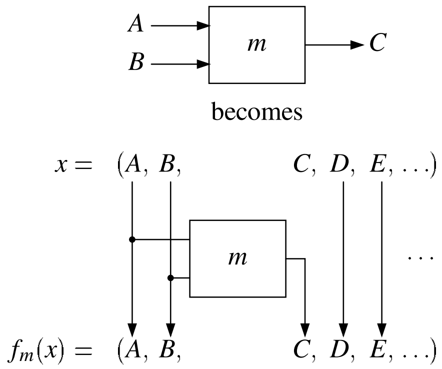

Figure 4: Module functions are extended to consume and produce the vector of all wires values

A system's behavior in an instant is defined as the least fixed point of the system function f, a composition of the module functions f_1, f_2, ..., f_n. These functions map a vector of wire values (inputs) to another vector of wire values (outputs). This definition captures the intuitive notion of a system being stable when its outputs have the values imposed by its inputs. In other words, if x_t is a vector of wire values at time t, then the system is stable when

$$f(x_t) = x_t.$$

I define behavior as the least fixed point because it is defined and unique under a simple constraint on f (Theorem 1), order-independent (Corollary 1), and makes the fewest assumptions about the system.

3 The Mathematical Framework

In this section, I present a mathematical framework for my specification scheme. I adopt synchronous time semantics (taken from a number of existing languages) and describe my systems using complete partial orders and monotonic functions (taken from the theory of program semantics).

3.1 Synchronous Semantics

Synchronous semantics, proposed by Berry, Benveniste, Pneuli, Halbwachs, and others[2], address the problem of reactive system specification. It employs the *strong synchrony hypothesis*: computation takes zero time. Such an assumption leads to the model of time shown in Figure 5, where real continuous time is divided into a series of discrete instants. In practice, a system that processes all its inputs before being presented with more appears synchronous. The reason to build synchronous systems is similar to the reason to build digital systems: a discrete computational domain makes it possible to engineer "perfect" systems in the presence of noise. Digital systems avoid noise from uncertain voltages; synchronous systems avoid noise from uncertain delays.

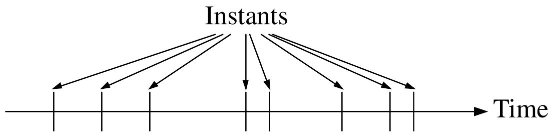

Figure 5: The discrete model of time in synchronous semantics

Instantaneous computation makes contradictory specifications possible, and any system that has zero delay must deal with them. Such a contradictory specification is shown in Figure 6. The problem arises from zero delay coupled with a communication cycle. My scheme deals with such contradictions by giving them a well-defined meaning. Other schemes attempt to analyze and detect such contradictions before the system is simulated, a costly technique.

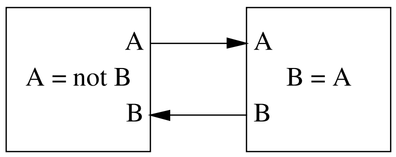

Figure 6: A contradictory zero-delay specification.

[2] Benveniste and Berry provide an excellent overview of this approach in a special issue of Proceedings of the IEEE [2].

3.2 CPOs and Monotonic Functions

To formally describe the semantics of my specifications, I use the concepts of complete partial orders and monotonic functions from the theory of program semantics[3]. Finite complete partial orders describe wire values, monotonic functions describe the modules, and the least fixed point of the composition of these functions assigns meaning to a system in an instant.

Definition 1 *A* finite complete partial order *(CPO) is a 3-tuple* $(S, \sqsubseteq, \perp)$ *where*

- *S is a finite set*
- *$\sqsubseteq$ is a binary relation that is*
  - *Transitive: $x \sqsubseteq y$ and $y \sqsubseteq z$ implies $x \sqsubseteq z$*
  - *Antisymmetric: $x \sqsubseteq y$ and $y \sqsubseteq x$ implies $x = y$*
  - *Reflexive: $x \sqsubseteq x$*
- *$\perp \in S$ is a distiguished element such that $\perp \sqsubseteq x$ for all x*

Figure 7 shows a pair of CPOs.

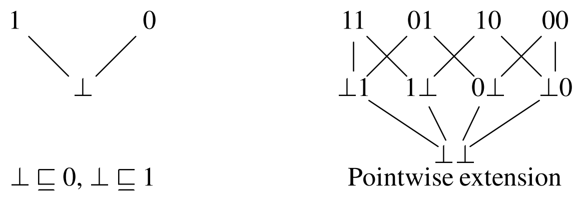

Figure 7: A simple CPO and its two-vector extension

Definition 2 *A* monotonic function *$f : S \to S$ on a finite complete partial order $(S, \sqsubseteq, \perp)$ satisfies*

$$x \sqsubseteq y \text{ implies } f(x) \sqsubseteq f(y)$$

for all $x, y \in S$.

A visualization of the effect of a monotonic function is shown in Figure 8.

[3]Scott and Strachey [19] pioneered this field. Modern-day books on the subject include Allison [1], Gunter [11], and Winskel [22]

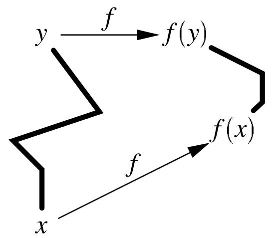

Figure 8: The effect of a monotonic function. If there is a path from x to y, then there is a similar path from $f(x)$ to $f(y)$.

3.3 The Least Fixed Point Theorem

The following well-known theorem is key to my scheme. It ensures the specifications are deterministic (i.e., there is exactly one behavior associated with a specification) and always have meaning. Moreover, the proof suggests how to compute the behavior efficiently.

Theorem 1 *A monotonic function f on a finite complete partial order $(S, \sqsubseteq, \perp)$ has a unique least fixed point, given by the limit of the ascending sequence $\perp \sqsubseteq f(\perp) \sqsubseteq f(f(\perp)) \sqsubseteq \cdots$.*

Proof. $\perp \sqsubseteq f(\perp)$, by definition of $\perp$. Since f is monotone, it follows that $f(\perp) \sqsubseteq f(f(\perp))$, $f(f(\perp)) \sqsubseteq f(f(f(\perp)))$, etc. Because S is finite, there must be some $u \in S$ that appears twice, i.e., such that $f^i(\perp) = f^j(\perp) = u$ for some $i < j$. Because this is an ascending chain, $f^i(\perp) \sqsubseteq f^{i+1}(\perp) \sqsubseteq \cdots \sqsubseteq f^j(\perp)$. However, since $f^i(\perp) = f^j(\perp) = u$, $f^{i+1}(\perp) = \cdots = f^{j-1}(\perp) = u$. Since $u = f^i(\perp)$, and $f^{i+1}(\perp) = f(f^i(\perp)) = u$, $f(u) = u$, so u is a fixed point.

This is a least fixed point. Let v be a fixed point, i.e., $f(v) = v$. It follows that $\perp \sqsubseteq v$, $f(\perp) \sqsubseteq f(v) = v$, ..., $u = f^i(\perp) \sqsubseteq v$. ■

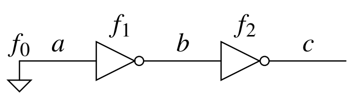

Figure 9: A simple system

To illustrate the behavior of a system defined using the least fixed point, consider the simple system in Figure 9. To compute its behavior in an instant, the fixed point of the function $f = f_2 \circ f_1 \circ f_0$ is found though iteration:

$$
\begin{aligned}
(a,b,c) &= (\perp,\perp,\perp) \\
f_0(\perp,\perp,\perp) &= (0,\perp,\perp) \\
f_1(0,\perp,\perp) &= (0,1,\perp) \\
f_2(0,1,\perp) &= (0,1,0) \\
f_2(f_1(f_0(0,1,0))) &= (0,1,0)
\end{aligned}
$$

3.4 The Order-Invariance Theorem

The following[4] allows the composition order of the module functions to be rearranged without affecting the least fixed point. Not only does this ensure the least fixed point is a canonical definition of the behavior of the system, but it allows the order to be arranged for the optimum convergence rate, for example.

Definition 3 *A set of functions $f_1, \ldots, f_n$ has the* compositional fixed point property *if for all permutations $a_1, \ldots, a_n$ of $1, \ldots, n$, $f_{a_1} \circ \cdots \circ f_{a_n}(x) = x$ if and only if $f_i(x) = x$ for all $i = 1, \ldots, n$.*

Theorem 2 *The module functions have the compositional fixed point property.*

Proof. Clearly, if $f_i(x) = x$ for all i, then any composition f of the f_i has $f(x) = x$, regardless of the functions.

Assume there is some fixed point x of a composition $f = f_{a_1} \circ \cdots \circ f_{a_n}$ such that $f(x) = x$, but $f_i(x) \neq x$ for some i. This implies there is some element of the vector x that is modified by f_i and f_j for some $j \neq i$. However, by construction, each element is modified by exactly one function (this is the "one wire," "one driver" rule), so this is a contradiction, and no such fixed point may exist. ■

Theorem 3 *The set of fixed points for any compositional permutation of functions with the compositional fixed point property is the same and is given by $\{x \mid \forall i. f_i(x) = x\}$.*

Proof. Clearly, members of the set $\{x \mid \forall i. f_i(x) = x\}$ are fixed points of all permuted compositions of functions. Now, let f and g be different compositional permutations of the elements of $\{f_i\}$ and assume $f(x) = x$ and $g(x) \neq x$. Since the elements of $\{f_i\}$ have the compositional fixed point property, $f(x) = x$ implies $f_i(x) = x$ for all i. However, this implies $g(x) = x$, since g is another permutation. It follows that the fixed points are the same for all permutations and are exactly $\{x \mid \forall i. f_i(x) = x\}$. ■

[4] Praveen Murthy and I proved this.

Corollary 1 *There is exactly one least fixed point for all permutations of the module functions.*

Proof. Follows from Theorems 1, 2, and 3. ■

3.5 Importing Foreign Functions

An objective of my scheme is to allow functions specified in any domain to be imported easily. One solution, presented below, is to make the foreign functions *strict*—require all inputs to be known before any output is produced. As many languages only describe strict functions, this is not a significant restriction.

Modules in my scheme must communicate through data embedded in a complete partial order and must be monotonic. For a foreign function whose arguments and results are taken from a finite (discrete) set, the first requirement is easily satisfied by building a *flat* domain, such as that depicted in Figure 10.

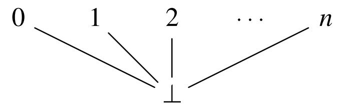

Figure 10: A flat CPO built from the set $\{1, \ldots, n\}$

The second requirement, monotonicity, is satified by making the module function strict. That is, if any of the function's inputs is $\perp$, then its output is $\perp$, i.e.,

$$f(\ldots,\perp,\ldots) = \perp.$$

Under this scheme, acyclic networks built exclusively from strict functions behave as expected. Cyclic networks with nothing but strict functions usually give $\perp$s on wires in a cycle, but adding one non-strict function per loop cures this problem.

This is not the only technique that could be used to import foreign functions. Non-strict functions could also be imported, but ensuring the monotonicity condition becomes more difficult.

4 Implementation

Any implementation of a language based on the synchronous semantics must evaluate the least fixed point of the system in every time step. Two other schemes have been devised to do this, but neither support heterogeneity.

A diverse collection of languages have evolved around synchronous semantics. Esterel [4] is a textual imperative language with concurrent and sequential statements. Lustre [12] is a textual declarative language with dataflow-style semantics. Signal [17] is a textual relational language, also with a dataflow flavor. Argos, a derivative of Statecharts [13], is a graphical hierarchical finite state machine language.

Compilers for these languages have been developed around common intermediate formats [21]. Esterel and Argos are translated into an imperative format "IC"; Signal and Lustre are translated into a declarative format "GC." Both can be converted into a sequential automaton format "OC," from which sequential code can be generated. The OC format describes a monolithic state machine, which can be exponentially larger than, say, the Esterel program it represents.

In effect, the IC-OC path performs an exhaustive simulation of the program and records these results in a table. At runtime, the correct behavior of the program (the least fixed point) is simply recalled from the table. This makes for very fast executables (virtually everything is compiled away), but does so at the expense of exponentially long compilation times and exponentially large executables. Using this scheme, a small (600-line) program with 32 states can take fifteen minutes to compile and produce a ten-megabyte executable [10].

The latest Esterel compiler from Berry's group translates an Esterel program into a cyclic boolean network [3]. This is transformed into an acyclic boolean network by an implicit version [20] of Malik's procedure for analyzing cyclic combinational circuits [18]. This analysis is probably NP-complete, but in practice is fast enough for programs of reasonable size.

The proof of Theorem 1 suggests the execution scheme I propose. A convergent iteration, it starts with $\perp$ on all the wires and repeatedly applies f, the composition of all the module functions, until a fixed point is reached. Theorem 1 ensures this will always converge, and do so in no more steps than there are wires (for flat domains).

Execution Scheme	Heterogeneous	Compilation Time	Executable Size	Execution Speed
Tabular FSM	no	exp.	exp.	const.
Boolean Network	no	exp.	poly.	poly.
Convergent Iteration	yes	poly.	poly.	poly.

Table 1: A comparison of synchronous language compilers. I propose the Convergent Iteration scheme.

This scheme, which I have implemented in a prototype compiler for the Esterel language [10], is compared to other compilers in Table 1. My shorter compilation times is mostly due to not doing causality checking. For example, the other compilers would flag the contradictory specification in Figure 6 as non-causal, whereas mine would compile and run it, giving a run-time error. I believe such expensive checking, while important to ensure correct designs, should be separated from compilation along with other difficult verification problems.

Corollary 1 ensures that any permutation of the module functions will produce the same result, so this order can be optimized to minimize the amount of work required to find the least fixed point. For example, finding the fixed point of the system in Figure 9 using the composition $f_0 \circ f_1 \circ f_2$ would take three times as many function evaluations as the composition $f_2 \circ f_1 \circ f_0$.

Finding a rapidly-converging ordering is a scheduling problem, and variants of it have appeared in many places. For acyclic systems, the solution is obvious—a topological ordering. For cyclic systems, the best ordering is less obvious.

In my proposed implementation, there are at least two approaches to system execution. One approach, fully static scheduling, decides on an evaluation order that will be used for each instant at compile-time. Another approach, fully dynamic scheduling, decides what function to evaluate next during execution. The advantage here is that data-dependent decisions can be made, often greatly accelerating things. However, the higher overhead may cancel out such gains.

Applicable existing scheduling techniques are numerous. The clustering scheme used in Buck's boolean dataflow domain [7] might find application here. Shiple et al. adopt Bourdoncle's Weak Topological Ordering scheme [5] to accelerate a similar problem. Similar problems can be found throughout engineering. For example, Buhl et al.'s SPARK system for solving nonlinear differential algebraic systems [8] addresses exactly this problem. A optimal solution may be expensive, however, since the minimum feedback arc set problem is NP-complete for planar graphs.

5 Current and Future Work

As a proof of concept, I have written a compiler for the Esterel synchronous language that uses my proposed convergent iteration technique [10]. Although preliminary, it suggests the convergent iteration technique is practical even for large systems: the system typically converged in two or three iterations.

The mathematical framework presented in Section 3 is solid, but will be extended and polished. It forms a solid foundation for future work, and the two main theorems allow significant optimizations without fear of non-convergence, non-deterministic behavior, etc.

I intend to implement this scheme in the Ptolemy multi-language simulation/prototyping environment [6]. This framework is ideally suited for experimenting with my system description scheme, since it has support for modules, interconnections, schedulers, and already has many well-developed language domains designed to be interfaced with others.

Work on scheduling algorithms will follow. I intend to explore first an optimal scheme, probably based on minimum feedback arc set size. I expect this will be impractical for large systems, so I also intend to explore heuristic approaches. I also plan to explore the tradeoffs between static and dynamic scheduling.

In conclusion, I expect to contribute a specification scheme that supports hetereogeneously-specified synchronous reactive systems, a mathematical framework for it, a set of scheduling algorithms for the scheme, and a practical implementation of it within a multi-language environment.

References

[1] Lloyd Allison. *A Practical Introduction to Denotational Semantics*. Cambridge University Press, 1986.

[2] Albert Benveniste and Gérard Berry. The synchronous approach to reactive real-time systems. *Proceedings of the IEEE*, 79(9):1270–1282, September 1991.

[3] G. Berry. A hardware implementation of pure Esterel. In *Proceedings of the International Workshop on Formal Methods in VLSI Design*. ACM SIG DA, January 1991.

[4] Gérard Berry and Georges Gonthier. The Esterel synchronous programming language: Design, semantics, implementation. *Science of Computer Programming*, 19(2):87–152, 1992.

[5] François Bourdoncle. Efficient chaotic iteration strategies with widenings. In *Formal Methods in Programming and Their Applications: Internation Conference Proceedings*, volume 735 of *Lecture Notes in Computer Science*, Novosibirsk, Russia, June 1993. Springer-Verlag.

[6] J. Buck, S. Ha, E. A. Lee, and D. G. Messerschmitt. Ptolemy: A mixed-paradigm simulation/prototyping platform in C++. In *Proceedings of the C++ At Work Conference*, Santa Clara, CA, November 1991.

[7] Joseph Tobin Buck. *Scheduling Dynamic Dataflow Graphs with Bounded Memory using the Token Flow Model*. PhD thesis, University of California, Berkeley, 1993. Available as UCB/ERL M93/69.

[8] W. F. Buhl, A. E. Erdem, F. C. Winkelmann, and E. F. Sowell. Recent improvements in SPARK: Strong component decomposition, multivalued objects, and graphical interface. Technical Report LBL-33906, Lawrence Berkeley Laboratory, August 1993.

[9] W.-T. Chang, A. Klavade, and E. A. Lee. Effective heterogeneous design and cosimulation. In *NATO Advanced Study Institute Workshop on Hardware/Software Codesign*, June 1995.

[10] Stephen Edwards. An Esterel compiler for a synchronous/reactive development system. Technical Report UCB/ERL M94/43, University of California, Berkeley, June 1994.

[11] Carl A. Gunter. *Semantics of Programming Languages*. MIT Press, 1992.

[12] Nicholas Halbwachs, Paul Caspi, Pascal Raymond, and Daniel Pilaud. The synchronous data flow programming language LUSTRE. *Proceedings of the IEEE*, 79(9):1305–1320, September 1991.

[13] D. Harel. Statecharts: A visual formalism for complex systems. *Science of Computer Programming*, 8(3):231–274, June 1987.

[14] D. Harel and A. Pnueli. *On the Development of Reactive Systems*, volume 13 of *NATO ASI Series. Series F, Computer and Systems Sciences*, pages 477–498. Springer-Verlag, 1985.

[15] Gilles Kahn. The semantics of a simple language for parallel programming. In *Information Processing 74: Proceedings of IFIP Congress 74*, pages 471–475, Stockholm, Sweden, August 1974. North-Holland.

[16] Brian W. Kernighan and Rob Pike. *The UNIX Programming Environment*. Prentice-Hall, 1984.

[17] Paul Le Guernic, Thierry Gautier, Michel Le Borgne, and Claude Le Maire. Programming real-time applications with SIGNAL. *Proceedings of the IEEE*, 79(9):1321–1336, September 1991.

[18] Sharad Malik. Analysis of cyclic combinational circuits. In *1993 IEEE/ACM International Conference on Computer-Aided Design: Digest of Technical Papers*, pages 618–625, Santa Clara, CA, November 1993. IEEE Computer Society Press.

[19] Dana Scott and Christopher Strachey. Toward a mathematical semantics for computer languages. In *Proceedings of the Symposium on Computers and Automata*, pages 19–46. Polytechnic Institute of Brooklyn, April 1971.

[20] Thomas R. Shiple, Gérard Berry, and Hervé Touati. Constructive analysis of cyclic circuits. In *Proceedings of the European Design and Test Conference*, March 1996.

[21] The C2A Group. Projet synchrone: Les formats communs des languages synchrones (common formats for the synchronous languages). Technical Report 157, INRIA, June 1993. Translated from the French by Wendell Baker.

[22] G. Winskel. *The Formal Semantics of Programming Languages: An Introduction*. Foundations of Computing. MIT Press, 1993.