CORE
Metadata, citation and similar papers at core.ac.uk
Provided by Dagstuhl Research Online Publication Server

Removing Cycles in Esterel Programs

Jan Lukoschus[1], Reinhard von Hanxleden[2]

[1] Christian-Albrechts-Universität zu Kiel, Institut für Informatik
24098, Kiel, Olshausenstr. 40, Germany
jlu@informatik.uni-kiel.de
[2] Christian-Albrechts-Universität zu Kiel, Institut für Informatik
24098, Kiel, Olshausenstr. 40, Germany
rvh@informatik.uni-kiel.de

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.

Keywords. Synchronous Languages, Compilation, Cyclic Circuits, Constructiveness, Esterel, Lustre, Hardware, Software

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 approaches 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].

Dagstuhl Seminar Proceedings 04491
Synchronous Programming - SYNCHRON'04
http://drops.dagstuhl.de/opus/volltexte/2005/160

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. Section 3 describes how to derive replacement expression for signals from an Esterel program. Optimization options are presented in Section 4. Section 5 demonstrates the transformation with further examples, including a program using *valued signals*. Section 6 provides experimental results, the paper concludes in Section 7.

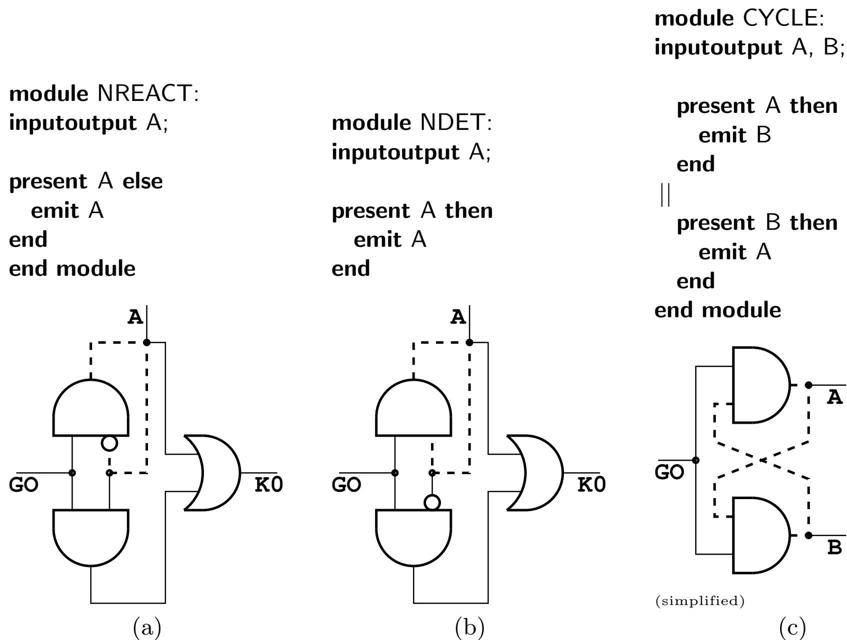

Fig. 1. Invalid cyclic Esterel programs. The wires shown as dashed lines indicate the cyclic dependencies.

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 consist of 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 its 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 may be 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].

```
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_, ST_0, ST_1, ST_2 in
  emit ST_0;
  [
    present [A or A_] then
      emit B_
    end;
    pause; emit ST_1;
    present [B or B_] then
      emit A_
    end
  ||
    present [B or B_] then
      emit C
    end
  ]; emit ST_2
  end signal
end module
```

(b)

```
module PAUSE_ACYC:
input A, B;
output C;
signal A_, B_, ST_0, ST_1, ST_2 in
  [
    emit ST_0;
    present [A or
        (ST_1 and (B or ST_0))] then
      emit B_
    end;
    pause; emit ST_1;
    present [B or B_] then
      emit A_
    end
  ||
    present [B or B_] then
      emit C
    end
  ] ; emit ST_2
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)

Fig. 2. Resolving a false cycle.

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 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.

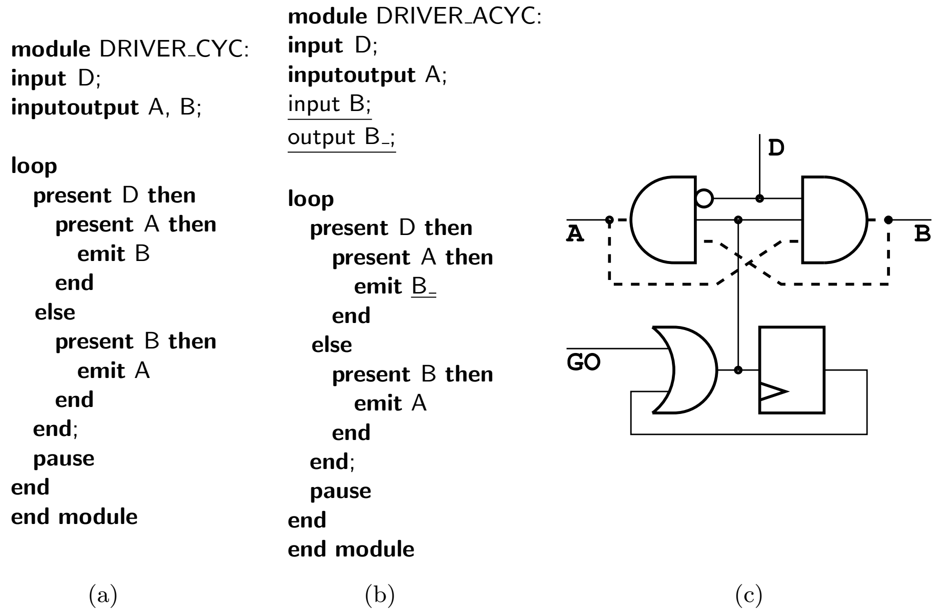

Fig. 3. False cyclic dependencies in a bidirectional bus driver. The wires shown as dashed lines indicate the cyclic dependency.

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 always at least one token present that breaks the cycle. Hence, a compiler should accept this program. Note that 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.

```
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
```

Fig. 4. Token Ring Arbiter with three stations.

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.* [20], 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 in [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-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 following section will discuss each transformation step along with it's worst-case increase in code size.

Step (1): The constructiveness of the Esterel program is a crucial precondition for this algorithm to work. This analysis can be done by using the v5 compiler [15] or by implementation of published algorithms [20,3].

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

$len \in \mathbf{N}$: Length of cycle
$Cycle = \{GEmit_i \mid i \in \mathbf{N}_{len}\}$
i: Guarded emit index, $i \in \mathbf{N}_{len}$
$GEmit_i = \langle GSig_i, GExp_i \rangle$: 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

Fig. 5. Notation.

Step (2): The core algorithm is only applicable to Esterel programs restricted in certain ways. Therefore we perform some preprocessing to simplify the structure of the program:

Step (2.2a): 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.

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 (2.2b): 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.

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.

Step (2.2c): We have to eliminate locally defined signals because replacement expressions for signals computed by our algorithm could carry references to local signals out of their scope. (Note that the programmer may still freely use local signal declarations.) Furthermore, our method of finding replacement expressions assumes that signals are unique, i. e., not re-incarnated. A simple approach to eliminate re-incarnation is based on loop-unrolling, which results in a potentially exponential increase in code size; using other techniques, this can

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 the kernel statements
   (c) Rename locally defined signals to make them unique and lift the definitions up to the top level. Furthermore, eliminate signal re-incarnation.
   (d) Transform suspend into equivalent present/trap statements.
   (e) Rename trap names making them unique
3. If P does not contain cycles: **Done**.
   Otherwise: Select a cycle $Cycle$, of length len.
4. Introduce state signals:
   (a) Add boot register:
      – Globally declare a new signal ST_0
      – Add "emit ST_0;" to the start of the program body.
   (b) Enumerate all pause and || statements starting from 1 and do for all pause_i and $||_i$:
      – Globally declare a new signal ST_i.
      – Replace "pause_i" by "pause; emit ST_i."
      – Replace "p $||_i$ q" by "[p || q]; emit ST_i."
5. Transform P into P'; for all $GSig_i \in Cycle$:
   (a) If $GSig_i$ is an output signal in the module interface, then add $GSig'_i$ to the list of output signals; otherwise, globally declare a new signal $GSig'_i$.
   (b) Replace "emit $GSig_i$" by "emit $GSig'_i$."
   (c) Replace tests for $GSig_i$ by tests for "($GSig_i$ or $GSig'_i$)."
   Let $GSigs'$ be the union of all $GSig'_i$.
6. Transform (still cyclic) P' into (acyclic) P'':
   (a) For all $GSig'_i \in GSigs'$ determine replacement expressions $Expr_i$.
   (b) Select some cycle signal $GSig'_i \in GSigs'$.
   (c) Iteratively transform $Expr_i$ to $Expr^*_i$ by replacement of all signals $GSig'_j \in (GSigs' \setminus GSig'_i)$ by their expressions $Expr_j$.
   (d) Replace $GSig'_i$ in $Expr^*_i$ by false (or true) and minimize result.
   Now $Expr^*_i$ does not involve any cyclic signals.
   (e) Replace all tests for $GSig'_i$ in P' by $Expr^*_i$.
7. Goto Step (3), treat P'' now as P.

Fig. 6. Transformation algorithm, for pure signals.

be reduced to a quadratic increase [3] or more efficient by the introduction of a "gotopause" statement into Esterel [21].

Step (2.2d): Program fragments of the form

```
suspend p when S
```

where p denotes the suspended body and S the suspension condition, are replaced by just the body p, where all "pause" statements inside p are replaced by "await not S." This transformation emulates the behavior of "suspend" by explicitly checking the suspension condition at the start of each instant. However, as the await statement is a derived statement, we have to transform it further into kernel statements; "await not S." then becomes:

```
trap T in
  loop
    pause;
    present S else exit T end
  end
end
```

An example program with suspend statements is discussed in Section 5.1 on page 20.

The complexity of this transformation is proportional to the number of "pause" statements inside "suspend" statements.

Step (2.2e): Now there may be multiple instances of the same trap name T. This constitutes a valid Esterel program, however, it simplifies the subsequent transformation to have unique trap names.

Step (3): Cycles in the program are identified by building a graph representing the control flow dependencies between "present" tests and signal emissions. That directed graph can be used to search for cyclic dependencies in the Esterel program. Only the signals that are part of the cycle are of further interest.

If there is more than one cycle present in the program, then the application of the algorithm is repeated for each cycle.

Steps (4): The introduction of state signals makes the current state of the program available to signal expressions. Each pause statement is supplemented with the emission of a unique signal ST_i. The first state signal ST_0 is emitted at program start. ST_0 resembles the boot register in the circuit representation of Esterel programs. Additionally all parallel operators (||) are extended by the emission of a state signal at termination.

The number of additional state signals and signal emissions is proportional to the number of pause statements in the program and therefore proportional to program size.

Steps (5.5a/5b): This step splits each cycle signal $GSig_i$ into two signals $GSig_i$ and $GSig'_i$. The signal with the original name $GSig_i$ is emitted outside the cycle, a signal with a new name $GSig'_i$ is emitted as part of the cycle. The motivation of this step is, to be able to distinguish between emissions from inside and outside the cycle and the aim of the replacement expression is to replace emissions inside 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 the complexity of this step is a constant factor of the program size.

Step (5.5c): All tests for cycle signals in the original program are extended by tests for their replacement signals, respectively. Using the SSA analogy, this corresponds to a ϕ-node [11].

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 (6.6a): The computation of replacement expressions is described in detail in Section 3 on page 16.

Step (6.6b): 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 in the next step.

Step (6.6c): $Expr_i$ contains references to other cycle signals $GSig'_j$. These are recursively replaced by their respective expressions $Expr_j$ into $Expr^*_i$. This unfolding of expressions is performed until only $GSig'_i$ besides other non cyclic signals is contained in $GExp^*_i$.

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 (6.6d): Since the program is known to be constructive, it follows that Sig'_i in Exp^*_i must not have any influence on the evaluation of Exp^*_i. Therefore we can replace Sig'_i in Exp^*_i by any constant value (true or false). Now Exp^*_i contains only non cyclic signals. It is important to replace just all Sig'_i and not any occurrence of Sig_i, because Sig_i is emitted outside the cycle therefore not part of the cycle.

The true or false values must be used to minimize the expression, because some Esterel compilers do not support those boolean constants.

Step (6.6e): The last transformation step in the algorithm replaces every occurrence of Sig'_i in present tests by it's replacement expression Exp^*_i. Now we have replaced one signal of the cycle by an expression which is not part of the cycle. Therefore we have broken the current cycle $Cycle$.

Step (7): 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 and elimination of signal re-incarnations; 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 6) also demonstrate that typically the transformation results in a code size reduction.

Application to example The algorithm is applied to the example PAUSE_CYC in Figure 2(a) on page 4, which is transformed into the acyclic program PAUSE_ACYC in Figure 2(c). The transformation of the program DRIVER_CYC in Figure 3(a), page 5, into DRIVER_ACYC in Figure 3(b) is similar.

Step (1): PAUSE_CYC is cyclic but nevertheless constructive, because a pause statement separates the execution of both parts of the cycle.

Steps (2.2a) to (2.2e) do not apply to PAUSE_CYC.

Step (3): PAUSE_CYC contains one cycle. $Cycle = \{\langle \mathsf{A}, \mathsf{B}\rangle, \langle \mathsf{B}, \mathsf{A}\rangle\}$.

Steps (4) and (5): To prepare the removal of the cycle, we first transform PAUSE_CYC into the equivalent program PAUSE_PREP, shown in Figure 2(b). It differs from PAUSE_CYC in the introduction of state signals ST_0 to ST_2 and 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. All tests for A and B in the original program are replaced by tests for [A or A_] and [B or B_], respectively.

Step (6.6a): The computation of replacement expressions for A_ and B_ according to Section 3 results in:

$$\mathsf{A_} = \mathsf{ST_1} \wedge (\mathsf{B} \vee \mathsf{B_}) \tag{1}$$

$$\mathsf{B_} = \mathsf{ST_0} \wedge (\mathsf{A} \vee \mathsf{A_}) \tag{2}$$

The equations for each signal now refer to other cycle signals; note that we consider A and B not cycle signals anymore, as they are not emitted within the cycle anymore.

Step (6.6b): In PAUSE_PREP, we arbitrarily select A_ as the signal to break the cycle.

Step (6.6c): To replace B_ in Equation (1), we get by substituting (2) into (1):

$$\mathsf{A_} = \mathsf{ST_1} \wedge (\mathsf{B} \vee (\mathsf{ST_0} \wedge (\mathsf{A} \vee \mathsf{A_}))). \tag{3}$$

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.

Step (6.6d): We could now simulate (3) using three-valued logic; however, here we 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 A_ in this case) to arbitrary, known values without changing the meaning of the program [12]. Applying this to Equation (3) yields, for A_ = false (absent):

$$\mathsf{A_} = \mathsf{ST_1} \wedge (\mathsf{B} \vee (\mathsf{ST_0} \wedge \mathsf{A})). \tag{4}$$

Similarly, for A_ = true (present):

$$\mathsf{A_} = \mathsf{ST_1} \wedge (\mathsf{B} \vee \mathsf{ST_0}). \tag{5}$$

We now have derived two equally valid replacement expressions for A_, which do not involve any cycle signal.

Step (6.6e): Finally we are ready to break the cycle in PAUSE_PREP. For that, we have to replace the signal selected in Step (6b) —in the cycle—by an expression that does not use any of the cycle signals, without changing the meaning of the program.

Substituting (5), the simpler of these expressions, for A_ in PAUSE_PREP yields the now acyclic program PAUSE_ACYC shown in Figure 2(c).

```
module TR3_ACYC:

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

signal ST_0, ST_1, ST_2, ST_3,
       ST_4, ST_5, ST_6, ST_7,
       ST_8, ST_9, ST_10 in
  emit ST_0;
signal P2, P3,
      % P1 deleted
       T1, T2, T3
in
[
    emit T1
  ||
    loop  % STATION1
      present
[T1 or (ST_0 or ST_7)
  and (T3 or (ST_0 or ST_4)
    and (T2 or (ST_0 or ST_1)
      and not R1)
    and not R2)
  and not R3] then
        present R1 then
          emit G1
        else
          emit P2
        end
      end;
      pause; emit ST_1;
    end loop
  ||
    loop
      present T1 then
        pause; emit ST_2;
        emit T2
      else
        pause; emit ST_3;
      end
    end
  ||
    loop   % STATION2
      present [T2 or P2]
      then
        present R2 then
          emit G2
        else
          emit P3
        end
      end;
      pause; emit ST_4
    end loop
  ||
    loop
      present T2 then
        pause; emit ST_5;
        emit T3
      else
        pause; emit ST_6
      end
    end
  ||
    loop   % STATION3
      present [T3 or P3]
      then
        present R3 then
          emit G3
        % else branch
        % deleted
        end
      end;
      pause; emit ST_7
    end loop
  ||
    loop
      present T3 then
        pause; emit ST_8;
        emit T1
      else
        pause; emit ST_9
      end
    end
 ]; emit ST_10
end module
```

Fig. 7. Non cyclic Token Ring Arbiter.

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 an optimization. There are no tests for the cycle signals outside of the cycle, so we do not need fresh cycle signals either. This and other optimizations are explained further in Section 4.

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

$$\mathsf{P2} = (\mathsf{ST_0} \vee \mathsf{ST_1}) \wedge (\mathsf{T1} \vee \mathsf{P1}) \wedge \overline{\mathsf{R1}}, \tag{6}$$

$$\begin{aligned}\mathsf{P3} &= (\mathsf{ST_0} \vee \mathsf{ST_4}) \wedge (\mathsf{T2} \vee \mathsf{P2}) \wedge \overline{\mathsf{R2}} \\ &= (\mathsf{ST_0} \vee \mathsf{ST_4}) \wedge (\mathsf{T2} \vee (\mathsf{ST_0} \vee \mathsf{ST_1}) \wedge (\mathsf{T1} \vee \mathsf{P1}) \wedge \overline{\mathsf{R1}}) \wedge \overline{\mathsf{R2}},\end{aligned} \tag{7}$$

$$\begin{aligned}\mathsf{P1} &= (\mathsf{ST_0} \vee \mathsf{ST_7}) \wedge (\mathsf{T3} \vee \mathsf{P3}) \wedge \overline{\mathsf{R3}} \\ &= (\mathsf{ST_0} \vee \mathsf{ST_7}) \wedge (\mathsf{T3} \vee (\mathsf{ST_0} \vee \mathsf{ST_4}) \wedge (\mathsf{T2} \vee (\mathsf{ST_0} \vee \mathsf{ST_1}) \\ &\qquad \wedge (\mathsf{T1} \vee \mathsf{P1}) \wedge \overline{\mathsf{R1}}) \wedge \overline{\mathsf{R2}}) \wedge \overline{\mathsf{R3}}.\end{aligned} \tag{8}$$

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:

$$\mathsf{P1} = (\mathsf{ST_0} \vee \mathsf{ST_7}) \wedge (\mathsf{T3} \vee (\mathsf{ST_0} \vee \mathsf{ST_4}) \wedge (\mathsf{T2} \vee (\mathsf{ST_0} \vee \mathsf{ST_1}) \wedge \mathsf{T1} \wedge \overline{\mathsf{R1}}) \wedge \overline{\mathsf{R2}}) \wedge \overline{\mathsf{R3}}. \tag{9}$$

Setting P1 to true yields:

$$\mathsf{P1} = (\mathsf{ST_0} \vee \mathsf{ST_7}) \wedge (\mathsf{T3} \vee (\mathsf{ST_0} \vee \mathsf{ST_4}) \wedge (\mathsf{T2} \vee (\mathsf{ST_0} \vee \mathsf{ST_1}) \wedge \overline{\mathsf{R1}}) \wedge \overline{\mathsf{R2}}) \wedge \overline{\mathsf{R3}}. \tag{10}$$

This is also the replacement expression applied when transforming TR3_CYC. The other transformation steps are fairly straightforward.

The replacement expression is fairly complex, but close inspection yields an optimization: The expression "$(\mathsf{ST_0} \vee \mathsf{ST_7})$" is contained in 10: The state signal ST_0 is emitted in the first instant and ST_7 is emitted in all instants but the first one. In a disjunction they will always return true. Therefore the expression can be replaced statically by true. The same holds for "$(\mathsf{ST_0} \vee \mathsf{ST_4})$" and "$(\mathsf{ST_0} \vee \mathsf{ST_1})$."

With this optimization 10 can be reduced to:

$$\mathsf{P1} = (\mathsf{T3} \vee (\mathsf{T2} \vee \overline{\mathsf{R1}}) \wedge \overline{\mathsf{R2}}) \wedge \overline{\mathsf{R3}}. \tag{11}$$

Further optimization opportunities are discussed in Section 4.

emit S:

$$\mathcal{R}((\mathsf{!S}), C) = \{(\mathsf{S}, C)\} \tag{12}$$
$$\mathcal{C}((\mathsf{!S}), C) = C$$

present S then p else q end:

$$\mathcal{R}((\mathsf{S?p,q}), C) = \mathcal{R}(\mathsf{p}, C \wedge S) \cup \mathcal{R}(\mathsf{q}, C \wedge \overline{S}) \tag{13}$$
$$\mathcal{C}((\mathsf{S?p,q}), C) = \mathcal{C}(p, C \wedge S) \vee \mathcal{C}(q, C \wedge \overline{S})$$

nothing:

$$\mathcal{R}((\mathsf{0}), C) = \emptyset \tag{14}$$
$$\mathcal{C}((\mathsf{0}), C) = C$$

pause; emit ST_i:

$$\mathcal{R}((\mathsf{1;!ST_i}), C) = \emptyset \tag{15}$$
$$\mathcal{C}((\mathsf{1;!ST_i}), C) = \mathsf{ST_i}$$

exit T:

$$\mathcal{R}((\mathsf{k}), C) = \{(\mathsf{T}, C)\} \tag{16}$$
$$\mathcal{C}((\mathsf{k}), C) = \mathsf{false}$$

trap T in p end:

$$\mathcal{R}((\{\mathsf{p}\}), C) = \mathcal{R}(\mathsf{p}, C) \tag{17}$$
$$\mathcal{C}((\{\mathsf{p}\}), C) = \mathcal{C}(\mathsf{p}, C) \vee \bigvee_{(\mathsf{T}, c) \in \mathcal{R}(\mathsf{p}, C)} c_i$$

p;q:

$$\mathcal{R}((\mathsf{p;q}), C) = \mathcal{R}(\mathsf{p}, C) \cup \mathcal{R}(\mathsf{q}, \mathcal{C}(\mathsf{p}, C)) \tag{18}$$
$$\mathcal{C}((\mathsf{p;q}), C) = \mathcal{C}(\mathsf{q}, \mathcal{C}(\mathsf{p}, C))$$

loop p end:

$$\mathcal{R}((\mathsf{p*}), C) = \mathcal{R}(\mathsf{p}, C \vee \mathcal{C}(p, C)) \tag{19}$$
$$\mathcal{C}((\mathsf{p*}), C) = \mathsf{false}$$

[p II q]; emit ST_i:

$$\mathcal{R}(((\mathsf{p|q});\mathsf{!ST_i}), C) = \mathcal{R}(\mathsf{p}, C) \cup \mathcal{R}(\mathsf{q}, C) \tag{20}$$
$$\mathcal{C}(((\mathsf{p|q});\mathsf{!ST_i}), C) = \mathsf{ST_i}$$

signal S in p end:

$$\mathcal{R}((\mathsf{p\backslash S}), C) = \mathcal{R}(\mathsf{p}, C) \tag{21}$$
$$\mathcal{C}((\mathsf{p\backslash S}), C) = \mathcal{C}(\mathsf{p}, C)$$

Fig. 8. Equations to determine replacement expressions for signals

```
present I1 then
  present I2 else
    emit A
  end
end
||
present A then
  emit B
end
```

(a)

```
present I1 then
  present I2 else
    emit A
  end
end
||
present [I1 and not I2] then
  emit B
end
```

(b)

Fig. 9. Replacing the signal test for A by their emission context.

3 Computing the replacement expressions

One step towards breaking cyclic dependencies in Esterel programs is to replace within the conditions of present tests the name of a certain signal by an expression (Step (6.6a) of the algorithm). That expression is derived from the control flow contexts of the program where the signal is set by emit statements.

The Logical Behavioral Semantics rules [3] serve as a base to derive the control flow context for a given Esterel program. Our rules are direct derivations from those rules with the aim of an easy implementation.

The main objective of the rules is to get replacement expressions for all signals. The replacement expression describes the signal context of each emission for that signal. Therefore as a prerequisite the signal context reaching emit statements is needed.

The rules to implement both tasks operate on an Esterel Program P and two sets:

- C : Current signal context expression;
- R : Accumulation of replacement expressions for signals.

The rules are implemented in two functions:

- $\mathcal{R}$: $P \times C \to R$
  This function returns a mapping of signal names to their signal contexts at the point of their emission.
- $\mathcal{C}$: $P \times C \to C$
  $\mathcal{C}$ takes the signal context delivered by previous statements and computes the signal context for sub statements and returns the signal context for following statements.

These functions are computed by structural induction over their first argument (an Esterel program); the corresponding definitions for each kernel statement are given in Figure 8. To determine the replacement expressions for all

signals in a program P, we compute $R := \mathcal{R}(P, \mathsf{false})$, where the false indicates that at the top-level, the signal context is empty. The result of $\mathcal{R}$ will be a set of pairs. Each pair consists of a signal name and a signal expression (condition). The expressions describe for their associated signal in which signal context it is emitted. Now the expressions for the same respective signals can be conjuncted to yield a single replacement expression for the emission of each signal.

As an example to illustrate how the definitions of $\mathcal{R}$ and $\mathcal{C}$ correspond to the behavioral semantics, consider the present statement.

$$\frac{s^+ \in E \qquad p \xrightarrow[E]{E',k} p'}{s?p,q \xrightarrow[E]{E',k} p'}$$

(a) present+

$$\frac{s^- \in E \qquad q \xrightarrow[E]{E',k} q'}{s?p,q \xrightarrow[E]{E',k} q'}$$

(b) present-

Fig. 10. Logical Behavioral Semantics of the present statement

The two SOS rules from the Logical Behavioral Semantics for the present statement, given in Figure 10 [3], select the rule to apply based on the presence of the condition signal s and the resulting control flow. The selected rule will add signal emissions etc. to the resulting context. The corresponding equations for $\mathcal{R}$ and $\mathcal{C}$(13) consider both possible control flow paths, and both paths may add signal emissions to R; however, each signal emission is tied to the condition for that part, thus reflecting the original semantics.

4 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) on page 4.

Replacing state signal tests by constants The replacement expression $GExp_i^*$ (Step (6.6e) 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 $GExp_k$, and we know that at this location in the program, ST_j must always be present, then we can replace ST_j by the constant true in $GExp_i^*$.
   In the program PAUSE_ACYC, this applies to the state signal ST_0 in the replacement expression "(ST_1 and (B or ST_0))," which we therefore can simplify to "(ST_1 and B)."
   More generally, we could replace a state signal by the constant true if we knew that it must be emitted in every instant. As it turns out, there cannot be any state signals that fulfill this condition by themselves; the boot state

signal ST_0 is only present in the initial instant, and all other state signals are emitted only after a pause statement, and hence cannot be emitted in the initial instant. However, another optimization is possible:

2. If a state signal is emitted in every instant except for the initial instant, we can replace it with pre(tick).
   In the Arbiter, for example, the state signals ST_1, ST_4, and ST_7 are the only state signals emitted in a loop that runs concurrently to the rest of the program—hence, as loops must not be instantaneous, they must be emitted at every iteration and are present at every instant except the initial one.
3. Tests for "ST_0 or pre(tick)" can be replaced by true.
   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. Hence this rule, together with the previous rule, leads to the simplified replacement expression already stated in Equation 11.
4. 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 ST_1; due to the pause statement between the evaluation of the replacement expression and the emission of ST_1, we can set ST_1 to false in the replacement expression. In this case, this reduces the whole replacement expression to false; therefore, the "[A or (ST_1 and (B or ST_0))]" 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 ST_0 and ST_1, 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 Step (5).

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 (6b) 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 6e) 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 (5c) by the expression "($GSig_i$ or $GSig'_i$)," 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 $GSig''_i$.

(a)

```
module SUSP_CYC:
output A,B;

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

(b)

```
module SUSP_PREP:
output A,B;

signal ST_0, ST_1,
    ST_2, ST_3, ST_4 in
  emit ST_0;
  [
    pause; emit ST_1;
    trap T1 in
      loop
        pause; emit ST_2;
        present B else
          exit T1
        end
      end;
    end;
    emit A
  ||
    trap T2 in
      loop
        pause; emit ST_3;
        present A else
          exit T2
        end
      end;
    end;
    emit B
  ]; emit ST_4
end signal
end module
```

(c)

```
module SUSP_ACYC:
output A,B;

signal ST_0, ST_1,
    ST_2, ST_3, ST_4 in
  emit ST_0;
  [
    pause; emit ST_1;
    trap T1 in
      loop
        pause; emit ST_2;
        present B else
          exit T1
        end
      end;
    end;
    emit A
  ||
    trap T2 in
      loop
        pause; emit ST_3;
        present
          [ST_2 and not ST_3]
        else
          exit T2
        end
      end;
    end;
    emit B
  ]; emit ST_4
end signal
end module
```

Fig. 11. Simple cyclic program with suspend.

5 Further Example Transformations

5.1 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 Step (2.2d).

As an example, consider the program SUSP_CYC in Figure 11(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 (2.2d), (4.4b) results in the preprocessed program SUSP_PREP shown in Figure 11(b). The end result, after applying the whole transformation algorithm, is SUSP_ACYC in Figure 11(c).

```
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)

Fig. 12. Esterel program with a cycle on valued signals.

5.2 Valued Signals

Figure 12(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 constitute 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 12(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 cannot exclude this case. However, 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.

6 Experimental Results

The proposed algorithm has been implemented in its basic form, so far without the optimizations mentioned in Section 4 and without support for valued signals, as an extension of the Columbia Esterel Compiler (CEC). For a first experimental evaluation, we have defined several variants of the Token Ring Arbiter:

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.

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

6.1 Synthesizing Software

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

v5-L: The publicly available Esterel compiler v5.92 [6,15]. 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 (an extension to ANSI-C offered by GCC-3.3 [16]) to reduce the runtime even further.

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.

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.

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(a) 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 2(b) 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.

Variant	Compiler	TR3	TR10	TR10p
cyclic (original)	v5-L	1.58	5.45	17.19
	v5-A	0.91	2.59	**5.28**
acyclic (transformed)	v5-L	1.47	5.16	12.20
	v5-A	0.92	2.59	**5.27**
	v7	1.72	6.04	12.43
	v7-O	**0.44**	**1.87**	6.02
	CEC	1.96	7.92	12.58
	CEC-g	1.03	3.71	5.49

(a)

	TR3	TR10	TR10p
$min(T_{cyclic})$	0.91	2.59	5.28
$min(T_{acyclic})$	0.44	1.87	5.27
reduction	52%	28%	0.2%

(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 $reduction = 100\% * (1 - min(T_{acyclic})/min(T_{cyclic}))$.

As an indication of the cost of the transformation algorithm in terms of processing time and source code increase, Table 3 lists transformation times and program sizes before and after the transformation of the token ring arbiter with 3, 10, 50, and 100 nodes. The size of the transformed code is nearly proportional with respect to the arbiter network size. The current run times show a sub-quadratic effort for the transformation. It should also be noted that at this

Variant	Compiler	TR3	TR10	TR10p
cyclic	v5-L	14273	21530	32244
(original)	v5-A	**13041**	**16091**	304095
	v5-L	14083	20220	29142
acyclic	v5-A	**13043**	**16093**	304097
(trans-	v7	14526	20271	27369
formed)	v7-O	13435	16315	**21121**
	CEC	14276	21924	28683
	CEC-g	13630	19710	24565

(a)

Variant	Compiler	TR3	TR10	TR10p
cyclic	v5-L	0.09	0.27	1.37
(original)	v5-A	**0.02**	**0.05**	10.85
	v5-L	0.03	0.07	**0.07**
acyclic	v5-A	0.04	**0.05**	10.42
(trans-	v7	0.10	0.19	0.37
formed)	v7-O	0.19	0.54	1.06
	CEC	0.15	0.30	0.69
	CEC-g	0.14	0.28	0.68

(b)

Table 2. (a) Size of compiled Esterel programs (in bytes) using the v5, v7, and CEC compiler. (b) Run times of the Esterel v5, v7, and CEC compilers (in seconds).

Transformation	TR3	TR10	TR50	TR100
original size	1565	3705	16348	32159
module expansion	1370	4391	22031	44092
cycle transformation	2033	6526	32765	65778
transformation time	0.62	2.3	19.2	63.5

Table 3. Transformation times (in seconds) and resulting program sizes (in bytes) for token ring arbiters with 3 to 100 nodes.

point the run times are dominated by lots of system I/O for debugging output. In a release version the transformation should be much faster.

6.2 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(a) compares the number of nodes synthesized. Considering the v5 compiler, there is a noticeable reduction in the number of nodes generated for the Arbiter. 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 4(b) compares the number of latches needed by the synthesization results. Here the CEC is able to reduce the number of latches considerably.

Table 4(c) compares the number of literals generated. The overall results are similar to the ones for the node count; the transformation has been lowered the literal count for the arbiter.

Variant	Compiler	TR3	TR10	TR10p
cyclic	v5	112	357	759
acyclic	v5	108	346	748
	v7	**52**	**171**	**351**
	CEC	146	468	756

(a)

Variant	Compiler	TR3	TR10	TR10p
cyclic	v5	10	31	55
acyclic	v5	10	31	55
	v7	10	31	55
	CEC	**4**	**11**	**47**

(b)

Variant	Compiler	TR3	TR10	TR10p
cyclic	v5	208	745	1551
acyclic	v5	197	645	1377
	v7	**108**	**360**	**702**
	CEC	221	725	1301

(c)

Variant	Compiler	TR3	TR10	TR10p
cyclic	v5	**82**	**266**	539
acyclic	v5	89	299	**524**
	v7	91	315	591
	CEC	89	313	679

(d)

Table 4. Comparison of: (a) node count for BLIF output, (b) latch count for BLIF output. (c) sum-of-product (lits(sop)) count for BLIF output. (d) sum-of-product (lits(sop)) - optimized by SIS

Table 4(c) compares the number of literals which remain after a SIS [19] optimization.

7 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. There are also numerous optimizations possible, some of which presented in Section 4, which we plan to implement and evaluate; in particular, we are interested in the extent to which these optimizations might be helpful for Esterel programs in general, not just as a post-processing step to the transformation proposed here.

Finally, as we have observed earlier, the concept of constructiveness is a fundamental building block for the transformation presented here; constructiveness allows us to ultimately break a cycle by replacing the occurrence of a self-dependent signal in a replacement expression for that signal by an arbitrary value (true or false). However, if we would like to determine in the first place whether a program is constructive or not, the transformation proposed here might be employed to accelerate this analysis; by replacing signal occurrences by expressions as computed by the algorithm (including possible self-references), one may replace a generally computationally expensive iterative procedure, which is a classical approach to analyze constructiveness, by a more efficient analysis.

Acknowledgments

We would like to thank Stephen Edwards for several fruitful discussions about different types of cyclic dependencies and for providing his CEC compiler as a solid foundation to build upon.

Klaus Schneider was the first to note that this transformation might also be helpful for constructiveness analysis, as after the transformation there would be no need to perform fix-point iterations.

Finally, 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. Sentovich, E. M., Singh, K. J., Lavagno, L., Moon, C., Murgai, R., Saldanha, A., Savoj, H., Stephan, P. R., Brayton, R. K., and Sangiovanni-Vincentelli, A. SIS: A System for Sequential Circuit Synthesis. Tech. Rep. UCB/ERL M92/41, University of California at Berkeley, May 1992.
20. 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).
21. Tardieu, O. Goto and Concurrency - Introducing Safe Jumps in Esterel. In *Proceedings of Synchronous Languages, Applications, and Programming, Barcelona, Spain* (Mar. 2004).