An Implementation of Constructive Synchronous Programs in POLIS

Gérard Berry*
Ecole des Mines de Paris and INRIA
2004 Route des Lucioles
06904 Sophia-Antipolis CEDEX
FRANCE

Ellen M. Sentovich
Cadence Berkeley Laboratories
2001 Addison Street, 3 floor
Berkeley, CA 94704-1103
USA

November 2, 1998

Abstract

Design tools for embedded reactive systems commonly use a model of computation that employs both synchronous and asynchronous communication styles. We form a junction between these two with an implementation of synchronous languages and circuits (Esterel) on asynchronous networks (POLIS). We implement fact propagation, the key concept of synchronous constructive semantics, on an asynchronous non-deterministic network: POLIS nodes (CFSMs) save state locally to deduce facts, and the network globally propagates facts between them. The result is a correct implementation of the synchronous input/output behavior of the program. Our model is compositional, and thus permits implementations at various levels of granularity from one CFSM per circuit gate to one CFSM per circuit. This allows one to explore various tradeoffs between synchronous and asynchronous implementations.

1 Introduction

Our purpose is to reduce the gap between two distinct models of concurrency that are fundamental in the embedded systems framework, the synchronous and asynchronous models, with application to systems written in the Esterel synchronous programming language and implemented in the POLIS system developed at UC Berkeley and Cadence.

The synchronous or zero-delay model is used in circuit design and in synchronous programming languages such as Esterel [6], Lustre [12], Signal [10], and SyncCharts [2] (a synchronous version of Statecharts [13]), see [11] for a global overview. In this model, all bookkeeping actions such as control transmission and signal broadcasting are conceptually performed in zero-time, only explicit delays taking time. Thus, a conceptual global clock controls precisely when statements simultaneously compute and exchange messages. The model makes it possible to base design on deterministic concurrency, which is much easier to deal with than classical non-deterministic concurrency. Compiling, optimizing, and verifying programs is done using powerful Boolean computation techniques, see [5].

The synchronous model is well-suited for direct specification and implementation of comparatively compact programs such as protocols, controllers, human-machine interface drivers, and glue logic. In this case, one can build a global clock slow enough to react to each possible environmental input.

In an asynchronous model, processes exchange information through messages with non-zero travel time. Asynchronous models are well-suited for network-based distributed systems specification and for hardware/software codesign, where the relative speed of components may vary

*This work was begun while the first author was visiting Cadence Berkeley Laboratories, August 1998.

widely. There are many asynchronous formalisms with varied communication policies. For example, CSP processes [14] communicate by rendezvous, while data-flow processes [15] exchange data through queues or buffers.

The POLIS [3] mixed synchronous/asynchronous model has been developed at UC Berkeley and Cadence, with primary focus on codesign. It is a Globally Asynchronous Locally Synchronous (GALS) model, in which synchronous nodes called CFSMs (Codesign Finite State Machines) are arranged in an asynchronous network and communicate using non-blocking 1-place buffers, and through a synthesized real-time operating system (RTOS) for the software part. The CFSMs can be programmed in a concurrent synchronous language such as Esterel, thus taking maximal advantage of the synchronous model at the node level. The model can be efficiently simulated and implemented in hardware and/or software; notice that 1-place buffers are much simpler to implement than FIFOs, especially at the hardware/software boundaries. However, POLIS networks have much less intrinsic semantic safety than FIFO-based dataflow Kahn networks [15], which are behaviorally deterministic, and their behavior must be carefully controlled. In particular, buffer overwriting in POLIS can lead to non-deterministic behaviors that can be hard to analyze and prove correct.

Here, we show that the behavior of a synchronous circuit or program can be nicely implemented in a POLIS network. Of course, one can implement a synchronous program in a single CFSM node in a straightforward way. Here, we are interested in *distributed implementations* where the synchronous behavior is split between asynchronously communicating units, without a global clock. In practice, this is useful when the application behavior is naturally synchronous but the execution architecture is distributed and possibly heterogeneous, with physical inputs and outputs linked to different computing units. We retain the synchronous philosophy when specifying an application and we benefit from the flexibility and efficiency of CFSM networks in the implementation. We propose a solution in which the CFSM granularity can be chosen at will: any part of the synchronous program can be implemented in a single synchronous CFSM, which makes it possible to partition the program according to the architecture constraints and the best synchrony/asynchrony compromise.

Other authors have proposed such distributed implementations of synchronous programs on asynchronous networks, see for example [9, 8], and we draw much from their work. However, our implementation takes maximal advantage of the semantics of the objects we deal with and it is presented differently, with a trivial correctness proof. Technically speaking, we present a POLIS implementation of *constructive synchronous circuits* [5, 18], which is a class of well-behaved cyclic circuits that generalizes the usual class of acyclic circuits. Since Esterel programs are translated into constructive circuits [4], this implementation handles Esterel as well.

The key of any implementation of synchronous programs is the realization of a conceptual zero-delay reaction to an input assignment. In a distributed asynchronous network, this must be done by a series of message exchanges. In our implementation, the messages are CFSM-events that carry proven facts about synchronous circuit wire or expression values. Such facts are exactly the logical information quanta on which the constructive semantics are based. The CFSM nodes generate output facts from input facts according to the semantic deduction rules. This is done over a *series of computations* since conceptually simultaneous facts now arrive at different times.

For a single reaction of a program, the number of events is uniformly bounded. No buffer overwrite can occur in the network. Although the internal behavior is non-deterministic, the overall behavior respects the synchronous semantics of the original program and thus is deterministic. This is true independently of the schedule employed by the RTOS. In addition, execution of successive synchronous reactions can be pipelined.

Finally, the implementation takes full advantage of the mathematical properties of the constructive semantics. In particular, the *compositionality* property makes it possible to arbitrarily group elementary circuit gates into CFSM nodes: this allows any level of granularity, from one single CFSM for the program at one extreme to one CFSM per individual gate at the other.

Clearly, there are many applications for which using only the synchronous formalism at specification level makes no sense, in which case our results are not directly applicable. Nevertheless, we think that they show that the apparent distance between synchrony and (controlled) asyn-

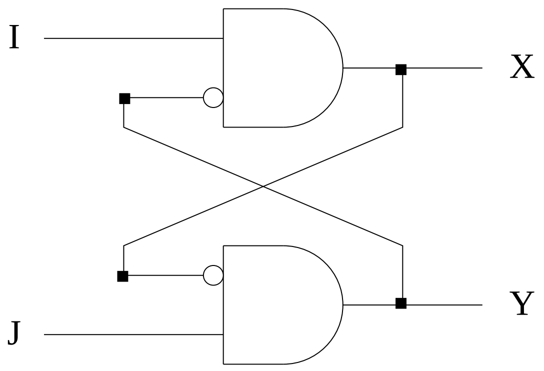

Figure 1: Circuit C_1

chrony can be reduced, and we hope that the technology we present can serve as a basis for future mixed-mode language developments.

We start in Section 2, by presenting the logical, semantical, and electrical views of constructive circuits. In Section 3, we briefly present the POLIS CFSM network model of computation. Our implementation of constructive circuits in this model is presented in Section 4, We discuss possible applications and synchrony/asynchrony tradeoffs in Section 5, and we conclude in Section 6.

2 Constructive Circuits

Constructive circuits are "well-behaved" possibly cyclic circuits that generalize the class of acyclic circuits. Acyclic circuits can be viewed in two different ways:

- as Boolean equation systems, then defining a Boolean function that associates an output value assignment with each input value assignment.
- as electrical devices made of wires and gates that propagate voltages and have certain delays: if the inputs are kept electrically stable long enough to one of two binary voltages (say 0V and 3V), the outputs stabilize to one of the binary voltages.

Relating the Boolean and electrical approaches is easy for acyclic circuits: when the outputs electrically stabilize, they take the voltages corresponding to the results of the Boolean input/output function. Constructive circuits have exactly the same characteristics even in the presence of cycles.

2.1 The Behavior of Cyclic Circuits

A circuit has input, output, and internal *wires*; the latter we also call *local variables*. In our examples, we use the letters I, J for the inputs and X, Y for the outputs and locals, making it precise which are the outputs where necessary. Each output or local variable is defined by an equation $X = E$, where E is an expression built using variables and the operators $\neg$ (negation), $\wedge$ (conjunction), an $\vee$ (disjunction). For simplicity, we assume that an expression E is either a variable, the negation of a variable, or a single n-ary operator $\wedge$ or $\vee$ applied to variables or the negation of variables. Any circuit can be put into this form by adding enough auxiliary variables.

A circuit can also be considered as a network of gates, as pictured in Figure 1. Each wire has a single source and multiple targets. The gates correspond to the operators.

As a running example, we shall consider the following circuit C_1, with outputs X and Y:

$$C_1 \left\{ \begin{array}{l} X = I \wedge \neg Y \\ Y = J \wedge \neg X \end{array} \right.$$

Notice that C_1 is cyclic: Y appears in the equation of X and conversely.

2.1.1 Circuits as Boolean Equations

In the Boolean view, we try to solve the circuit equations using Boolean values 0 and 1. An *input assignment* i associates 0 or 1 with some input variables. An input assignment is *complete* if it associates a value with any input variable. For a complete assignment i, A *Boolean solution* of the circuit is an assignment of values 0 or 1 to the other variables that satisfies the equations.

An acyclic circuit has exactly one Boolean solution for each complete input assignment. A cyclic circuit may have zero, one or several solutions for a given complete input assignment. For example, consider the case where there is no input and one output X. For $X = \neg X$, there is no solution. For $X = 1 \vee X$, there is a unique solution $X = 1$. For $X = X$, there are two solutions $X = 0$ and $X = 1$.

For C_1, there is a unique solution if $I = 0$ or $J = 0$. The solution is $X = 0$ and $Y = J$ if $I = 0$, or $Y = 0$ and $X = I$ if $J = 0$. If $I = J = 1$, the equations reduce to $X = \neg Y$, $Y = \neg X$, and there are two solutions, $X = 0$, $Y = 1$ and $X = 1$, $Y = 0$.

2.1.2 Circuits as Electrical Devices

In the electrical view, one preferably uses the graphical presentation and vocabulary. Wires associated with variables carry two different voltages, also called 0 and 1 for simplicity, and logic gates implement the Boolean operators. Wires and gates can have propagation delays. We shall not be very accurate here about delays; technically, the delay model we refer to is the up-bounded inertial delay model described in [7, 17]. A complete input assignment is realized by keeping the input wires stable over time at the appropriate voltages. Voltages propagate in the circuit wires according to the laws of electricity, and the property we are interested in is wire voltage stabilization after a bounded time. The non-input wires are assumed to be initially unstable.

Outputs of acyclic circuits always stabilize. Outputs of cyclic circuits may or may not stabilize. For example, the output of $X = 1 \vee X$ stabilizes, while that of $X = \neg X$ oscillates between 0 and 1. The output of $X = X$ remains unstable. When wires stabilize, their values always satisfy the equations.

Stabilization may depend on delays. For example, in the *Hamlet* circuit[1] defined by $X = X \vee \neg X$, the output X stabilizes to 1 for some delays and does not stabilize for others, see [5]. Stabilization may also depend on the input assignment: for C_1, outputs stabilize to the right Boolean values unless $I = J = 1$, in which the behavior is delay-dependent, with no stabilization for some delays.

2.2 Constructive Boolean Logic

Notice that the perfect match between Boolean and electrical solution is lost for cyclic circuits: for *Hamlet*, $X = X \vee \neg X$, the Boolean output function is well-defined and yields $X = 1$, while electrical stabilization may not occur. *Hamlet* has a unique Boolean solution because 1 happens to be a solution while 0 is not. Finding the solution involves propagating *non-causal* information and this cannot be done by non-soothsaying electrons in wires. Fortunately, Boolean logic can be weakened into *constructive* Boolean logic, in which the $X = 1$ solution to *Hamlet* is rejected, thereby rendering the Boolean and electrical results the same: no solution exists. Constructive Boolean Logic precisely models electrical behavior.

2.2.1 Facts and Proofs

Constructive Boolean logic deals with *facts* and *proofs*. A *fact* has the form $E = 0$ or $E = 1$ where E is a Boolean expression. An *input fact* is $I = 0$ or $I = 1$ for an input variable I. An input assignment i is a set of input facts. Facts are deduced from other facts by *deduction rules*. There are deduction rules for each type of gate and one rule to handle equations. Here are the rules for the $\wedge$ conjunction operator:

[1] Think of X as *to be*.

$$\frac{E=0}{E\wedge F=0}\quad (\textit{l-and})$$

$$\frac{F=0}{E\wedge F=0}\quad (\textit{r-and})$$

$$\frac{E=1\quad F=1}{E\wedge F=1}\quad (\textit{b-and})$$

The facts above the horizontal bar are the *premises* and the fact below the bar is the *conclusion.* Rule (b-and) reads as follows: from the facts $E=1$ and $F=1$, deduce $E\wedge F=1$. The rules for $\vee$ (*or*-gate) are dual. The rules for negation are:

$$\frac{E=0}{E=1}\quad (\textit{not-0})$$

$$\frac{E=1}{E=0}\quad (\textit{not-1})$$

Notice that $X\vee Y$ behaves as $\neg(\neg X\wedge\neg Y)$, just as in classical Boolean logic. The rules for a circuit equation $X=E$ are

$$\frac{E=b}{X=b}\quad (X=E\ :\ b)$$

where b can be either 0 or 1.

A *proof* is a sequence of facts that starts by the facts of an input assignment and such that any other fact can be deduced from the previous facts using a rule. The following *consistency lemma* shows the soundness of the proof system. It is easily shown by induction on the length of the proof.

Lemma 1 *If there exists a proof of a fact $E=0$ (resp. $E=1$), then there is no proof of $E=1$ (resp. $E=0$).*

2.2.2 Proof Examples

We give some proof examples for C_1. We present them in *annotated proof* form, writing at each step the deduced fact, the premises, and the applied deduction rule. Here is an annotated proof P_{01} for C_1 with complete input assignment $I=0,\ J=1$:

(1)	$I=0$	input
(2)	$J=1$	input
(3)	$I\wedge\neg Y=0$	from (1) by (*l-and*)
(4)	$X=0$	from (3) by $(X=I\wedge\neg Y\ :\ 0)$
(5)	$\neg X=1$	from (4) by (*not-0*)
(6)	$J\wedge\neg X=1$	from (2) and (5) by (*b-and*)
(7)	$Y=1$	from (6) by $(Y=J\wedge\neg X\ :\ 1)$

Here is the dual proof P_{10} for $I=1,\ J=0$:

(1)	$J=0$	input
(2)	$I=1$	input
(3)	$J\wedge\neg X=0$	from (1) by (*l-and*)

(4)	$Y = 0$	from (3) by $(Y = J \wedge \neg X\ :\ 0)$
(5)	$\neg Y = 1$	from (4) by (*not-0*)
(6)	$I \wedge \neg Y = 1$	from (2) and (5) by (*b-and*)
(7)	$X = 1$	from (6) by $(X = I \wedge \neg Y\ :\ 1)$

Notice that the deduction ordering is X first, Y next in P_{01}, while it is the reverse ordering Y first, X next in P_{10}. This is the main difference between acyclic and constructive circuits: in acyclic circuits, one can find a data-independent variable ordering valid for all input assignments. In constructive circuits, such an ordering exists for each input assignment, but it may be data-dependent.

2.2.3 Example of Non-Provable Circuits

The circuits $X = X$ and $X = \neg X$ are both rejected as having no output proof, and for the very same reason: there is no way to start a proof. Notice that the existence or non-existence of a Boolean solution is not relevant. The circuit $X = X$, for example, has two Boolean solutions: $X = 0$ and $X = 1$. However, to verify either solution one would have to first make an *assumption* about the solution, and then verify the validity of the assumption. Constructive proofs must only propagate facts; they are not allowed to make assumptions

Constructive Boolean logic rejects the *Hamlet* circuit $X = X \vee \neg X$, for which no output fact can be proven. As above, there is no way to start a proof without making an assumption. The law of excluded middle $X \vee \neg X = 1$ does not hold in constructive logic, unless X has already been proved to be 0 or 1.

2.2.4 Output Proofs and Complete Proofs

An *output proof* is a proof that proves a fact for each output variable. A *complete proof* is a proof that proves a fact for each variable. A circuit is *output constructive* w.r.t. a complete input assignment i if there is an output proof starting with the facts in i. The circuit is *completely constructive* w.r.t. i if there is a complete proof starting with the facts in i.

The difference is that no fact is needed for an intermediate variable in an output proof if this variable is not needed to prove the output facts. It is even allowed that no fact about this variable can be proved. Consider for example $X = I \wedge Y$, $Y = Y$ where only X is an output. If $I = 0$, then $X = 0$ but no fact for Y can be proved. The circuit is output constructive but not completely constructive for this input assignment.

Although output constructiveness seems more general, we shall deal with complete constructiveness in the sequel since it is much easier to handle. Complete constructiveness is also required by the semantics of Esterel [4].

2.2.5 Constructive Logic Matches Delay Independence

Constructive Boolean logic exactly represents delay independence: given a complete input assignment, a circuit electrically stabilizes its output wires (resp. all its wires) for any gate and wire delays if and only if it is output constructive (resp. completely constructive). This fundamental result is shown in [18, 17] using techniques originally developed for asynchronous circuit analysis [7].

Notice that a given fact can have several proofs. Delay assignments actually select proofs. Consider $X = I \wedge Y$, $Y = J \wedge K$, where X is the output. For $I = J = K = 0$, there are two proofs of $X = 0$: the first one deduces $X = 0$ from $I = 0$, the second one deduces the same fact from $Y = 0$, itself deduced from $J = 0$ and $K = 0$. Electrically speaking, the first proof occurs when $I = 0$ propagates through X's *and* gate before $Y = 0$, while the second proof occurs if there is a long delay on the I input wire, long enough for $Y = 0$ to propagate through X's *and* gate before $I = 0$.

2.3 Scott's Fixpoint Semantics

The classical model of Boolean logic is binary, variables taking values in $B = \{0,1\}$. Constructive Boolean logic has a natural ternary semantic model.

2.3.1 The Ternary Model

The ternary domain is $B_\perp = \{\perp, 0, 1\}$. The *undefined* value $\perp$ (read *bottom*) represents absence or non-provability of information. The domain is partially ordered by Scott's *information ordering* $\perp \leq 0$ and $\perp \leq 1$, the total values 0 and 1 being incomparable[2]. Tuples $x, y \in B_\perp^n$ are partially ordered componentwise: $x \leq y$ iff $x_k \leq y_k$ for all k. Functions are required to be monotonic (increasing): for $f : B_\perp^m \to B_\perp^n$, one must have $f(x) \leq f(y)$ in $B_\perp^n$ if $x \leq y$ holds in $B_\perp^m$. A composition of monotonic functions is monotonic. Functions are partially ordered by $f \leq g$ if $f(x) \leq g(x)$ for all x.

2.3.2 The Fixpoint Theorem

The key result in Scott's semantics is the *fixpoint theorem*, which we state here in a simple case. Let $f : B_\perp^n \to B_\perp^n$ be monotonic, and let a fixpoint of f be an element x of $B_\perp^n$ such that $f(x) = x$. The theorem states that f has a least fixpoint $lfp(f)$, which is the (finite) limit of the increasing sequence

$$\perp \leq f_i(\perp) \leq f_i^2(\perp) \leq f_i^3(\perp) \leq \ldots$$

The function *lfp* that associates the least fixpoint $lfp(f)$ with f is itself monotonic.

2.3.3 The Basic Ternary Operators

The Boolean operators are extended as follows to the ternary logic. There is no choice for negation, which must be monotonically defined by $\neg\perp = \perp$, $\neg 0 = 1$, and $\neg 1 = 0$. For conjunction $\wedge$, we choose the *parallel extension*, which is the least monotone function such that $0 \wedge \perp = \perp \wedge 0 = 0$ and $1 \wedge 1 = 1$; it closely corresponds to electrical gate behavior and to our proof rules. The extension of disjunction $\vee$ is dual.

Other possible extensions of $\wedge$ are the *strict* extension such that $0 \wedge \perp = \perp \wedge 0 = \perp$, the *left sequential* extension such that $0 \wedge \perp = 0$ but $\perp \wedge 0 = \perp$, and the symmetrical right sequential extension. They are definable from the parallel extension in constructive logic (hint: the expression $X \vee \neg X$ has value 1 if and only if X is defined).See [16, 1] for a complete discussion of these extensions. It is interesting to note that the parallel extension cannot be defined in sequential languages such as C and requires a parallel interpretation mechanism, hence its name.

2.3.4 Circuits as Fixpoint Operators

A circuit with input vector $i \in B_\perp^m$ and other variables in vector $x \in B_\perp^n$ defines an equation of the form $x = f(i, x)$, where the k-th component of f is given by the right-hand-side of the equation for x_k. Given an input assignment i, let us write $f_i(x) = f(i, x)$; then f_i is a function from $B_\perp^n$ to itself. We call a *solution* of the circuit w.r.t. i the least fixpoint $lfp(f_i)$ of f_i. For example, in circuit C_1, the least fixpoint for input $I = 0,\ J = 1$ is $X = 0,\ Y = 1$, while the least fixpoint for $I = 1,\ J = 1$ is $X = \perp,\ Y = \perp$.

The next theorem shows that the constructively deducible facts exactly correspond to the fixpoint solution.

Theorem 1 *Given a circuit C defining a function f and an input assignment i, a fact $X = b$, $b \in \{0,1\}$, is constructively provable if and only if the X-component of the least fixpoint of f_i has value b.*

[2] Unfortunately, some authors use $\{0, 1, X\}$ with $0 \leq X$ and $1 \leq X$ to mean the same thing!

The proof is standard and left to the reader (use inductions on term size and proof length).

Notice that the theorem does not require the input assignment to be complete. It is also valid when some inputs are $\perp$. Then, no fact for these inputs can be used in deductions.

This concludes the theory of constructive circuits: electrically stabilizing in a delay-independent way is the same as being provable in constructive Boolean logic or as having a non-$\perp$ value in the least fixpoint.

2.4 Algorithms for Circuit Constructiveness

There are algorithms to detect whether a circuit is constructive for a given input assignment or for all complete input assignments. Here, we present a linear-time algorithm that works for one complete input assignment. It is used in the Esterel v5 compiler, for interpretation mode (option `-I`). Algorithms checking constructiveness for all inputs or for some input classes are much more complex. The BDD-based algorithm used in the Esterel v5 compiler (option `-causal`) is presented in [18, 17, 19]. It will not be considered here.

2.4.1 An Interpretation Algorithm

The running data structure of the algorithm is composed of two sets of facts called DONE and TODO and of an array PRED of integer values indexed by non-input variable names. The TODO set initially contains the input facts, and the DONE set is initially empty. The array entry PRED[X] is initialized to the number of predecessors of X, which is the number of variable occurrences in the definition equation of X, also called the fanin number in the electrical presentation.

The algorithm successively takes a fact from TODO, puts it in DONE, and propagates its constructive consequences, which may add new facts to TODO and decrement the predecessor counts. Propagating the consequences of a fact $V = b$ works as follows:

- All variables that refer to V in their definition decrement their predecessor count according to the number of occurrences of V in their definition.
- If $V = b$ immediately determines that $W = c$, then that fact is added to TODO. This occurs if $b = 0$ and W is defined by a conjunction where V appears positively, in which case $c = 0$, or by a disjunction where V appears negatively, in which case $c = 1$ (symmetrically if $b = 1$). This fact propagation rule corresponds to deduction rules such as (*l-and*) and (*r-and*), possibly combined with (*not-0*) and (*not-1*).
- If the predecessor count of a variable W falls to 0 and the value of W is not yet determined, a new fact $W = c$ is added to TODO, where c is the identity of the definition operator of W, i.e. 1 for $\wedge$ and 0 for $\vee$. This corresponds to rules such as (*b-and*).

2.4.2 Execution Example

For C_1 with inputs $I = 0,\ J = 1$, we start in the following state:

$$\text{TODO}:\ I = 0\ .\ J = 1 \qquad \text{DONE}: \qquad \text{PRED}: X:2\ .\ Y:2$$

We remove $I = 0$ from TODO and put it in DONE. We decrement the predecessor count of X. Since $I = 0$ immediately implies $X = 0$, we add that fact to TODO:

$$\text{TODO}:\ J = 1\ .\ X = 0 \qquad \text{DONE}:\ I = 0 \qquad \text{PRED}: X:1\ .\ Y:2$$

We now process $J = 1$. The only consequence is that the number of predecessors of Y is decremented, since $J = 1$ does not determine Y by itself:

$$\text{TODO}:\ X = 0 \qquad \text{DONE}:\ I = 0\ .\ J = 1 \qquad \text{PRED}: X:1\ .\ Y:1$$

We now process $X = 0$. This fact does not directly determine the value of Y, but it exhausts its predecessor list:

$$\text{TODO}: \qquad \text{DONE}: \ I = 0 \ . \ J = 1 \ . \ X = 0 \qquad \text{PRED}: X : 1 \ . \ Y : 0$$

We can now deduce that the value of Y is 1 since Y is an empty conjunction. We add this fact to TODO:

$$\text{TODO}: \ Y = 1 \qquad \text{DONE}: \ I = 0 \ . \ J = 1 \ . \ X = 0 \qquad \text{PRED}: X : 1 \ . \ Y : 0$$

We have computed all the facts we need. However, it is useful to perform the last step, which will bring us back to a nice clean state. Processing $Y = 1$ puts this fact in DONE and decrements X's predecessor count:

$$\text{TODO}: \qquad \text{DONE}: \ I = 0 \ . \ J = 1 \ . \ X = 0 \ . \ Y = 1 \qquad \text{PRED}: X : 0 \ . \ Y : 0$$

Since we build proofs, the result of the algorithm does not depend on the order in which we pick facts in TODO.

For the input $I = 0, \ J = 0$, here is a run where the output values are computed faster but cleanup is longer:

$$\begin{array}{lll}
\text{TODO}: \ I = 0 \ . \ J = 0 & \text{DONE}: & \text{PRED}: X : 2 \ . \ Y : 2 \\
\text{TODO}: \ J = 0 \ . \ X = 0 & \text{DONE}: \ I = 0 & \text{PRED}: X : 1 \ . \ Y : 2 \\
\text{TODO}: \ X = 0 \ . \ Y = 0 & \text{DONE}: \ I = 0 \ . \ J = 0 & \text{PRED}: X : 1 \ . \ Y : 1 \\
\text{TODO}: \ Y = 0 & \text{DONE}: \ I = 0 \ . \ J = 0 \ . \ X = 0 & \text{PRED}: X : 1 \ . \ Y : 0 \\
\text{TODO}: & \text{DONE}: \ I = 0 \ . \ J = 0 \ . \ X = 0 \ . \ Y = 0 & \text{PRED}: X : 0 \ . \ Y : 0
\end{array}$$

For the non-constructive input $I = 1, \ J = 1$, we rapidly reach a deadlock:

$$\begin{array}{lll}
\text{TODO}: \ I = 1 \ . \ J = 1 & \text{DONE}: & \text{PRED}: X : 2 \ . \ Y : 2 \\
\text{TODO}: \ J = 1 & \text{DONE}: \ I = 1 & \text{PRED}: X : 1 \ . \ Y : 2 \\
\text{TODO}: & \text{DONE}: \ I = 1 \ . \ J = 1 & \text{PRED}: X : 1 \ . \ Y : 1
\end{array}$$

There are no remaining facts in TODO, and yet no fact has been established for X or Y and their predecessor counts are positive.

The following result shows that our algorithm is correct and complete:

Theorem 2 *Let C be a circuit with n variables and i be a complete input assignment. The circuit is output constructive w.r.t. i if and only if the algorithm starts with i and computes a fact for each output variable. The circuit is completely constructive w.r.t. I if and only if the algorithm terminates with all predecessor counts 0.*

For a completely constructive circuit, the algorithm always takes the same number of steps, which is the sum of all the fanin counts.

3 POLIS and the CFSM model

Recall our goal is to implement synchronous circuits within the POLIS system. POLIS [3] is a software tool developed at UC Berkeley for the synthesis of control-dominated reactive systems that are targeted for mixed hardware/software implementations. The primary feature of POLIS is its underlying CFSM model of computation; it is within this model that we implement synchronous circuits.

3.1 CFSMs : Overview

The model of computation consists of a network of communicating Codesign Finite State Machines (CFSMs). The communication style is called GALS: globally asynchronous locally synchronous. At the node level, each CFSM has synchronous semantics: when run, a CFSM reads inputs, computes, and writes outputs instantaneously. At the network level, the CFSMs communicate asynchronously: communication is done via data transmission through buffers, and no assumptions are made about the relative delays of the computations performed by each CFSM or about the delays of the data transmission.

3.2 CFSM Communication

Each CFSM has a set of inputs and outputs, and CFSMs are connected with nets. A *net* associates an output of one CFSM to some inputs of other CFSMs. The information transmitted between CFSMs is composed of a *status* and a *value* which are stored in 1-place communication buffers. For each net, there is one associated *value buffer* and multiple *status buffer*s, one for each attached CFSM input. Thus, each CFSM has a local copy of the status of each of its inputs, while the value is stored in a shared buffer. A *CFSM input buffer* is composed of the local status buffer and the shared value buffer. [3] The status buffer stores either 1 or 0, representing presence or absence of valid data in the value buffer.

A *CFSM input assignment* is the set of values stored in the input buffers for a CFSM. It is equivalent to the circuit input assignment given in Section 2.1.1. A CFSM input assignment may be complete or partial. A *captured input assignment* corresponds to the statuses and values that are actually read from the buffers when a CFSM in run.

3.3 CFSM Computation

A CFSM computation is called a *CFSM execution* or *CFSM run*. When a CFSM executes, it reads its inputs, makes its computation, writes its outputs, and resets (*consumes*) its inputs.

Input reading: A CFSM *atomically* reads and resets the status buffers: it simultaneously reads all status buffers and sets them to 0, ready for the arrival of new inputs.[4]

It subsequently reads the values of the present inputs. This determines the captured input assignment.

Computation: The CFSM uses the captured input assignment to make its computation: it computes its outputs and next states based on the values given in its state transition table. The computation is done synchronously, which means that the CFSM reacts precisely to the captured input assignment, regardless of whether the inputs change while the CFSM is computing.

Output writing: For each output, a CFSM writes the value buffer and subsequently atomically sets the status buffers for each associated CFSM input. [5]

A *CFSM-event* consists of an output emitting its data and the corresponding input status buffers being set to 1.

3.4 CFSM Network Computation

A network computation is called a *network execution* or *network run* and corresponds to several CFSM executions.

Each CFSM network has an associated scheduler(s). The scheduler continuously reads the current input assignments, determines which CFSMs are runnable, and chooses the order in

[3] Note that in [3], the word *event* is used both for the status alone and for the status/value pair.

[4] In POLIS, a CFSM may have an *empty execution*, which means that it does not react to its current inputs. In this case, the current inputs are saved, and any inputs that are received while the CFSM is determining its empty reaction are added to the input assignment, which is restored and thus read at the next run. We do not use this feature here.

[5] Atomic reads and writes are more expensive, since they require an implementation that guarantees that these actions can happen simultaneously. The decision was made in POLIS to make status buffer reading and writing atomic, and not value-buffer reading and writing, because atomically reading and writing of short bit strings can be implemented efficiently, and because this guarantees certain desirable behavioral properties in the system.

which to run them.[6] A CFSM is *runnable* if it has at least one input status buffer set to 1. A CFSM is run by the scheduler sometime after it is runnable.

Typically, an input assignment is given to the network, and the scheduler runs the CFSMs according to its schedule until there are no further changes in the communication buffers. This is called a *complete network execution.*

Time effectively passes when control is returned to the scheduler, and thus instantaneous communication between CFSM modules is not possible.

4 Implementing Constructive Circuits in CFSM Networks

In this section, we explain our realization of the synchronous behavior of a circuit on a CFSM network. To facilitate the exposition, we restrict ourselves to the extreme case of one CFSM per gate. More realistic levels of granularity will be handled in Section 5.

In Section 2.4, we presented an algorithm to compute the behavior of a circuit for a given circuit input assignment. The essential ingredients were a set TODO of facts to propagate, a set DONE of established facts, and a predecessor counter for each variable. The basic idea of the CFSM network implementation presented here is to distribute a similar algorithm over a network of CFSMs, associating a CFSM with each circuit gate (equation).

We start by studying the reaction to a single input assignment and then present various ways of chaining reactions to handle circuit input assignment sequences, to obtain the cyclic behavior characteristic of synchronous systems.

4.1 Fact propagation in a CFSM network

We implement each gate as a CFSM that reads and write facts, which are encoded in POLIS CFSM-events sent by one gate to its fanouts. The arrival of a fact at a gate makes the gate runnable, and, when run, if there is a provable output fact from the facts received so far, the gate CFSM outputs it. Fact propagation between gates is directly performed by the underlying POLIS scheduling and CFSM-event broadcasting mechanisms. A POLIS execution schedule is thus precisely a proof (fact propagation) ordering.

Facts arrive sequentially at a gate CFSM. Therefore, a combinational circuit gate must be implemented by a sequential CFSM that remembers which facts it has received so far. The sequential state of a gate CFSM encodes the number of predecessors of the interpretation algorithm of Section 2.4.

4.2 The Basic Gate CFSM

For ease of exposition, we write the gate CFSMs in Esterel. This makes the gate specification very flexible, which will be useful in the next sections. No preliminary knowledge of Esterel is required.

To handle our running example C_1, it suffices to describe the *AndNot* gate $C = A \wedge \neg B$. Other gates are similar. The Esterel program for *AndNot* has the following interface:

```
module AndNot :
input A : boolean, B : boolean;
output C : boolean;
```

Here, `A`, `B`, and `C` are Esterel signals of type `boolean`, the values of which are called `true` and `false`. Esterel signals are just like POLIS buffers, with some additional notation. An event of a `boolean`-valued signal such as `A` has two components: a binary *presence status* component, also written `A`, which can take values *present* and *absent*, and a *value* component of type `boolean`, written `?A`. We choose to encode the fact $A = 0$ (resp. $A = 1$) by `A` present with value `true` (resp.

[6] In POLIS, scheduler is automatically synthesized with parameters, such as the type of scheduling algorithm, given by the user.

false)[7]. Notice that we use two pieces of information, the status and the value, to represent a fact, i.e. the stable value of a wire. A *present* status component indicates stability, i.e. that a fact has been propagated to this point, and the value component represents the Boolean value of the fact.

Like a POLIS captured input assignment, An Esterel input assignment defines the presence status of each input signal and the value of each present signal. For instance, for AndNot, A(true).B(false) is an Esterel input assignment in which A is present with value true and B is present with value false, encoding the facts $A = 0$ and $B = 1$, and A(false) is an input assignment where A is present with value false and B is absent, encoding the fact $A = 0$.

Like a CFSM, an Esterel program repeatedly reacts to an externally provided input assignment by generating an output assignment. The processing of an input assignment is also called a *reaction* or an *instant*. In POLIS, a run of an Esterel CFSM triggers exactly one reaction of the Esterel program, with the same input assignment.

Unlike in POLIS, communication in Esterel is instantaneous: a signal emitted by a statement is instantaneously received by all the statements that listen to it. Similarly, control propagation is instantaneous; for example, in a sequence '$p; q$', q immediately starts when p terminates. The only statements that break the flow of control are explicit delays such as "await S" that waits for the *next* occurrence of a signal S.

Finally, in Esterel, signal presence status is not memorized from reaction to reaction, but value is : the value of the Esterel expression ?A of A in a reaction where A is absent is the one it had in the previous reaction. Notice that the value of a signal may change only when the signal is present.

Our first attempt to write the Esterel body of AndNot is:

```
[
   await A;
   if not ?A then emit C(false) end if
||
   await B;
   if ?B then emit C(false) end if
];
if (?A and not ?B) then emit C(true) end if
```

The program reads as follows. First, we start two parallel threads. The first thread waits for the presence of A, and the second threads waits for the presence of B. The first input assignment can have A present, B present, or both (an empty assignment with neither A nor B present would leave the program in the same state; such an assignment is permitted in Esterel but will never be generated by the POLIS scheduler). If A is absent, the first thread continues waiting. If A is present, the first thread immediately checks A's value ?A and immediately outputs C(false) if ?A is false, thus mimicking the (*l-and*) deduction rule; the thread terminates immediately in either case. The second thread behaves symmetrically but checks for the truth of ?B to emit C(false). If both A and B are present, the threads evolve simultaneously.

The Esterel parallel construct '||' terminates immediately when both branches have terminated. Therefore, the above parallel statement terminates exactly when both A and B have been received, either simultaneously or in successive input assignments. In that instant, C(true) is emitted if the possibly memorized values ?A and ?B are respectively true and false, mimicking the (*b-and*) deduction rule with negated second argument.

4.2.1 Avoiding Double Output

Our gate CFSM almost works, but not quite, since C(false) can be emitted twice (possibly at different instants) if ?A is false and ?B is true. The gate should output C only once. To correct this problem, we use an auxiliary Boolean signal Caux:

[7] Other equivalent encodings can be considered. One can for example use a pair of pure signals for each variable, one for presence and one for value. The encoding we use makes a clear difference between availability and value.

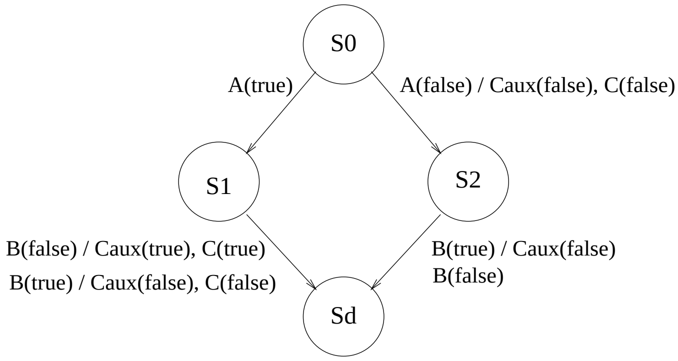

Figure 2: Partial state transition graph for `module AndNot`

```
signal  Caux : combine boolean with and  in
   [
      await A;
      if not ?A then emit Caux(false) end if
   ||
      await B;
      if ?B then emit Caux(false) end if
   ];
   if (?A and not ?B) then emit Caux(true) end if
||
   await Caux;
   emit C(?Caux)
end signal
```

The first branch of the outermost parallel behaves as before but emits `Caux` instead of `C`. The second branch waits for `Caux` to emit `C` with the same value, and immediately terminates. If `Caux` is emitted twice in succession by the first branch, the second emission is simply unused since the "`await Caux`" statement has already terminated. The "`combine boolean with and`" declaration smoothly handles simultaneous double emission, also called *collision*. For this example, collision occurs if `A(false)` and `B(true)` occur simultaneously, in which case both "`emit C(false)`" statements are simultaneously executed. The `combine` declaration specifies that the result value `?Caux` is the conjunction of the separately emitted values. Here, we could as well use disjunction, for only `false` values will be combined.

4.2.2 The Gate CFSM State Graph

The gate CFSM state transition graph (STG) is partially shown in Figure 2. The transitions are shown for the cases in which `A` is received before `B`, the other cases (`B` arriving first or `A` and `B` arriving simultaneously) are similar and not pictured. This partial STG is shown to help visualize the sequential state traversal in a familiar syntax, but is not a practical input mechanism for reactive modules compared to the Esterel language. For example, a module that waits for n signals concurrently will have 2^n states, while the Esterel description has size n. Note also that the C_{aux} signal is shown in the output list for visualization purposes; it is an internal signal that is not seen by any other module.

4.2.3 Gate CFSM Execution Example

To become familiar with the Esterel semantics, let us run the `AndNot` program on two different input assignment sequences. We start in state S_0 where we are waiting for the inputs `A` and `B` and internally for `Caux`, pictured by underlining the active `await` statements:

```
signal  Caux : combine boolean with and  in
   [
      await A;
      if not ?A then emit Caux(false) end if
   ||
      await B;
      if ?B then emit Caux(false) end if
   ];
   if (not ?A and ?B) then emit Caux(true) end if
||
   await Caux
   emit C(?Caux)
end signal
```

Assume the first gate input assignment is `A(true)` and `B` absent. Then, "`await A`" terminates, and we execute the test for "`not ?A`"; since the test fails, the first parallel branch terminates without emitting `Caux`. We then reach state S_1, in which we continue waiting for `B` and `Caux`:

```
signal  Caux : combine boolean with and  in
   [
      await A;
      if not ?A then emit Caux(false) end if
   ||
      await B;
      if ?B then emit Caux(false) end if
   ];
   if (?A and not ?B) then emit Caux(true) end if
||
   await Caux;
   emit C(?Caux)
end signal
```

If we now input `B(false)`, we execute the `?B` test, which also fails. Since the second parallel branch terminates, the parallel statement terminates immediately; we execute the "`?A and not ?B`" test, which succeeds. We emit `Caux(true)`, which makes the "`await Caux`" statement instantaneously terminate; the output `C(true)` is emitted, since ?`Caux` = `true`. We reach the dead state S_d where no signal is awaited.

Assume now that the first gate input assignment is `A(false)` and `B` absent. Then, starting from S_0, we execute the first test, which succeeds and emits `Caux(false)`. The "`await Caux`" statement immediately terminates and `C(false)` is emitted. We continue waiting for `B`, in the following state S_2:

```
signal  Caux : combine boolean with and  in
   [
      await A;
      if not ?A then emit Caux(false) end if
   ||
      await B;
      if ?B then emit Caux(false) end if
   ];
   if (?A and not ?B) then emit Caux(true) end if
||
   await Caux;
   emit C(?Caux)
end signal
```

Then, when `B` occurs in a later input assignment, the "`await B`" statement terminates and the program reaches the dead state S_d. If `?B` is true, the emission of `Caux(false)` is performed but unused. This last step of waiting for `B` mimics the last cleanup step of the propagation algorithm of Section 2.4. It will be essential to chain cycles in Section 4.4.

If `A` and `B` occur together in the first input assignment, then `AndNot` immediately emits `C` with the appropriate value and transitions directly from state S_0 to dead state S_d.

Notice that the number of predecessor waited for in the algorithm of Section 2.4 is exactly the number of underlined statements among "`await A`" and `await B`".

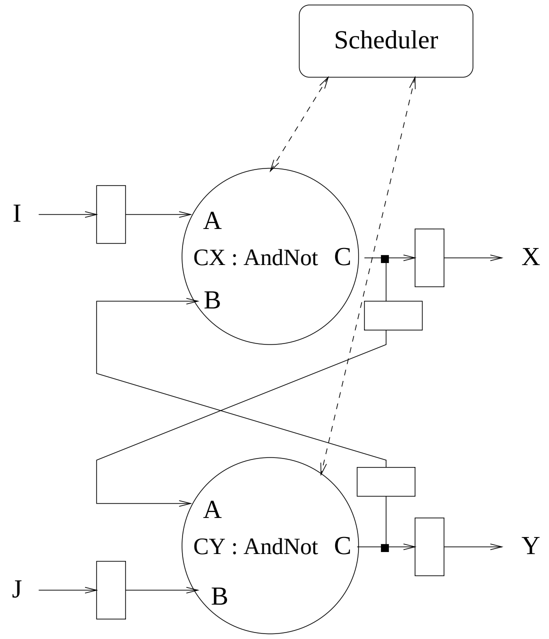

Figure 3: CFSM network for circuit C_1

4.3 Performing a Single Reaction on a Network of Gates

Given a circuit C, the CFSM network for C is obtained by creating an input buffer for each input signal in C, an output buffer for each output signal, and a gate CFSM for each equation in C. Gate CFSM outputs are broadcast to the gate CFSMs that use them, as specified by the circuit equations.

To run the network for a given circuit input assignment i, it suffices to put the input values defined by i in each of the network input buffers. Then, the gate CFSMs directly connected to inputs become runnable. As soon as a gate has computed its result, it puts it in its output buffer, the result's value is automatically transferred to all fanout CFSM input buffers by the network, and these CFSMs become runnable.

4.3.1 An Execution Example

Consider the network for C_1, pictured in Figure 3, where the CFSMs for X and Y are called `CX` and `CY`. The rectangular buffers are the 1-place buffers used to communicate CFSM-events between modules. Note that there are two information storage mechanisms at work during the execution of this circuit:

1. The CFSM-gates as implemented by the Esterel modules internally store which signals they have received and thus which they are still waiting for using their implicit states.

2. The CFSM-network as implemented in POLIS stores a copy of each CFSM-event, one for each fanout of that event, using the 1-place buffers.

Consider the input assignment $I = 0$ and $J = 1$. We first put `false` in I's buffer and `true` in J's buffer. The CFSMs `CX` and `CY` become runnable. Assume `CX` is run first. Then it captures the partial input assignment `A(false)` and `B` absent, which encodes $I = 0$. The `CX` CFSM outputs `C(false)`, which is the encoding for $X = 0$, and goes to state S_1. The `false` event is made visible at `CY`'s `B` input buffer after some time.

- Assume first that `CY` is run before the arrival of `CX`'s output. Then `CY` captures the partial input assignment `A(true)` and `B` absent, which encodes the fact $J = 1$. The `CY` CFSM emits no output and continues waiting for its `B` input, in state S_2. When X's `false` value is written in `CY`'s `B` input buffer, `CY` is made runnable and runs with captured input assignment `A` absent and `B(false)`; it emits `C(true)`, which encodes $Y = 1$, and goes to the dead state.

- Assume instead that `CX`'s `false` output is written in `CY`'s input buffer `B` before `CY` is run. Then, when `CY` is later run, it captures the complete input assignment `A(true).B(false)`, which encodes the facts $J = 1$ and $X = 0$. It emits `C(true)` and goes directly to the dead state.

Once `CY` has emitted its output `C(true)`, the `true` value is written in `CX`'s input buffer `B`, and `CX` is made runnable again. Then, `CX` is run with input assignment `B(true)` and `A` absent, which encodes $Y = 1$, and `CX` goes to the dead state.

4.3.2 Correctness of the CFSM Implementation

The CFSM network computes a proof in the same way as the interpretation algorithm of Section 2.4, but with dynamic and concurrent scheduling of fact propagation. Building a new fact is equivalent to generating a CFSM-event. Propagating a fact is equivalent to broadcasting the CFSM-event to the fanouts and running the fanout CFSMs, which is exactly what the network automatically provides.

The following theorem summarizes the results:

Theorem 3 *Let C be a circuit. Let n be the number of output or local variables (fanouts), and let f be the number of variable occurrences in the right-hand-sides of C's equations (fanins). Let i be a circuit input assignment. For any run of the network associated with C initialized with i, the following holds:*

1. *The number of created CFSM-events is bounded by n, and the number of CFSM runs is bounded by f. No buffer overwrite can occur.*

2. *If, in some complete network execution sequence, exactly n CFSM-events have been created, then the implemented circuit is completely constructive w.r.t. i, and the output gate CFSM generated events are the encodings of the output values of C w.r.t. i. All complete execution sequences give the same result independent of the schedule, and all gate CFSMs terminate in the dead state once all CFSM-events have been processed.*

3. *If, for some complete run, less than n CFSM-events have been created, then this is true for all runs and C is not completely constructive w.r.t. i.*

Output constructive circuits can be handled by a slight modification of the result, but loosing the nice fact that all gate CFSMs terminate in the dead state, which if useful when chaining reactions, which we demonstrate in the next section.

4.4 Chaining Reactions

A synchronous circuit or program is meant to be used sequentially, the user or RTOS providing a sequence of input assignments and reading a sequence of output assignments. In our POLIS implementation, the user alternates writing circuit input assignments in the network input buffers and reading the computed circuit output assignments in the network output buffers. Since POLIS uses 1-place buffers for communication, we must make sure that no buffer overwrite occurs in the network. In particular, we cannot let the user overwrite an input buffer until its value has been completely processed by the gates connected to it. Here are four possible user-level protocols:

- *Wait for a given amount of time.* This is the technique used for single-clocked electrical circuits. Since the number of operations to be performed is uniformly bounded, if the underlying machinery (CPUs, network, etc...) has predictable performance, we are

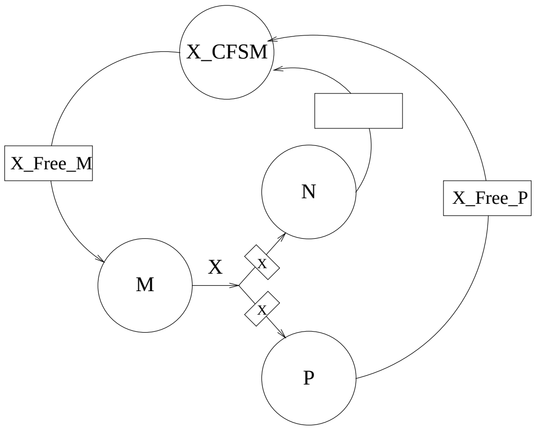

Figure 4: Circuit C_1

guaranteed that the reaction is complete after a maximal (predictable) time and that no buffer overwriting occurs. This solution is often used in cycled-based control systems implemented in software and in Programmable Logic Controllers (PLCs). This protocol can be realized in our implementation with the addition of performance estimation, in order to compute the frequency with which new inputs can be fed to the synchronous circuit.

- *Compute and return a termination signal.* If the circuit is completely constructive w.r.t. the input, we know that the computation has finished when all the gate CFSMs have read all their inputs, i.e. when the network has processed a given number of CFSM-events. We can either modify the scheduler to have it report completion to the user or build an explicit termination signal by having each gate output a separate CFSM-event when it has processed all inputs. These CFSM-events are gathered by an auxiliary gate that generates a termination event for the user when all its input have arrived. These centralized solutions are not in the spirit of distributed systems.

- *Implement a local flow control protocol at each gate CFSM.* This is a much more natural solution in a distributed setting and it makes it possible to *pipeline* the execution: for each input, the user may enter a new value as soon as the flow-control protocol says so, without waiting for the reaction to be complete. The protocol must ensure that an input for a conceptual synchronous cycle never interferes with values for other cycles.

- *Queue input events*: this solution is used in [9, 8]. It implies that the user can always write new inputs and is never blocked. In our implementation, the same flow control problem is simply pushed inside the network, since CFSMs do not communicate using queues.

We now present a flow-control protocol that supports pipelining. The reactions remain globally well-ordered as required by the synchronous model: the n-th value of input I is processed in the same conceptual synchronous cycle as the n-th value of input J; however, because of pipelining, internal network CFSM scheduling and CFSM-event generation can occur in intricate orderings.

To make the gate reusable, is suffices to embed their bodies into an Esterel "`loop...end`" infinite loop. Then, instead of going to the dead state, a gate CFSM returns to its initial state. This is why it is much easier to handle complete proofs. To deal with more general output proofs, we should add complicated gate reset mechanism, while reset is automatically performed by complete proofs.

Thanks to the flexibility of Esterel code, the protocol only requires a slight modification of our basic gate code, and the addition of a new module. The corresponding CFSM network is shown in Figure 4.

Consider an output `X` of a CFSM `M`, read for example by two other CFSMs `N` and `P`. With `X` and `N` (resp. `P`) we associate a signal `X_Free_N` (resp. `X_Free_P`) that is written by `N` (resp.

P). With X and M we associate a signal X_Free_M read by M and written by an auxiliary module X_CFSM which consumes X_Free_N and X_Free_P and writes X_Free_M when both X_Free_N and X_Free_P have received a value. The buffers in Figure 4 for each signal are those used by POLIS; the actual information determining when the signal X is free to be written by M is contained in the implicit states of X_CFSM. The new module is written as follows:

```
module X_CFSM:
input X_Free_N, X_Free_P;
output X_Free_M;
loop
   [
      await X_Free_N
   ||
      await X_Free_P
   ];
   emit X_Free_M
end loop
end module
```

Similarly, for a network input I broadcast to N and P, we generate a network output buffer I_Free filled by the auxiliary CFSM reading I_Free_N and I_Free_P, and for any network output O a network input buffer O_Free filled by the user when it is ready to accept a new value of O.

We require M to write its X output only when X_Free_M holds 0, then consuming that value. We require N (resp. P) to write 0 in X_Free_N (resp. X_Free_P) when it reads its local copy of the input X. The AndNot CFSM is modified as follows:

```
module AndNot :
input A : boolean, B : boolean;
output A_Free, B_Free;
output C : boolean;
input C_Free;

loop
   signal  Caux : combine boolean with and  in
      [
         await A;
         emit A_Free;
         if not ?A then emit Caux(false) end if
      ||
         await B;
         emit B_Free;
         if ?B then emit Caux(false) end if
      ];
      if (?A and not ?B) then emit Caux(true) end if
   ||
      [
         await Caux;
      ||
         await C_Free
      ];
      emit C(?Caux)
   end signal
end loop
```

The output C is emitted only when the last of Caux and C_Free has been received.

When the gate CFSM is instantiated at a node M, the A_Free, B_Free, and C_Free buffers must be appropriately renamed A_Free_M, B_Free_M, and C_Free_M, to avoid name clashes.

The flow-control mechanism acts in two ways. First, it prevents buffer overwriting. Second, it makes pipelining possible. Given a circuit input assignment i_n at cycle n, the new value of a circuit input I for cycle $n+1$ can be written in I's network input buffer as soon as `I_Free` is full. Therefore, it is not necessary to wait for the global end of a cycle to locally start a new one.

We have a last technical problem to solve. Assume that an `AndNot` gate CFSM starts circuit cycle n. Assume that the gate CFSM receives an `A` input event, say `A(false)` with `B` absent. The gate sends back `A_Free`. From then on, the gate can receive two inputs:

- The `B` input event that holds `B`'s value in cycle n. This input should be processed normally since the gate CFSM is currently processing cycle n.
- The out-of-order `A` input event that holds `A`'s value for cycle $n+1$. Processing this input should be deferred until `B` has been processed.

In the current POLIS network model, a CFSM is made runnable as soon as it receives an input event. Therefore, the gate can be made runnable with input `A` for cycle $n+1$ while it is still processing cycle n. At this point, the gate should either internally memorize `A`'s value or rewrite it in the `A` buffer, leaving in both cases the `A_Free` flow control buffer empty until it has finished cycle n. Both solutions are expensive and somewhat ugly.

We suggest a slight modification to the POLIS scheduling policy. A CFSM should tell the scheduler which input buffers it is currently interested in, and the scheduler should not make the CFSM runnable if none of these buffers holds an event. When the CFSM is run, its captured input assignment should only contain the events in the buffers the CFSM is explicitly waiting for, leaving the rest in their input buffers. In the above example, the gate CFSM tells the scheduler it is only waiting for `B`. If the new value of `A` comes in, the CFSM is not made runnable. When `B` occurs, the gate is made runnable, and it will run with input `B` only. Once the gate has processed `B`, it tells the scheduler that it is now waiting for both `A` and `B`. Since `A` is already there, the gate can be immediately made runnable again.

The final version of the gate CFSM involves the auxiliary `Wait` signals sent to the scheduler to implement this mechanism:

```
module AndNot :
input A : boolean, B : boolean;
output A_Free, B_Free;
output A_Wait, B_Wait;
output C : boolean;
input C_Free;
output C_Free_Wait;

loop
   signal  Caux : combine boolean with and  in
      [
         abort
            sustain A_Wait
         when A;
         emit A_Free;
         if not ?A then emit Caux(false) end if
      ||
         abort
            sustain B_Wait
         when B;
         emit B_Free;
         if ?B then emit Caux(false) end if
      ];
      if (?A and not ?B) then emit Caux(true) end if
   ||
      [
         await Caux;
      ||
         abort
            sustain C_Free_Wait
```

```
            when C_Free
        ];
        emit C(?Caux)
    end signal
end loop
```

The "`Await A`" statement has become "`abort sustain A_Wait when A`". The "`sustain A_Wait`" statement emits `A_Wait` in each clock cycle. the "`abort` p `when A`" aborts its body p right away when `A` occurs, not executing p at abortion time. Therefore, `A_Wait` is emitted until `A` is received, that instant excluded.

5 Mixed Synchronous/Asynchronous Implementation

We now have two very different levels of granularity for implementing an Esterel program in POLIS: compiling the program into a single CFSM node or building a separate CFSM for each gate of the program circuit. The first does not support distribution, while the second is clearly too inefficient: the associated overhead is unacceptable for large programs since it involves scheduling each individual gate CFSM multiple times.

We now briefly explain how we can deal with many other implementation choices with different levels of granularity, using the compositional and incremental character of the constructive semantics. When doing so, we retain the full synchronous semantics of the program, but we trade off synchrony and asynchrony *in the implementation*.

The idea as one moves to a larger granularity implementation is to partition the set of gates into gate clusters $G_1, G_2, \ldots, G_p$. Each cluster G_k groups its gates into a single CFSM, the clusters being connected by the POLIS network as before. The partition can be arbitrary, and chosen to match any locality or performance constraints. Facts are processed both synchronously and asynchronously, but again their proofs are derived from the synchronous constructive semantics. In particular, synchronous fact processing is done *within a cluster* using the algorithm of Section 2.4, in a single CFSM and in one computation of that CFSM; asynchronous fact processing is done *across the network* and thus between CFSMs. Some facts will be both synchronously and asynchronously processed, e.g. an output from gate g_1 that is an input to another gate g_1' in the same cluster G_1 and to g_2 in another cluster G_2.

What makes this possible is the ability of our centralized and distributed algorithms to deal with *partial deduction*: given a partial input assignment i, both algorithms generate all the facts that can be deduced from i. If a new fact is added to i, the algorithms incrementally deduce its consequences. Therefore, it does not matter whether facts are handled synchronously in a gate cluster or asynchronously in the POLIS cluster network.

Consider for example the following circuit C_2 obtained by adding an output Z to C_1:

$$C_2 \left\{ \begin{array}{l} X = I \wedge \neg Y \\ Y = J \wedge \neg X \\ Z = X \wedge Y \end{array} \right.$$

Consider first the clusters $G_1 = \{X, Y\}$ and $G_2 = \{Z\}$. Assume that we receive the fact $I = 0$. Then, G_1 deduces $X = 0$ and outputs that fact to G_2, which can make a local transition to reach the state S_1 where it waits only for Y; G_1 also internally remembers in its local state that Y has lost a predecessor. Thus, $X = 0$ was synchronously propagated to Y in the same cluster, and asynchronously propagated to Z in the other cluster through another call to a CFSM,. If we now receive $J = 1$, G_1 deduces $Y = 1$ and sends that fact to G_2, which can now output $Z = 0$.

With the same input sequence, consider the clusters $G_1 = \{X, Z\}$, $G_2 = \{Y\}$. When receiving $I = 0$, G_1's CFSM instantaneously generates the facts $X = 0$ and $Z = 0$, so $Z = 0$ is determined synchronously. The fact $X = 0$ is asynchronously propagated to G_2 by the network, and G_2's CFSM transitions to a state where it waits only for J. When $J = 1$ occurs, the CFSM outputs $Y = 1$; that fact is propagated to G_1's CFSM, which goes back to its initial state.

Optimal solutions to the problem of determining a set of clusters is beyond the scope of this paper. A number of clustering algorithms exist in the literature, and the design may be entered in a partitioned fashion that leads to a natural clustering as well. In our case, clustering according to the source code module structure is an obvious candidate for a clustering heuristic, as well as clustering according to the frequency of use of signals (like clocks in Lustre). Here, we simply point out that our algorithms and the semantics behind them permit any level of granularity: from individual gates implemented as separate CFSMs, to an entire synchronous program implemented as a single CFSM. Thus, the tradeoff between synchronous and asynchronous implementation of a synchronous program can be fully explored.

6 Conclusions and Future Work

We have described a method for implementing synchronous Esterel programs or circuits on globally asynchronous locally synchronous (GALS) POLIS networks. The method is based on fact propagation algorithms that directly implement the constructive semantics of synchronous programs. We have developed flow-control techniques that automatically ensure that no POLIS buffer can be overwritten and that make pipelining possible.

Initially, we have associated a POLIS CFSM with each circuit gate, which is unrealistic in practice. However, our method is fully compositional, and fact propagation can be performed either synchronously in a node or asynchronously between nodes. This makes it possible to cluster gates into bigger synchronous nodes and to explore the tradeoff between synchronous and asynchronous implementation.

For simplicity, we have only dealt with the pure fragment of Esterel where signals carry no value. Extension to full value-passing Esterel constructs raises no particular difficulty. A complete implementation is currently being developed.

References

[1] R. Amadio and P.L. Curien. *Domains and Lambda-Calculi.* Cambridge University Press, 1998.

[2] C. André. Representation and Analysis of Reactive Behaviors: A Synchronous Approach. In *Proc. CESA'96*, Lille, France, July 1996.

[3] F. Balarin, M. Chiodo, P. Giusto, H. Hsieh, A. Jurecska, L. Lavagno, C. Passerone, A. Sangiovanni-Vincentelli, E. Sentovich, K. Suzuki, and B. Tabbara. *Hardware-Software Co-Design of Embedded Systems: The POLIS Approach.* Kluwer Academic Press, June 1997.

[4] G. Berry. *The Constructive Semantics of Esterel.* Draft book, preliminary version available from http://www.inria.fr/meije/esterel, 1995.

[5] G. Berry. *The Foundations of Esterel. To appear in* Proof, Language and Interaction: Essays in Honour of Robin Milner, G. Plotkin, C. Stirling and M. Tofte, editors, MIT Press, 1998. See http://www.inria.fr/meije/esterel, 1998.

[6] G. Berry and G. Gonthier. The Esterel Synchronous Programming Language: Design, Semantics, Implementation. *Science of Computer Programming*, 19(2):87–152, 1992.

[7] J. A. Brzozowski and C-J. Seger. *Asynchronous Circuits.* Monographs in computer science. Springer-Verlag, New York, 1995.

[8] B. Caillaud, P. Caspi, A. Girault, and C. Jard. Distributing automata for asynchronous networks of processors. *European Journal of Automation (RAIRO-APII-JESA)*, 31(3):503–524, 1997.

[9] P. Caspi, A. Girault, and D. Pilaud. Distributing reactive systems. In *Seventh International Conference on Parallel and Distributed Computing Systems, PDCS'94*, Las Vegas, USA, October 1994. ISCA.

[10] P. Le Guernic, M. Le Borgne, T. Gauthier, and C. Le Maire. Programming Real-Time Applications with Signal. *Another Look at Real Time Programming, Proceedings of the IEEE, Special Issue*, September 1991.

[11] N. Halbwachs. *Synchronous Programming of Reactive Systems*. Kluwer, 1993.

[12] N. Halbwachs, P. Caspi, and D. Pilaud. The Synchronous Dataflow Programming Language Lustre. *Another Look at Real Time Programming, Proceedings of the IEEE, Special Issue*, September 1991.

[13] D. Harel. Statecharts: A Visual Approach to Complex Systems. *Science of Computer Programming*, 8:231–274, 1987.

[14] C. A. R. Hoare. *Communicating Sequential Processes*. International Series in Computer Science. Prentice-Hall, 66 Wood Lane End, Hemel Hempstead, Hertfordshire, HP2 4RG, UK, 1985.

[15] G. Kahn. The Semantics of a Simple Language for Parallel Programming. In *Proc. of the IFIP Congress 74*. North-Holland Publishing Co., 1974.

[16] G. Plotkin. LCF as a programming language. *Theoretical Computer Science*, 5(3):223–256, 1977.

[17] T. Shiple. *Formal Analysis of Cyclic Circuits*. PhD thesis, U.C. Berkeley, 1996.

[18] T. Shiple, G. Berry, and H. Touati. Constructive Analysis of Cyclic circuits. In *Proceedings of European Design and Test Conference*, March 1996.

[19] H. Toma. *Analyse Constructive et Optimisation Séquentielle des Circuits Génerés à partir du Langage Synchrone Réactif Esterel*. PhD thesis, Ecole des Mines de Paris, Centre de Mathématiques Appliquées, 1997.