INSTITUT FÜR INFORMATIK

UND PRAKTISCHE MATHEMATIK

Efficient Compilation of Cyclic Synchronous Programs

Jan Lukoschus, Reinhard von Hanxleden,
Stephen A. Edwards

Bericht Nr. 0402
April 2004

CHRISTIAN-ALBRECHTS-UNIVERSITÄT

KIEL

Institut für Informatik und Praktische Mathematik der
Christian-Albrechts-Universität zu Kiel
Olshausenstr. 40
D – 24098 Kiel

Efficient Compilation of Cyclic Synchronous Programs

Jan Lukoschus[1], Reinhard von Hanxleden[1],
Stephen A. Edwards[2]

Bericht Nr. 0402
April 2004

[1]Christian-Albrechts-Universität zu Kiel, Institut für Informatik und Praktische Mathematik, Olshausenstr. 40, 24098 Kiel, Germany; E-mail: {jlu,rvh}@informatik.uni-kiel.de

[2]Columbia University, Department of Computer Science, New York, NY 10027; E-mail: sedwards@cs.columbia.edu

Abstract

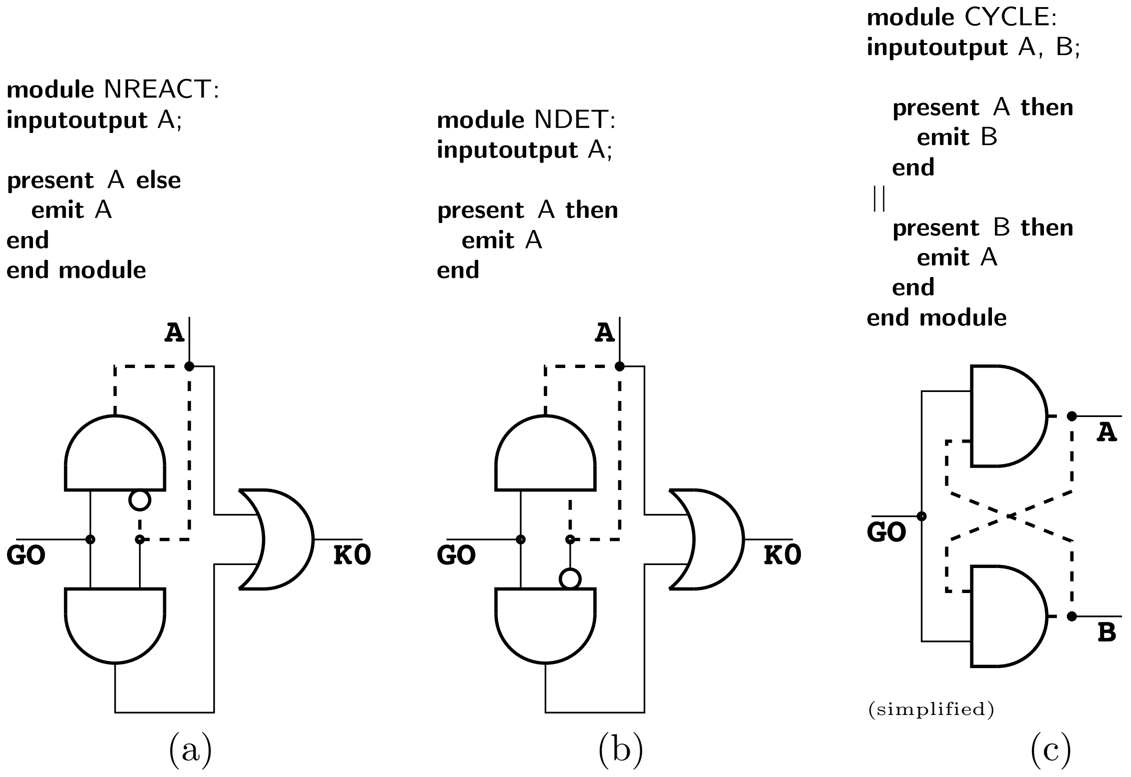
Synchronous programs may contain cyclic signal interdependencies. This prohibits a static scheduling, which limits the choice of available compilation techniques for such programs. This paper proposes an algorithm which, given a constructive synchronous program, performs a semantics-preserving source-level code transformation that removes cyclic signal dependencies, and also exposes opportunities for further optimization. The transformation exploits the monotonicity of constructive programs, and is illustrated in the context of Esterel; however, it should be applicable to other synchronous languages as well. Experimental results indicate the efficacy of this approach, resulting in reduced run times and/or smaller code sizes, and potentially reduced compilation times as well. Furthermore, experiments with generating hardware indicate that here as well the synthesis results can be improved.

1 Introduction

One of the strengths of synchronous languages [2] is their deterministic semantics in the presence of concurrency. It is possible to write a synchronous program which contains cyclic interdependencies among concurrent threads. Depending on the nature of this cycle, the program may still be valid; however, translating such a cyclic program poses challenges to the compiler. Therefore, not all techniques that have been proposed for compiling synchronous programs are applicable to cyclic programs. Hence, cyclic programs are currently only translatable by techniques that are relatively inefficient with respect to execution time, code size, or both. This paper proposes a technique for transforming valid, cyclic synchronous programs into equivalent acyclic programs, at the source-code level, thus extending the range of efficient compilation schemes that can be applied to these programs.

The focus of this paper is on the synchronous language Esterel [5]; however, the concepts introduced here should be applicable to other synchronous languages as well, such as Lustre [17].

Next we will provide a classification of cyclic programs, followed by an overview of previous work on compiling Esterel programs and handling cycles. Section 2 introduces the transformation algorithm for *pure signals*, which do not carry a value. Optimization options are presented in Section 3. Section 4 demonstrates the transformation with further examples, including a program using *valued signals*. Section 5 provides experimental results, the paper concludes in Section 6.



```
module NREACT:
inputoutput A;

present A else
  emit A
end
end module
```

```
module NDET:
inputoutput A;

present A then
  emit A
end
```

```
module CYCLE:
inputoutput A, B;

  present A then
    emit B
  end
||
  present B then
    emit A
  end
end module
```

Figure 1: Invalid cyclic Esterel programs. The wires shown as dashed lines indicate the cyclic dependencies.

1.1 Cyclic Programs

The execution of an Esterel program is divided into discrete *instants*. An Esterel program communicates through *signals* that are either present or absent throughout each instant; this property is also referred to as the *synchrony hypothesis*. If a signal S is *emitted* in one instant, it is considered *present* from the beginning of that instant on. If a signal is not emitted in one instant, it is considered *absent*.

The Esterel language consists of a set of *primitive* statements, from which other statements are *derived* [3]. The primitives that directly involve signals are signal (signal declaration), emit (signal emission), present (conditional), and suspend (suspension).

Consider the three short Esterel programs shown in Figure 1. The first program NREACT involves the signal A, which is an input signal, meaning that it can be emitted in the environment, and also an output signal, meaning that it can be tested by the environment. Here the environment may be either the external environment of the program, or it may be other Esterel modules. The body of NREACT states that if A is present (emitted by the environment), then nothing is done, which is not problematic. However, if A is absent, then the else part is activated: A is emitted, which invalidates the former presence test for A. Such a contradiction is an invalid behavior of an Esterel program; such a program is over-constrained, or *not reactive*, and should be rejected by the compiler. This problem also becomes apparent when synthesizing this program into hardware, as the gate representation of this program is an inverter with it's output directly fed back to the input. This is obviously not a stable circuit and hence forbidden in Esterel.

The program NDET in Figure 1(b) is similar to NREACT, but with else changed to then. Here a present A will result in an emission of A in the then branch of the present statement, which would justify taking the then branch. Conversely, an absent A will skip the emission of A. Hence, this program is under-constrained, or *not deterministic*. A compiler should reject NDET. This also becomes apparent at the gate representation of NDET, which is a driver gate that transmits the input value to the output. Now the output is fed back to the input to map the behavior of the program. As a certain gate delay is inevitable, this circuit is an oscillator instead of providing stable outputs.

Programs NREACT and NDET have the same underlying problem: They involve a signal that is *self dependent*. In both programs the emission of A depends on a guard containing A. In these two examples, we have a *direct* self dependence, where the emission of a signal immediately depends on the presence of a signal. However, we may also have *indirect* self dependencies, in which a signal depends on itself via some other, intermediate signals. Consider program CYCLE in Figure 1(c), which contains two parallel threads, both testing for the signal emitted by the other one. However, the signals are emitted only if the other one has been emitted already; the emission of A depends on the presence of B and vice versa. In this case, we have a *cyclic dependency*, or *cycle* for short, and the program should again be rejected. We will refer to the emission of a signal that is guarded by a signal test (using present, suspend, or a derived statement) as a *guarded emit*.

All three programs shown in Figure 1 involve cyclic signal dependencies and are invalid, and hence of no further interest to us. However, there are programs that contain dependency cycles and yet are valid. A program is considered valid, or *constructive*, if we can establish the presence or absence of each signal without speculative reasoning, which may be possible even if the program contains cycles. The equivalent formulation in hardware is that there are circuits that contains cycles and yet are self-stabilizing, irrespective of delays [4].

Consider the program PAUSE_CYC in Figure 2(a): The cyclic dependency consists of an emission of B guarded by a test for A and an emission of A guarded by a test for B. At run time, however, the dependencies are

```
module PAUSE_CYC:
input A, B;
output C;

  present A then
    emit B
  end;
  pause;
  present B then
    emit A
  end
||
  present B then
    emit C
  end
end module
```

(a)

```
module PAUSE_PREP:
input A, B;
output C;

signal A_, B_, ST1, ST2 in
  emit ST1;
  present [A or A_] then
    emit B_
  end;
  pause;
  emit ST2;
  present [B or B_] then
    emit A_
  end
||
  present [B or B_] then
    emit C
  end
end signal
end module
```

(b)

```
module PAUSE_ACYC:
input A, B;
output C;

signal A_, B_, ST1, ST2 in
  emit ST1;
  present [A or
      (ST2 and (B or ST1))] then
    emit B_
  end;
  pause;
  emit ST2;
  present [B or B_] then
    emit A_
  end
||
  present [B or B_] then
    emit C
  end
end signal
end module
```

(c)

```
module PAUSE_OPT:
input A, B;
output C;

signal B_ in
  present A then
    emit B_
  end;
  pause;
  present B then
    emit A
  end
||
  present [B or B_] then
    emit C
  end
end signal
end module
```

(d)

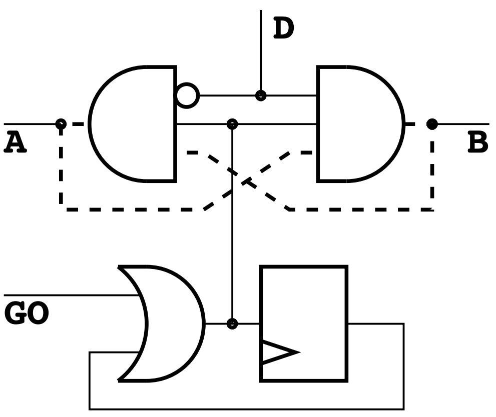
Figure 2: Resolving a false cycle.

```
module DRIVER_CYC:
input D;
inputoutput A, B;

loop
  present D then
    present A then
      emit B
    end
  else
    present B then
      emit A
    end
  end;
  pause
end
end module
```

(a)

```
module DRIVER_ACYC:
input D;
inputoutput A;
input B;
output B_;

loop
  present D then
    present A then
      emit B_
    end
  else
    present B then
      emit A
    end
  end;
  pause
end
end module
```

(b)



(c)

Figure 3: False cyclic dependencies in a bidirectional bus driver. The wires shown as dashed lines indicate the cyclic dependency.

separated by a pause statement into separate execution instants. The emission of B in the first instant has no effect on the test for B in the second instance.

In such a case, where not all dependencies are active in the same execution instant, we will call the cyclic dependency a *false cycle*. In contrast, the programs shown in Figure 1 all contained *true cycles*, where all dependencies involved were present at the same instant. A cycle may be false because it is broken by a register, as is the case in PAUSE_CYC, or because it is broken by a guard, as is the case in program DRIVER_CYC shown in Figure 3(a). Programs that only contain false cycles are still constructive and hence are valid programs that should be accepted by a compiler.

So far, we have considered only programs that contained true cycles and were invalid (NREACT, NDET, CYCLE) or that contained false cycles and were valid (PAUSE_CYC, DRIVER_CYC). However, there also exist programs that contain true cycles, with all dependencies evaluated at the same instant, and yet are valid programs. A classic example of a truly cyclic, yet constructive program is the Token Ring Arbiter [18]; Figure 4 shows a version with three stations. Each network station consists of two parallel threads: One computes the arbitration signals, the other passes a single token in each instant from one station to the next in each instant.

An inspection of the Arbiter reveals that there is a true cycle involving signals P1, P2, and P3. However, the program is still constructive as there is

```
module TR3_CYC:

input  R1, R2, R3;
output G1, G2, G3;

signal P1, P2, P3,
       T1, T2, T3
in
    emit T1
  ||
    loop   % STATION1
      present [T1 or P1]
      then
        present R1 then
          emit G1
        else
          emit P2
        end
      end ;
      pause
    end loop
  ||
    loop
      present T1 then
        pause;
        emit T2
      else
        pause
      end
    end
  ||
    loop   % STATION2
      present [T2 or P2]
      then
        present R2 then
          emit G2
        else
          emit P3
        end
      end ;
      pause
    end loop
  ||
    loop
      present T2 then
        pause;
        emit T3
      else
        pause
      end
    end
  ||
    loop   % STATION3
      present [T3 or P3]
      then
        present R3 then
          emit G3
        else
          emit P1
        end
      end ;
      pause
    end loop
  ||
    loop
      present T3 then
        pause;
        emit T1
      else
        pause
      end
    end
end module
```

Figure 4: Token Ring Arbiter with three stations.

always at least one token present which breaks the cycle. Hence, a compiler should accept this program. (However, the same program, but without the first thread that emits T1 in the first instant, should be rejected—this illustrates that determining constructiveness of a program is a non-trivial process.)

1.2 Related Work

A number of different approaches for compiling Esterel programs into either software or hardware have been proposed.

An early approach to synthesize software, employed by Berry *et al.*'s V3 compiler [5] and others [1, 8], builds an automaton through exhaustive simulation. This approach can compile cyclic programs. The resulting code is very fast, but potentially very large, as it is affected by possible state explosion.

Another approach, used by the v5 compiler [6], is to translate an Esterel program into a net-list, which can either be realized in hardware or which can be simulated in software. Using the technique proposed by Shiple *et al.* [19], this approach handles cycles by re-synthesizing cyclic portions into acyclic portions, employing the algorithm by Bourdoncle [7]. This approach offers better scalability than the automata-based approach, as it does not suffer from possible state explosion; however, the software variant tends to be rather slow, as it simulates the complete circuit during each instant, irrespective of which parts of the circuit are currently active.

A third approach to synthesize software is to generate an event-driven simulator, which breaks the simulated circuit into a number of small functions that are conditionally executed [10, 12, 9]. These compilers tend to produce code that is compact and yet almost as fast as automata-based code. The drawback of these techniques is that so far, they rely on the existence of a static schedule and hence are limited to acyclic programs. One approach to overcome this limitation, which has been suggested earlier by Berry and has been described by one of the authors [12], is to unroll the strongly connected regions of the Conditional Control Flow Graph; Esterel's constructive semantics guarantees that all unknown inputs to these strongly connected regions can be set to arbitrary, known values without changing the meaning of the program.

As it turns out, the transformation we are proposing here also makes use of this property of constructiveness to resolve cycles; however, unlike the approaches suggested earlier [13, 14], it does so at the source code level. Hence this makes it possible to compile originally cyclic programs using for example the existing efficient compilers that implement event-

Basics

$\mathbf{N}$: Set of natural numbers

For $n \in \mathbf{N} : \mathbf{N}_n =_{def} \{i \in \mathbf{N} \mid i < n\}$

P: Given Esterel Program

$\mathcal{S}$: Set of signals used in P

Guarded emits

$len \in \mathbf{N}_i$: Length of cycle

$Cycle = \{GEmit_i \mid i \in \mathbf{N}_{len}\}$

i: Guarded emit index, $i \in \mathbf{N}_{len}$

$GEmit_i = \langle GExp_i, GSig_i \rangle$: A guarded emit

$GExp_i$: Boolean expression involving signals $GIn \subseteq \mathcal{S}$

$GSig_i \in \mathcal{S}$: Signal emitted in guarded emit

$GSigs$: Set of original cycle signals

$GSig_{(i \bmod len)+1} \in GIn$: Cycle property

$GSig'_i$: A fresh signal used to replace emission of $GSig$ in $GEmit_i$

$GSigs'$: Set of fresh cycle signals

ST_i: A fresh state signal (used to indicate testing of guarded emit)

$STs = \{ST_i \mid i \in \mathbf{N}_{len}\}$: Set of state signals

Figure 5: Notation.

driven simulators. Furthermore, the experimental results indicate that this transformation can also improve the code resulting from the techniques that can already handle cyclic programs, such as the net-list approach employed by the V5 compiler. It also turns out that the compilation itself can be sped up by transforming cyclic programs into acyclic ones first.

2 The Basic Transformation Algorithm

Figure 5 introduces the notation we will use for our transformation. Figure 6 presents the algorithm for transforming cyclic Esterel programs into acyclic programs. The algorithm is applicable to programs with cycles that involve pure signals only.

The algorithm is introduced on the basis of the example PAUSE_CYC in

Input: Program P, potentially containing cycles
Output: Modified program P'', without cycles

1. Check constructiveness of P.
   If P is not constructive: **Error**.

2. Preprocessing of P:

   (a) If P is composed of several modules, instantiate them into one flat main module.

   (b) Expand derived statements that build on on the kernel statements emit/present/suspend.

   (c) Spread suspend blocks across enclosed statements.

   (d) Transform suspend into equivalent present/trap statements.

3. If P does not contain cycles: **Done**.
   Otherwise: Select a cycle $Cycle$, of length len.

4. Transform P into P'; for all $GEmit_i \in Cycle$:

   (a) Declare a new signal $GSig'_i$ in the same scope as $GSig_i$. If $GSig_i$ is an output signal in the module interface, then add $GSig'_i$ to the list of output signals instead.

   (b) Replace "emit $GSig_i$" by "emit $GSig'_i$".

   (c) Replace tests for $GSig_i$ by tests for "($GSig_i$ or $GSig'_i$)".

   (d) Declare a new signal ST_i in the scope enclosing the entire cycle.

   (e) Place "emit ST_i;" before the guarded emit.

5. Transform (still cyclic) P' into (acyclic) P'':

   (a) Select some cycle signal $GSig'_i \in GSigs'$.

   (b) Let $Cycle'$ be the cycle in P' that corresponds to $Cycle$ in P', (similarly for $GEmit'_i$, $GExp'_i$, GIn'_i). In all guarded emits in $Cycle'$, replace tests for $GSig'_i$ by $GExp^*_i$, where $GExp^*_i$ is an expression that does not involve any signals in $GSigs'$.

6. Goto Step 3, treat P'' now as P.

Figure 6: Transformation algorithm, for pure signals.

Figure 2(a), which is transformed into the acyclic program PAUSE_ACYC in Figure 2(c). The transformation of the program DRIVER_CYC in Figure 3(a) into DRIVER_ACYC in Figure 3(b) is similar. We also discuss for each transformation step the worst-case increase in code size.

Step 1: First we have to check the constructiveness. We can use one of the standard techniques, as for example proposed by Shiple *et al.* [19].

Step 2a: The expansion of modules is a straightforward textual replacement of module calls by their respective body. No dynamic runtime structures are needed, since Esterel does not allow recursions. Just the replacement of formal parameter names by their actual signals must be done. Since PAUSE_CYC does not contain any sub-modules, there is nothing to do in this step for PAUSE_CYC.

The complexity of this module expansion can reach exponential growth of code size, but this expansion is done by every Esterel compiler and not a special requirement of this transformation algorithm.

Step 2b: Regarding the statements handling signals, the transformation algorithm is expressed in terms of Esterel kernel statements. Therefore statements that are derived from emit, present, or suspend must be reduced to these statements. PAUSE_CYC consists entirely of kernel statements, therefore no replacement is needed.

One derived statement is replaced by a fixed construct of kernel statements, therefore the complexity of this step is a constant factor on the number of statements in the program.

Steps 2c and 2d: An example program with suspend statements is discussed in Section 4.1 on page 17.

Each statement inside a suspend statement is enclosed in a block made up of suspend/present statements of constant size. Therefore the complexity is also a constant factor on the number of statements in the program.

Step 3: If there is more than one cycle present in the program, then the application of the algorithm is repeated for each cycle. PAUSE_CYC contains only one cycle. $\{\langle \mathsf{A}, \mathsf{B}\rangle, \langle \mathsf{B}, \mathsf{A}\rangle\}$.

Step 4a and 4b: To prepare the removal of the cycle, we first transform PAUSE_CYC in Step 4 into the equivalent program PAUSE_PREP, shown in Figure 2(b). It differs from PAUSE_CYC in that the signals carrying the cycle (A and B) have been replaced by fresh signals (A_ and B_) which are only emitted within the cycle. In a way, this introduction of fresh signals, which are emitted exclusively in the cycle, is akin to Static Single Assignment (SSA) [11].

For each signal in the program, at most one replacement signal is added, thus complexity of this task is a constant factor of the program size.

Step 4c: All tests for A and B in the original program are replaced by tests for [A or A_] and [B or B_], respectively. Using the SSA analogy, this corresponds to a ϕ-node.

Each changed signal test is expanded by an expression of constant size, therefore we get a constant factor on the number of signal test expressions in the program.

Step 4d: The declarations of ST1 and ST2 are simply added to the declarations of A_ and B_.

The number of added signals is likewise of linear complexity of the program size.

Step 4e: PAUSE_PREP contains the fresh state signals ST1 and ST2, which indicate the evaluation of a guarded emit. This makes the activity of the guard expression itself available for another present test.

The number of added signal emissions is limited by the number of emit statements in the program, and thus of linear complexity.

Step 5a: One signal in the set of cyclic signals must be selected as a point to break the cyclic dependency. Basically any signal in the cycle will work; the actual selection can be based on the smallest replacement expression computed for the next step. In PAUSE_PREP, we select A_ as the signal to break the cycle.

Step 5b: Finally we are ready to break the cycle in PAUSE_PREP. For that, we have to replace the signal selected in Step 5a —in the cycle—by an expression that does not use any of the cycle signals, without changing the meaning of the program.

To compute the replacement expression for A_ in PAUSE_PREP, we note that A_ is present iff (if and only if) ST2 is present and B or B_ is present; expressed as an equation, it is:

$$\mathsf{A\_} = \mathsf{ST2} \wedge (\mathsf{B} \vee \mathsf{B\_}). \tag{1}$$

This equation now refers to another cycle signal, B_; note that we consider B not a cycle signal anymore, as it is not emitted within the cycle anymore. To replace B_ in Equation 1, we observe:

$$\mathsf{B\_} = \mathsf{ST1} \wedge (\mathsf{A} \vee \mathsf{A\_}). \tag{2}$$

Substituting 2 into 1 yields:

$$\mathsf{A\_} = \mathsf{ST2} \wedge (\mathsf{B} \vee (\mathsf{ST1} \wedge (\mathsf{A} \vee \mathsf{A\_}))). \tag{3}$$

This is now an equation which expresses the cycle signal A_ as a function of itself and other signals that are not part of the cycle; so we have unrolled the cycle. We could now simulate this using three-valued logic; however, we can also make use of the constructiveness of the program, which guarantees monotonicity. This means that a more defined input always produces an equal or more defined output. Hence, if the program is known to never produce undefined outputs, we can set all unknown inputs (such as $\mathsf{A\_}$ in this case) to arbitrary, known values without changing the meaning of the program [12]. Applying this to Equation 3 yields, for $\mathsf{A\_} = \mathit{false}$ (absent):

$$\mathsf{A\_} = \mathsf{ST2} \land (\mathsf{B} \lor (\mathsf{ST1} \land \mathsf{A})). \tag{4}$$

Similarly, for $\mathsf{A\_} = \mathit{true}$ (present):

$$\mathsf{A\_} = \mathsf{ST2} \land (\mathsf{B} \lor \mathsf{ST1}). \tag{5}$$

We now have derived two equally valid replacement expressions for $\mathsf{A\_}$, which do not involve any cycle signal. Substituting 5, the simpler of these expressions, for $\mathsf{A\_}$ in PAUSE_PREP yields the now acyclic program PAUSE_ACYC shown in Figure 2(c).

The complexity of the replacement expressions depends on the length of the cycle, because the length of the cycle dictates the number of replacement iterations needed to eliminate all but the first cycle signals in the guard expression. The length of the cycle and the size of each replacement are limited by the number of signals in the program. So there is a quadratic dependency of the size of the replacement expression to the program size. The number of times the replacement expression will be inserted in the program is likewise dependent on the program size. Thus the growth in program size for one cycle is of cubic complexity.

Step 6: The transformation algorithm must be repeated for each cycle and the upper limit of cycles to resolve is the number of signals in the program.

Overall, a very conservative estimate results in a code size of $\mathcal{O}(n^4)$, where n is the source program size after module expansion; however, we expect the typical code size increase to be much lower. In fact, we often experience an actual reduction in source size, as the transformation often offers optimization opportunities where statements are removed. As for the size of the generated object code, here the experimental results (Section 5) also demonstrate that typically the transformation results in a code size reduction.

Application to the Token Ring Arbiter Before transforming program TR3_CYC from Figure 4 into the acyclic TR3_ACYC shown in Figure 7, we can apply some optimizations. We first note that the guarded emits that constitute the cycle involving the cycle signals $\mathit{GSigs} = \{\mathsf{P1}, \mathsf{P2}, \mathsf{P3}\}$ are always executed. Hence we do not need to introduce fresh state signals.

```
module TR3_ACYC:

input  R1, R2, R3;
output G1, G2, G3;

signal P2, P3,
       % P1 deleted
       T1, T2, T3
in
   emit T1
  ||
   loop  % STATION1
     present [T1 or
        (not R3 and (T3
        or (not R2 and
           (T2 or not R1))))
      ] then
       present R1 then
         emit G1
       else
         emit P2
       end
     end ;
     pause
   end loop
  ||
   loop
     present T1 then
       pause;
       emit T2
     else
       pause
     end
   end
  ||
   loop   % STATION2
     present [T2 or P2]
     then
       present R2 then
         emit G2
       else
         emit P3
       end
     end ;
     pause
   end loop
  ||
   loop
     present T2 then
       pause;
       emit T3
     else
       pause
     end
   end
  ||
   loop   % STATION3
     present [T3 or P3]
     then
       present R3 then
         emit G3
       % else branch
       % deleted
       end
     end;
     pause
   end loop
  ||
   loop
     present T3 then
       pause;
       emit T1
     else
       pause
     end
   end
end module
```

Figure 7: Non cyclic Token Ring Arbiter.

Furthermore, there are no tests for the cycle signals outside of the cycle, so we do not need fresh cycle signals either. These optimizations are explained further in Section 3.

We now select signal P1 to break the cycle. We can compute the expression to replace P1 in the test in STATION1 as follows:

$$\begin{aligned} \mathsf{P2} &= \overline{\mathsf{R1}} \wedge (\mathsf{T1} \vee \mathsf{P1}), && (6)\\ \mathsf{P3} &= \overline{\mathsf{R2}} \wedge (\mathsf{T2} \vee \mathsf{P2}) \\ &= \overline{\mathsf{R2}} \wedge (\mathsf{T2} \vee (\overline{\mathsf{R1}} \wedge (\mathsf{T1} \vee \mathsf{P1}))), && (7)\\ \mathsf{P1} &= \overline{\mathsf{R3}} \wedge (\mathsf{T3} \vee \mathsf{P3}) \\ &= \overline{\mathsf{R3}} \wedge (\mathsf{T3} \vee (\overline{\mathsf{R2}} \wedge (\mathsf{T2} \vee (\overline{\mathsf{R1}} \wedge (\mathsf{T1} \vee \mathsf{P1}))))). && (8) \end{aligned}$$

Equation 8 now again expresses a cycle carrying signal (P1) as a function of itself and other signals that are outside of the cycle. Again we can employ the constructiveness of TR3_CYC to replace P1 in this replacement expression by either *true* or *false*. Setting P1 to *false* yields:

$$\mathsf{P1} = \overline{\mathsf{R3}} \wedge (\mathsf{T3} \vee (\overline{\mathsf{R2}} \wedge (\mathsf{T2} \vee (\overline{\mathsf{R1}} \wedge \mathsf{T1})))). \quad (9)$$

Setting P1 to *true* yields:

$$\mathsf{P1} = \overline{\mathsf{R3}} \wedge (\mathsf{T3} \vee (\overline{\mathsf{R2}} \wedge (\mathsf{T2} \vee \overline{\mathsf{R1}}))). \quad (10)$$

This is also the replacement expression applied when transforming TR3_CYC. The other transformation steps are fairly straightforward, and offer further optimization opportunities, as also discussed in the next section.

3 Optimizations

The application of the algorithm in Figure 6 exposes opportunities for further optimizations. For example, the program PAUSE_ACYC can be optimized into the program PAUSE_OPT shown in Figure 2(d).

Replacing state signal tests by constants The replacement expression $GExp_i^*$ (Step 5b of the algorithm) may reference some state signal $ST_j \in STs$ that can be shown to be always present or absent:

1. If $GExp_i^*$ replaces $GSig_i'$ in $GExp_j$, we know that at this location in the program, ST_j must always be present. Therefore, we can replace ST_j by the constant *true* in $GExp_i^*$.

   In the program PAUSE_ACYC, this applies to the state signal ST1 in the replacement expression "(ST2 and (B or ST1))," which we therefore can simplify to "(ST2 and B)."

2. More generally, we can replace a state signal by the constant *true* whenever we know that it must be emitted in every instant.

   This applies for example to the Token Ring Arbiter, where we know that all guarded emits that constitute the cycle are evaluated in every instant of the program.

3. Correspondingly, it may also be the case that a state signal is always absent when tested in some replacement expression $GExp_i^*$. In particular, this is the case when we have a false cycle.

   In the program PAUSE_ACYC, this applies to the state signal ST2; due to the pause statement between the evaluation of the replacement expression and the emission of ST2, we can set ST2 to $false$ in the replacement expression. In this case, this reduces the whole replacement expression to $false$; therefore, the "[A or (ST2 and (B or ST1))]" from PAUSE_ACYC gets reduced to just "A" in PAUSE_OPT.

Eliminating emission of state signals If all tests for a state signal are replaced by constants, the state signal is no longer needed and therefore does not need to be emitted any more.

In the program PAUSE_ACYC, this applies to both ST1 and ST2, we can therefore drop the corresponding emit in the optimized PAUSE_OPT.

Absence of External Emissions of Cycle Signals If a cycle signal $GSig_i$ is not emitted outside of the cycle, we do not need to generate a fresh signal $GSig'_i$, but can instead just use $GSig_i$. In this case, one may skip Steps 4a, 4b, and 4c.

This is the case in the Arbiter, where the signals carrying the cycle (P1/P2/P3) are not emitted outside of the cycle.

Absence of External Tests of Cycle Breaking Signal If the signal $GSig'_i$ that is selected in Step 5a to break the cycle is not tested outside of the cycle, this means that after replacing the tests for $GSig'_i$ within the cycle (Step 5b) by $GExp_i^*$, the signal $GSig'_i$ is not tested anywhere in the program. One can therefore eliminate its emission.

This also applies to the Arbiter, where signal P1, which we replaced within the cycle, becomes superfluous. We can therefore eliminate the emit P1, and the enclosing else branch.

Simplification of External Tests Depending on how often one must replace a particular signal $GSig_i$ in Step 4c by the expression "($GSig_i$ or

```
module PAUSES:
inputoutput A, B, C, D;

signal T1, T2 in
  loop
    pause; pause; pause;
    present [A or T1] then
      emit B;
      emit C
    end
  end
||
  loop
    pause; pause; pause;
    pause; pause;
    present [B or T2] then
      emit A;
      emit D
    end
  end
||
  loop
    emit T1;
    pause;
    emit T2;
    pause
  end
end module
```

(a)

```
module PAUSES_ACYC:
inputoutput A, B, C, D;
output B_;

signal T1, T2, ST1 in
  loop
    pause; pause; pause;
    emit ST1;
    present [A or T1] then
      emit B_;
      emit C
    end
  end
||
  loop
    pause; pause; pause;
    pause; pause;
    present [(B or ST1)
        or T2] then
      emit A;
      emit D
    end
  end
||
  loop
    emit T1;
    pause;
    emit T2;
    pause
  end
end module
```

(b)

Figure 8: Example requiring extra state signals.

$GSig_i')$", it may be beneficial to introduce another fresh signal $GSig_i''$. This signal must be emitted whenever $GSig_i$ or $GSig_i'$ are present, for example using a new globally parallel statement of the form "every [$GSig_i$ or $GSig_i'$] do emit $GSig_i''$ end". Then it suffices to replace tests for $GSig_i$ by tests for $GSig_i''$.

4 Further Example Transformations

Figure 8 gives another program with a cycle involving signals A and B. However, the cycle is only present at certain instants; the guarded emit of B takes place every 3rd instant, whereas the guarded emit of A takes place every 5th instant, hence the cycle is active only every 15th instants. Here the transformation into an acyclic version requires the emission of the signal ST1, which indicates the evaluation of the guard dependency with

```
module SUSP_CYC:
output A,B;

  pause;
  suspend
    emit A
  when immediate B
||
  suspend
    emit B
  when immediate A
end module
```

(a)

```
module SUSP_ACYC:
output A,B;
output B_;

signal ST1 in
    pause;
    suspend
      emit A;
    when immediate ST1
  ||
    trap T in
      loop
        emit ST1;
        present [not A] then
          emit B_;
          exit T
        end;
        pause;
      end loop
    end trap
end signal
end module
```

(b)

Figure 9: Simple cyclic program with suspend.

sink B. As an optimization, due to the invariant "T1 **or** T2 = **true**", which is ensured by the third parallel thread, it suffices to just add an "**or** ST1" to the guard in the guard dependency involving A in the transformed program.

4.1 Suspend

So far we presented only cycles with a present test as a guard for an emit statement. Another way to influence the execution of emit is the suspend statement. A complication with suspend is that, unlike with present, one cannot easily generate a signal that is emitted unconditionally whenever the guard of a suspend is evaluated. The transformation algorithm therefore first transforms the suspend statements into equivalent present/trap statements, in Steps 2c and 2d.

As an example, consider the program SUSP_CYC in Figure 9(a). The program again contains a cyclic dependency on the signals A and B, the emission of each signal is inhibited by the presence of the other signal.

Applying Steps 2c and 2d involves a chain of transformations not shown here; the end result, after applying the whole transformation algorithm, is SUSP_ACYC in Figure 9(b).

```
module VALUE_CYC:

input  S;
input  A : integer , B : integer ;
output X : integer , Y : integer ;

present S then
  present A then
    emit B(?A)
  end
 else
  present B then
    emit A(?B)
  end
end;

present A then emit X(?A) end;
present B then emit Y(?B) end
end module
```

(a)

```
module VALUE_ACYC:

input  S;
input  A : integer , B : integer ;
output X : integer , Y : integer ;

signal B_:integer, B__:integer in
    present S then
      present A then
        emit B_(?A)
      end
    else
      present B then
        emit A(?B)
      end
    end;

    present A then emit X(?A) end;
    present B__ then emit Y(?B__) end
  ||
    every B  do emit B__(?B) end
  ||
    every B_ do emit B__(?B_) end
  end signal
end module
```

(b)

Figure 10: Esterel program with a cycle on valued signals.

4.2 Valued Signals

Figure 10(a) contains an Esterel program VALUE_CYC with a (false) cycle on the signals A and B. Both signals carry values of type integer. The pure signal S is used to select one of two data flows: From A to B or vice versa. This results in two guarded emits with reversed signal use. Therefore both guarded emits build a cycle. The cycle is *false*, because signal S ensures that only one of both emits is active in an instant. After the guarded emits are evaluated, two additional guarded emits copy the values from A and B to the outputs X and Y, respectively.

Figure 10(b) contains the program VALUE_ACYC, an acyclic transformation of VALUE_CYC. Two additional signals are introduced: B_ and B__. B_ is the replacement for B to break the cycle, and B__ is used to represent the state of B outside the cycle. Two additional parallel statements forward the values of B (from the module interface) and B_ (from the cycle) to the signal B__.

There are two emit statements for the same signal B__. Both are executed in the same instant if B is emitted on the module interface and B is emitted in the cycle. Therefore we need a *combine* function if we can not exclude this case. But this problem is not introduced by the transformation of the cycle. The original VALUE_CYC contains needs a combine function, too. Consider S, A and B being present at the module interface. Then the first guarded emit will be executed and B will be emitted in the same instant a second time. The same holds for S absent, then A will be emitted twice. We can compile the program without a combine function if we assert that A and B are not both present in the same instant. This resolves the problem for VALUE_ACYC, too.

5 Experimental Results

The proposed algorithm is currently not implemented in a ready-to-use compiler. However some preliminary benchmarks can be obtained by applying the algorithm manually to some small cyclic Esterel programs. The examples used here are:

TR3: This is the Token Ring Arbiter with three network stations. The implementation is as in Figure 4.

TR10: This is an extension of tr3 from three to ten network stations. The aim is to test the scaling of the algorithm for code size and runtime.

TR10p: While the former test cases implemented only the arbiter part of the network without any local activity on the network stations, this test program adds some simple concurrent "payload" activity to each network station to simulate a CPU performing some computations with occasional access to the network bus.

VALUE: The program VALUE in Figure 10 is included to test the application of the algorithm to valued signals.

All programs are tested in the originally cyclic and in the transformed acyclic version.

5.1 Synthesizing Software

To evaluate the transformation in the realm of generating software, we used six different compilation techniques:

v5-L: The Esterel compiler v5.92 [6] is publicly available [15]. It is used in this case with option -L to produce code based on the circuit representation of Esterel. The code is organized as a list of equations ordered by dependencies. This results in a fairly compact code, but with a comparatively slow execution speed. This compiler is able to handle constructive Esterel programs with cyclic dependencies.

v5-A: The same compiler, but with the option -A, produces code based on a flat automaton. This code is very fast, but prohibitively big for programs with many weakly synchronized parallel activities. This option is available for cyclic programs, too.

v7: The Esterel v7 compiler (available at Esterel Technologies) is used here in version v7_10i8 to compile acyclic code based on sorted equations, like the v5 compiler.

v7-O: The former compiler, but with option -O, applies some circuit optimizations to reduce program size and runtime.

CEC: The Columbia Esterel Compiler (available with source code [9]) produces event driven C code, which is fast with small code size. However, this compiler cannot handle cyclic dependencies. Thus it can only be applied to the transformed cyclic programs.

CEC-g: The CEC with -g produces code using computed goto targets to reduce the runtime even further. This feature is an extension to ANSI-C offered by GCC-3.3 [16].

A simple C back-end is provided for each Esterel program to produce input signals and accept output signals to and from the Esterel part. The back-end iterates over the first three token ring examples 10,000,000 times and 30,000,000 times for the last (simpler) valued signal example. These iteration counts result in handy execution times in the range of about 0.8 to 18 seconds. These times where obtained on a desktop PC (AMD Athlon XP 2400+, 2.0 GHz).

Table 1(a) compares the execution speed of the example programs for the v5, v7, and CEC compilers with their respective options. The v5 compiler is applied both to the original cyclic programs and the transformed acyclic programs. The CEC and v7 compiler can handle only acyclic code.

When comparing the runtime results of the v5 compiler (with sorted equations) for the cyclic and acyclic versions of the token ring arbiter, there is a noticeable reduction in runtime for the transformed acyclic programs. This came as a bit of a surprise. It seems that the v5 compiler is a little bit less efficient in resolving cyclic dependencies in sorted equations. For the automaton code there are only minor differences in runtime. The acyclic version of the VALUE program is less efficient in runtime using the v5 compiler. The reason for this lies in the additional parallel activities introduced by the transformation algorithm.

For the two token ring arbiter variants without payload, the v7 compiler produces the fastest code. The third token ring example with payload is executed fastest with the CEC compiler, but only slightly better than the v5 compiler in automata mode. Again the the optimized sorted equation code of v7 fits bests to the VALUE example.

Table 1(b) compares the fastest code for our cyclic programs to the fastest code for the transformed acyclic programs. For each test program the relative reduction in runtime is listed.

Table 2 lists the sizes of the compiled binaries. All compilers produce code of similar sizes, but with one exception: The v5 compiler produces a very big automaton code for the third token ring example. That program contains several parallel threads which are only loosely related. If someone tries to map such a program on a flat automaton, it is well known that such a structure results in a "state explosion". Actually, we had to limit the number of parallel tasks in this example to get the program to compile in reasonable time.

Table 3 contains the compilation times for the different Esterel compilers to compile the various test programs. The v5 compiler for sorted equations code needs only little time to compile the acyclic versions of the test programs. In fact, it is among the fastest compilers in all four acyclic test cases. When this compiler is applied to cyclic programs, the compilation times are several times slower but within reasonable limits. When compiling for automaton code with the v5 compiler, then the compilation time is mostly independent of cyclic and acyclic properties of the compiled program. The compilation times are low for small programs with few states, but drastically higher for programs with many independent, parallel states. The CEC compiler is comparatively slow for small acyclic programs, but the compilation time does not rise that much for more complex programs. The v7 compiler behaves similarly.

5.2 Synthesizing Hardware

To evaluate the effect of our transformation on hardware synthesis, we compared again the results of the v5, v7, and CEC compilers, for the same set of benchmarks as for the software synthesis. Again only v5 can handle the untransformed, cyclic code version; furthermore, v5 is the only compiler that can generate hardware for valued signals. The compilers differ in which hardware description languages they can produce, but a common format supported by all of them is the Berkeley Logic Interchange Format (BLIF), therefore we base our comparisons on this output format.

Table 4 compares the number of nodes synthesized. Considering the v5 compiler, there is a noticeable reduction in the number of nodes generated for the Arbiter. However, for the VALUE benchmark the number of nodes increases, again due to the extra parallel threads added for joining the valued signals. When considering the synthesis results of v7 and CEC for the acyclic version of the Arbiter, v7 produces the best overall results, with the node count less than half of v5's synthesis results for the cyclic variants.

Table 5 compares the number of literals generated. The overall results

Variant	Compiler	TR3	TR10	TR10p	VALUE
cyclic	v5-L	1.60	5.43	17.26	5.43
(original)	v5-A	0.90	2.60	**5.28**	8.01
	v5-L	1.35	4.75	11.75	5.83
acyclic	v5-A	0.90	2.59	**5.28**	8.73
(trans-	v7	1.75	6.05	13.18	6.74
formed)	v7-O	**0.49**	**1.97**	6.02	**4.65**
	CEC	1.22	5.40	11.23	6.25
	CEC-g	0.83	3.06	**5.25**	5.32

(a)

	TR3	TR10	TR10p	VALUE
$min(T_{cyclic})$	0.90	2.60	5.28	5.43
$min(T_{acyclic})$	0.49	1.97	5.25	4.65
reduction	46%	24%	0.6%	14%

(b)

Table 1: (a) Run times (in seconds) of cyclic and acyclic Esterel programs compiled with the v5, v7, and CEC compiler. (b) Relative runtime reduction from the fastest cyclic version to the fastest version for the acyclic transformation, with $reduction = 100\% * (1 - min(T_{acyclic})/min(T_{cyclic}))$.

Variant	Compiler	TR3	TR10	TR10p	VALUE
cyclic	v5-L	14271	21530	32244	12906
(original)	v5-A	**13039**	**16091**	304095	13358
	v5-L	13937	19772	28566	13330
acyclic	v5-A	**13041**	**16093**	304097	13510
(trans-	v7	14460	19916	27033	14315
formed)	v7-O	13449	16314	**21217**	13836
	CEC	13506	19764	27315	13089
	CEC-g	13164	18302	23341	**12787**

Table 2: Size of compiled Esterel programs (in bytes) using the v5, v7, and CEC compiler.

Variant	Compiler	TR3	TR10	TR10p	VALUE
cyclic (original)	v5-L	0.09	0.29	1.42	0.04
	v5-A	**0.02**	0.06	11.07	0.03
acyclic (trans-formed)	v5-L	0.03	**0.04**	**0.08**	**0.01**
	v5-A	**0.02**	**0.04**	10.43	0.02
	v7	0.09	0.20	0.29	0.07
	v7-O	0.14	0.42	1.02	0.13
	CEC	0.11	0.23	0.56	0.11
	CEC-g	0.14	0.25	0.54	0.09

Table 3: Run times of the Esterel v5, v7, and CEC compilers (in seconds).

Variant	Compiler	TR3	TR10	TR10p	VALUE
cyclic	v5	107	338	686	**116**
acyclic	v5	68	295	643	164
	v7	**48**	**160**	**298**	-
	CEC	106	344	614	-

Table 4: Comparison of node count for BLIF output.

Variant	Compiler	TR3	TR10	TR10p	VALUE
cyclic	v5	205	735	1499	**1007**
acyclic	v5	155	591	1281	1792
	v7	**97**	**328**	**628**	-
	CEC	167	559	1117	-

Table 5: Comparison of sum-of-product (lits(sop)) count for BLIF output.

are similar to the ones for the node count; the transformation has increased the literal count for VALUE, but has been lowered significantly for the arbiter.

6 Conclusions and future work

We have presented an algorithm for transforming cyclic Esterel programs into acyclic programs. This expands the range of available compilation techniques, and, as to be expected, some of the techniques that are restricted to acyclic programs produce faster and/or smaller code than is possible with the compilers that can handle cyclic codes as well. Furthermore, the experiments showed that the code transformation proposed here can even improve code quality produced by the same compiler.

We have presented the transformation for Esterel programs; however, as mentioned in the introduction, this transformation should also be applicable to other synchronous languages, such as Lustre. Lustre is also a synchronous language, but data-flow oriented, as opposed to the control-oriented nature of Esterel. To our knowledge, none of the compilers available for Lustre can handle cyclic programs, even though valid cyclic programs (such as the Token Ring Arbiter) can be expressed in the language. Hence in the case of Lustre, applying the source-level transformation proposed here is not only a question of efficiency, but a question of translatability in the first place.

Regarding future work, the transformation algorithm spells out only how to handle cycles carried by pure signals. We have presented an example for removing a cycle involving a valued signal, but this still has to be generalized.

Furthermore, as it stands, programmers can apply the algorithm presented here to remove cycles from a program just manually, perhaps guided by information from the compiler regarding the exact location of cycles. However, to be truly usable, the transformation should be performed automatically. Therefore, we plan to implement this transformation as a preprocessing step within the CEC.

Acknowledgment

We thank Xin Li for conducting the hardware synthesis experiments.

References

[1] Balarin, F., Chiodo, M., Giusto, P., Hsieh, H., Jurecska, A., Lavagno, L., Sangiovanni-Vincentelli, A., Sentovich, E. M., and Suzuki, K. Sythesis of Software Programs for Embedded Control Applications. In *IEEE Transactions of Computer-Aided Design of Integrated Circuits and System* (June 1999), vol. 18, pp. 834–849.

[2] Benveniste, A., Caspi, P., Edwards, S. A., Halbwachs, N., Guernic, P. L., and de Simone, R. The Synchronous Languages Twelve Years Later. In *Proceedings of the IEEE, Special Issue on Embedded Systems* (Jan. 2003), vol. 91, pp. 64–83.

[3] Berry, G. *The Constructive Semantics of Pure Esterel.* Draft Book, 1999.

[4] Berry, G. The foundations of Esterel. *Proof, Language and Interaction: Essays in Honour of Robin Milner* (2000). Editors: G. Plotkin, C. Stirling and M. Tofte.

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

[6] Berry, G., and the Esterel Team. *The Esterel v5_91 System Manual.* INRIA, June 2000.

[7] Bourdoncle, F. Efficient chaotic iteration strategies with widenings. In *Formal Methods in Programming and Their Applications: International Conference Proceedings* (June 1993), vol. 735 of *Lecture Notes in Computer Science*, Springer.

[8] Castelluccia, C., Dabbous, W., and O'Malley, S. Generating efficient protocol code from an abstract specification. *IEEE/ACM Transactions on Networking 5*, 4 (1997), 514–524.

[9] CEC: The Columbia Esterel Compiler. `http://www1.cs.columbia.edu/~sedwards/cec/`.

[10] Closse, E., Poize, M., Pulou, J., Venier, P., and Weil, D. SAXO-RT: Interpreting Esterel semantic on a sequential execution structure. In *Electronic Notes in Theoretical Computer Science* (July 2002), F. Maraninchi, A. Girault, and E. Rutten, Eds., vol. 65, Elsevier.

[11] Cytron, R., Ferrante, J., Rosen, B. K., Wegman, M. N., and Zadeck, F. K. Efficiently computing static single assignment form and the control dependence graph. *ACM Transactions on Programming Languages and Systems 13*, 4 (October 1991), 451–490.

[12] Edwards, S. A. An Esterel compiler for large control-dominated systems. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems 21*, 2 (Feb. 2002).

[13] Edwards, S. A. Making Cyclic Circuits Acyclic. In *Proceedings of the 40th conference on Design automation* (June 2003).

[14] Edwards, S. A., and Lee, E. A. The Semantics and Execution of a Synchronous Block-Diagram Language. In *Science of Computer Programming* (July 2003), vol. 48, Elsevier.

[15] Esterel web. `http://www-sop.inria.fr/esterel.org/`.

[16] The GNU compiler collection. `http://gcc.gnu.org/`.

[17] Halbwachs, N., Caspi, P., Raymond, P., and Pilaud, D. The synchronous data-flow programming language LUSTRE. *Proceedings of the IEEE 79*, 9 (September 1991), 1305–1320.

[18] Pandya, P. The saga of synchronous bus arbiter: On model checking quantitative timing properties of synchronous programs. In *Electronic Notes in Theoretical Computer Science* (2002), F. Maraninchi, A. Girault, and Éric Rutten, Eds., vol. 65, Elsevier.

[19] Shiple, T. R., Berry, G., and Toutati, H. Constructive Analysis of Cyclic Circuits. In *Proc. International Design and Test Conference ITDC 98, Paris, France* (Mar. 1996).