Removing Cycles in Esterel Programs

Jan Lukoschus[1] Reinhard von Hanxleden[2]

Dept. of Computer Science and Applied Mathematics
Christian-Albrechts-Universität zu Kiel
Olshausenstr. 40, D-24098 Kiel, Germany

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 efficiency 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.

Key words: Synchronous Languages, Compilation, Cyclic Circuits, Constructiveness, Esterel, Lustre, Hardware, Software

---

1 Introduction

One of the strengths of synchronous languages [1] 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,

---

[1] Email: `jlu@informatik.uni-kiel.de`
[2] Email: `rvh@informatik.uni-kiel.de`

This is a preliminary version. The final version will be published in
Electronic Notes in Theoretical Computer Science
URL: `www.elsevier.nl/locate/entcs`

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 [4]; however, the concepts introduced here should be applicable to other synchronous languages as well, such as Lustre [15].

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. The application of the transformation algorithm is exemplified on two different types of cyclic dependencies in Section 4. Optimization options are presented in Section 5. Section 6 provides experimental results, the paper concludes in Section 7.

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* [2]. The primitives that directly involve signals are signal (signal declaration), emit (signal emission), present (conditional), and suspend (suspension).

The emission of signals can be conditionally executed depending on tests for the presence of other signals. This establishes *dependency relations* between signals. A closed circle of such dependency relations in an Esterel program is called a *dependency cycle*. Such a cycle is problematic, because the evaluation of a condition must not be invalidated by following signal emissions. If that occurs the program is invalid and must be rejected.

A simple example for such an invalid program is:

```
present A else emit A end
```

When A is not present at the signal test then the else part is executed and A is emitted which invalidates the former signal test.

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 [3].

Consider the program PAUSE_CYC in Figure 1(a): the cyclic dependency consists of an emission of B guarded by a test for A and an emission of A

```
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)

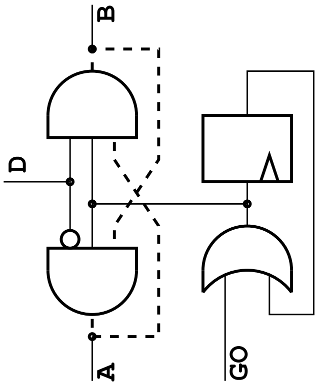
Fig. 1. Resolving a false cycle.

```
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
```

```
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
```



(a) (b) (c)

Fig. 2. False cyclic dependencies in a bidirectional bus driver. The wires shown as dashed lines indicate the cyclic dependency.

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.

In such a case, where not all dependencies are active in the same execution instant, we will call the cyclic dependency a *false cycle*. 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 2(a). Programs that only contain false cycles are still constructive and hence are valid programs that should be accepted by a compiler.

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 3 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. 3. 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.

The approach used by the v5 compiler [5] 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 [6]. 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.

Another approach implemented in the v5 compiler builds an automaton through exhaustive simulation of the netlist code. The resulting code is very fast, but potentially very large, as it is affected by possible state explosion.

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 [8,10,7]. 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 [10], 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 [11,12], 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.

The compilation of Esterel can not only be complicated by cyclic dependencies, but also by *signal reincarnation*, also known as *schizophrenia* [2]. An efficient cure for schizophrenia in Esterel has been proposed by Tardieu [21]. Since it is also based on source code transformation and orthogonal to our approach to handle cycles, it is suited to be sequentially applied to Esterel programs together with the transformation proposed here.

2 The Basic Transformation Algorithm

Figure 4 introduces the notation we will use for our transformation. Figure 5 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 (i): The constructiveness of the Esterel program is a precondition for our transformation. This analysis can be performed using the methods developed by Shiple et al. and by Berry [20,2]; one available implementation is offered by the v5 compiler [13].

Conversely, the fact that our transformation is valid if and only if the transformed program is constructive can be exploited to employ this transformation to aid in constructiveness analysis in the first place; see also the comments on Step (vi.d) on page 10.

Step (ii): The core algorithm is only applicable to Esterel programs restricted in certain ways, requiring the following preprocessing steps:

Step (ii.a): The expansion of modules is a straightforward textual replacement of module calls by their respective body. No dynamic runtime

Basics

$\mathbf{N}$: Set of natural numbers
For $n \in \mathbf{N} : \mathbf{N}_n =_{def} \{i \in \mathbf{N} \mid i \leq n\}$
P: Given Esterel Program
Σ: Set of signals used in P

Cycle Terminology

$\mathcal{E}_i = \langle \sigma_i, \mathcal{G}_i \rangle$, with $i \in \mathbf{N}$: Guarded emit
$\mathcal{G}_i$: Boolean expression ("Guard") involving signals $\Sigma_i \subseteq \Sigma$
$\sigma_i \in \Sigma$: Signal emitted in guarded emit
$\mathcal{C} = \{\mathcal{E}_i \mid i \in \mathbf{N}_l\}$, with $l \in \mathbf{N}$: Cycle of length l
$\sigma_{(i \bmod l)+1} \in \Sigma_i$: Cycle property

Signals

$\Sigma_{\mathcal{C}}$: Set of original cycle signals
σ'_i: Fresh signal used to replace emission of σ in $\mathcal{E}_i$
$\Sigma'_{\mathcal{C}}$: Set of fresh cycle signals
ST_i: Fresh state signal (used to indicate testing of guarded emit)
$ST = \{ST_i \mid i \in \mathbf{N} \cup \{0\}\}$: Set of state signals

Fig. 4. Notation.

structures are needed, since Esterel does not allow recursions.

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 (ii.b): 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 (ii.c): 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 reincarnation is based on loop-unrolling, which results in a potentially exponential increase in code size; using other techniques, this can be reduced to a quadratic increase [2], or even lower complexity by the introduction of a `gotopause` statement [21].

Step (ii.d): Program fragments of the form

```
suspend p when S
```

where `p` denotes the suspended body and `S` the suspension condition, are re-

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

- (i) Check constructiveness of P. If P is not constructive: **Error**.
- (ii) 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 reincarnation.
  - (d) Transform `suspend` into equivalent `present`/`trap` statements.
  - (e) Rename trap names making them unique.
- (iii) 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 $\mathsf{ST\_}i$.
    - Replace pause_i by "$\mathsf{pause;\ emit\ ST\_}i$."
    - Replace "$\mathsf{p}\ \|_i\ \mathsf{q}$" by "$\mathsf{[p\ \|\ q];\ emit\ ST\_}i$."
- (iv) If P does not contain cycles: **Done**.
  Otherwise: Select a cycle $\mathcal{C}$, of length l.
- (v) Transform P into P'; do for all $\sigma_i \in \mathcal{C}$:
  - (a) If σ_i is an output signal in the module interface, then add σ'_i to the list of output signals; otherwise, globally declare a new signal σ'_i.
  - (b) Replace "$\mathsf{emit}\ \sigma_i$" by "$\mathsf{emit}\ \sigma'_i$."
  - (c) Replace tests for σ_i by tests for "$(\sigma_i\ \mathsf{or}\ \sigma'_i)$."
- (vi) Transform (still cyclic) P' into (acyclic) P'':
  - (a) For all $\sigma'_i \in \Sigma'_\mathcal{C}$ determine replacement expressions $\mathcal{R}_i$.
  - (b) Select some cycle signal $\sigma'_i \in \Sigma'_\mathcal{C}$.
  - (c) Iteratively transform $\mathcal{R}_i$ to $\mathcal{R}^*_i$ by replacement of all signals $\sigma'_j \in (\Sigma'_\mathcal{C} \setminus \sigma'_i)$ by their expressions $\mathcal{R}_j$.
  - (d) Replace σ'_i in $\mathcal{R}^*_i$ by `false` (or `true`) and minimize result.
    Now $\mathcal{R}^*_i$ does not involve any cyclic signals.
  - (e) Replace all tests for σ'_i in P' by $\mathcal{R}^*_i$.
- (vii) Goto Step (iv), treat P'' now as P.

Fig. 5. Transformation algorithm, for pure signals.

placed 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
```

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

Step (ii.e): 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.

Steps (iii): The introduction of state signals makes the current state of the program available to signal expressions. (Note that many of the signals may be eliminated again by subsequent optimizations, see Section 5.) 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 and parallel operators in the program and therefore proportional to the size of the program.

Step (iv): 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 Steps (v) through (vi) are performed for each cycle.

Steps (v.a/b): This step splits each cycle signal σ_i into two signals σ_i and σ'_i. The signal with the original name σ_i is emitted outside the cycle, a signal with a new name σ'_i is emitted as part of the cycle. The motivation of this step is to distinguish between emissions from inside and outside the cycle; the aim of the replacement expression (see Step (vi.a)) 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) [9].

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 (v.c): All tests for cycle signals in the original program are extended by tests for their replacement signals. Using the SSA analogy, this corresponds to a ϕ-node [9].

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 (vi.a): The computation of replacement expressions is described in detail in Section 3 on page 11.

Step (vi.b): One signal in the set of cyclic signals must be selected as a point to break the cyclic dependency. Any signal in the cycle will work; the actual selection can be based on the smallest replacement expression computed in the next step.

Step (vi.c): $\mathcal{R}_i$ contains references to other cycle signals σ'_j. These are recursively replaced by their respective expressions $\mathcal{R}_j$ into $\mathcal{R}^*_i$. This unfolding of expressions is performed until only σ'_i and non-cyclic signals are referenced in $\mathcal{R}^*_i$.

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 (vi.d): Since the program is known to be constructive, it follows that σ'_i in $\mathcal{R}^*_i$ must not have any influence on the evaluation of $\mathcal{R}^*_i$. Therefore we can replace σ'_i in $\mathcal{R}^*_i$ by any constant value (true or false). Now $\mathcal{R}^*_i$ contains only non cyclic signals. It is important to replace just σ'_i and not σ_i, because σ_i is emitted outside the cycle and therefore not part of the cycle.

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

σ'_i has no influence on $\mathcal{R}^*_i$ for all reachable signal states and control states if the program is constructive. This does not necessarily hold for *all* states of signals in $\mathcal{R}^*_i$, but only for those reached at runtime at the evaluation of $\mathcal{R}^*_i$. This follows from the iterative process of signal replacement in Step (vi.c) which is equivalent to a symbolic version of a three-valued fix point iteration proposed by Malik [17] and Shiple *et al.* [20]. Note that if the result of the derived expression $\mathcal{R}^*_i$ is independent of σ'_i for all *reachable* signal combinations at the control point where $\mathcal{R}^*_i$ is evaluated, then the program is constructive with regard to signal σ_i at that control point. As mentioned above, this observation could be used to aid constructiveness analysis, as this would eliminate the need to perform a fixed-point iteration; nevertheless, to determine the reachable control flow and signal space remains a nontrivial problem.

Step (vi.e): The last transformation step in the algorithm replaces every occurrence of σ'_i in `present` tests by it's replacement expression $\mathcal{R}^*_i$. 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 $\mathcal{C}$.

Step (vii): 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 reincarnations; 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.

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 (vi.a) 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 [2] 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 of each `emit` statement 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:

- $\mathcal{R}$: $P \times C \rightarrow R$
  This function returns a mapping of signal names to their signal contexts at the point of their emission.
- $\mathcal{C}$: $P \times C \rightarrow C$
  $\mathcal{C}$ takes the signal context delivered by previous statements, 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 6. To determine the replacement expressions emit S:

$$\begin{aligned} \mathcal{R}((\mathsf{!S}),C) &= \{(\mathsf{S},C)\} \\ \mathcal{C}((\mathsf{!S}),C) &= C \end{aligned} \tag{1}$$

present S then p else q end:

$$\begin{aligned} \mathcal{R}((\mathsf{S?p,q}),C) &= \mathcal{R}(\mathsf{p},C \wedge S) \cup \mathcal{R}(\mathsf{q},C \wedge \overline{S}) \\ \mathcal{C}((\mathsf{S?p,q}),C) &= \mathcal{C}(p,C \wedge S) \vee \mathcal{C}(q,C \wedge \overline{S}) \end{aligned} \tag{2}$$

nothing:

$$\begin{aligned} \mathcal{R}((\mathsf{0}),C) &= \emptyset \\ \mathcal{C}((\mathsf{0}),C) &= C \end{aligned} \tag{3}$$

pause; emit ST_i:

$$\begin{aligned} \mathcal{R}((\mathsf{1;!ST\_i}),C) &= \emptyset \\ \mathcal{C}((\mathsf{1;!ST\_i}),C) &= \mathsf{ST\_i} \end{aligned} \tag{4}$$

exit T:

$$\begin{aligned} \mathcal{R}((\mathsf{k}),C) &= \{(\mathsf{T},C)\} \\ \mathcal{C}((\mathsf{k}),C) &= \mathsf{false} \end{aligned} \tag{5}$$

trap T in p end:

$$\begin{aligned} \mathcal{R}((\{\mathsf{p}\}),C) &= \mathcal{R}(\mathsf{p},C) \\ \mathcal{C}((\{\mathsf{p}\}),C) &= \mathcal{C}(\mathsf{p},C) \vee \bigvee_{(\mathsf{T},c_i)\in\mathcal{R}(\mathsf{p},C)} c_i \end{aligned} \tag{6}$$

p;q:

$$\begin{aligned} \mathcal{R}((\mathsf{p;q}),C) &= \mathcal{R}(\mathsf{p},C) \cup \mathcal{R}(\mathsf{q},\mathcal{C}(\mathsf{p},C)) \\ \mathcal{C}((\mathsf{p;q}),C) &= \mathcal{C}(\mathsf{q},\mathcal{C}(\mathsf{p},C)) \end{aligned} \tag{7}$$

loop p end:

$$\begin{aligned} \mathcal{R}((\mathsf{p*}),C) &= \mathcal{R}(\mathsf{p},C \vee \mathcal{C}(p,C)) \\ \mathcal{C}((\mathsf{p*}),C) &= \mathsf{false} \end{aligned} \tag{8}$$

[p || q]; emit ST_i:

$$\begin{aligned} \mathcal{R}(((\mathsf{p|q});\mathsf{!ST\_i}),C) &= \mathcal{R}(\mathsf{p},C) \cup \mathcal{R}(\mathsf{q},C) \\ \mathcal{C}(((\mathsf{p|q});\mathsf{!ST\_i}),C) &= \mathsf{ST\_i} \end{aligned} \tag{9}$$

signal S in p end:

$$\begin{aligned} \mathcal{R}((\mathsf{p}\backslash\mathsf{S}),C) &= \mathcal{R}(\mathsf{p},C) \\ \mathcal{C}((\mathsf{p}\backslash\mathsf{S}),C) &= \mathcal{C}(\mathsf{p},C) \end{aligned} \tag{10}$$

Fig. 6. Equations to determine replacement expressions for signals

for all signals in a program P, we compute $R := \mathcal{R}(P, \mathsf{ST\_0})$, where $\mathsf{ST\_0}$ denotes the boot signal, present only at startup in the very first instant. 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 in which signal context each signal is emitted. Now the expressions for the same signals can be

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

(a) present+ (b) present-

Fig. 7. Logical Behavioral Semantics of the present statement

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. The two SOS rules from the Logical Behavioral Semantics for the present statement, given in Figure 7 [2], 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 $\mathcal{C}$ (2) 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 Example Transformations

We will illustrate the transformation algorithm described in Section 2 by applying it to the PAUSE_CYC and TR3_CYC examples.

4.1 Transforming PAUSE_CYC

The algorithm is applied to the example PAUSE_CYC in Figure 1(a) on page 3, which is transformed into the acyclic program PAUSE_ACYC in Figure 1(c). The transformation of the program DRIVER_CYC in Figure 2(a), page 4, into DRIVER_ACYC in Figure 2(b) is similar.

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

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

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

Steps (iii) and (v): To prepare the removal of the cycle, we first transform PAUSE_CYC into the equivalent program PAUSE_PREP, shown in Figure 1(b). It differs from PAUSE_CYC in the introduction of state signals ST_0 to ST_2 and in the fact 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 (vi.a): The computation of replacement expressions for A_ and B_ according to Section 3 results in:

$$\mathsf{A\_} = \mathsf{ST\_1} \wedge (\mathsf{B} \vee \mathsf{B\_}) \tag{11}$$
$$\mathsf{B\_} = \mathsf{ST\_0} \wedge (\mathsf{A} \vee \mathsf{A\_}) \tag{12}$$

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. The similarity to a system of linear equations is apparent and we solve the the equations likewise:

Step (vi.b): In $\mathsf{PAUSE\_PREP}$, we arbitrarily select $\mathsf{A\_}$ as the signal to break the cycle.

Step (vi.c): To replace $\mathsf{B\_}$ in Equation (11), substituting (12) into (11) results in:

$$\mathsf{A\_} = \mathsf{ST\_1} \wedge (\mathsf{B} \vee (\mathsf{ST\_0} \wedge (\mathsf{A} \vee \mathsf{A\_}))). \tag{13}$$

This is now an equation which expresses the cycle signal $\mathsf{A\_}$ as a function of itself and other signals that are not part of the cycle; so we have unrolled the cycle.

Step (vi.d): We could now simulate (13) 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 $\mathsf{A\_}$ in this case) to arbitrary, known values without changing the meaning of the program [10]. Applying this to Equation (13) yields for $\mathsf{A\_} = \mathsf{false}$ (absent):

$$\mathsf{A\_} = \mathsf{ST\_1} \wedge (\mathsf{B} \vee (\mathsf{ST\_0} \wedge \mathsf{A})). \tag{14}$$

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

$$\mathsf{A\_} = \mathsf{ST\_1} \wedge (\mathsf{B} \vee \mathsf{ST\_0}). \tag{15}$$

We now have derived two equally valid replacement expressions for $\mathsf{A\_}$, which do not involve any cycle signal.

Step (vi.e): Finally we are ready to break the cycle in $\mathsf{PAUSE\_PREP}$. For that, we have to replace the signal selected in Step (b)—in the cycle—by an expression that does not use any of the cycle signals, without changing the meaning of the program.

Substituting (15), the simpler of these expressions, for $\mathsf{A\_}$ in $\mathsf{PAUSE\_PREP}$ yields the now acyclic program $\mathsf{PAUSE\_ACYC}$ shown in Figure 1(c).

4.2 Transforming the Token Ring Arbiter

Before transforming program $\mathsf{TR3\_CYC}$ from Figure 3 into the acyclic program $\mathsf{TR3\_ACYC}$ shown in Figure 8, 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 5.

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

```
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. 8. Non cyclic Token Ring Arbiter.

$$\begin{aligned}
\mathsf{P2} &= (\mathsf{ST\_0} \vee \mathsf{ST\_1}) \wedge (\mathsf{T1} \vee \mathsf{P1}) \wedge \overline{\mathsf{R1}}, && (16)\\
\mathsf{P3} &= (\mathsf{ST\_0} \vee \mathsf{ST\_4}) \wedge (\mathsf{T2} \vee \mathsf{P2}) \wedge \overline{\mathsf{R2}}\\
&= (\mathsf{ST\_0} \vee \mathsf{ST\_4}) \wedge (\mathsf{T2} \vee (\mathsf{ST\_0} \vee \mathsf{ST\_1}) \wedge (\mathsf{T1} \vee \mathsf{P1}) \wedge \overline{\mathsf{R1}}) \wedge \overline{\mathsf{R2}}, && (17)\\
\mathsf{P1} &= (\mathsf{ST\_0} \vee \mathsf{ST\_7}) \wedge (\mathsf{T3} \vee \mathsf{P3}) \wedge \overline{\mathsf{R3}}\\
&= (\mathsf{ST\_0} \vee \mathsf{ST\_7}) \wedge (\mathsf{T3} \vee (\mathsf{ST\_0} \vee \mathsf{ST\_4}) \wedge (\mathsf{T2} \vee (\mathsf{ST\_0} \vee \mathsf{ST\_1})\\
&\qquad \wedge (\mathsf{T1} \vee \mathsf{P1}) \wedge \overline{\mathsf{R1}}) \wedge \overline{\mathsf{R2}}) \wedge \overline{\mathsf{R3}}. && (18)
\end{aligned}$$

Equation (18) 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:

$$\mathsf{P1} = (\mathsf{ST\_0} \vee \mathsf{ST\_7}) \wedge (\mathsf{T3} \vee (\mathsf{ST\_0} \vee \mathsf{ST\_4}) \wedge (\mathsf{T2} \vee (\mathsf{ST\_0} \vee \mathsf{ST\_1}) \wedge \mathsf{T1} \wedge \overline{\mathsf{R1}}) \wedge \overline{\mathsf{R2}}) \wedge \overline{\mathsf{R3}}. \tag{19}$$

Setting P1 to true yields:

$$\mathsf{P1} = (\mathsf{ST\_0} \vee \mathsf{ST\_7}) \wedge (\mathsf{T3} \vee (\mathsf{ST\_0} \vee \mathsf{ST\_4}) \wedge (\mathsf{T2} \vee (\mathsf{ST\_0} \vee \mathsf{ST\_1}) \wedge \overline{\mathsf{R1}}) \wedge \overline{\mathsf{R2}}) \wedge \overline{\mathsf{R3}}. \tag{20}$$

The shorter expression (20) is 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 (20): 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 (20) can be reduced to:

$$\mathsf{P1} = (\mathsf{T3} \vee (\mathsf{T2} \vee \overline{\mathsf{R1}}) \wedge \overline{\mathsf{R2}}) \wedge \overline{\mathsf{R3}}. \tag{21}$$

Further examples including cycles over suspend and valued signals are included in a technical report [16].

5 Optimizations

The most important optimization refines the treatment of the present statement in Equation (2). Consider the following program fragment:

pause; **emit** ST_3;
present S **then emit** A **else emit** B **end**;
emit C

The application of the rules listed in Figure 6 on page 12 would result in a replacement expression for signal $\mathsf{C} = (\mathsf{ST\_3} \wedge \mathsf{S}) \vee (\mathsf{ST\_3} \wedge \overline{\mathsf{S}})$. It is obvious that this can be simplified to $\mathsf{C} = \mathsf{ST\_3}$. Or more generally:
Equation (2):

$$\mathcal{C}((\mathsf{S?p,q}), C) \;=\; \mathcal{C}(p, C \wedge S) \vee \mathcal{C}(q, C \wedge \overline{S})$$

can be simplified to

$$\mathcal{C}((\mathsf{S?p,q}), C) \;=\; C$$

if

$$\mathcal{C}(p, C \wedge S) = C \wedge S \quad \text{and} \quad \mathcal{C}(q, C \wedge \overline{S}) = C \wedge \overline{S}$$

holds.

This optimization alone yields a considerable reduction in the size of replacement expressions.

Another optimization (currently not implemented) would be to try to determine which state signals are always present or absent in a replacement expression. For example, the program PAUSE_ACYC can be optimized into the program PAUSE_OPT shown in Figure 1(d) on page 3 by taking the reachable presence status of the signals ST_0 and ST_1 into account.

Some more obvious optimizations (described further in an extended version of this paper [16], but currently unimplemented) apply to unused signals, which can be eliminated:

- State signals (introduced in Step (iii)) which are not part of any actually used replacement expression can be eliminated.
- Renamings for signals which are not referenced outside the cycle are not needed.
- If the signal which is selected for cycle breaking is not tested outside the cycle, then the emission of that signal can be removed from the program.

Note, however, that the usefulness of these optimizations may depend on further processing of the program, where these unused signals may be removed anyway.

6 Experimental Results

The proposed algorithm has been implemented in its basic form, so far 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 3.

`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 [5,13]. 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, as 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 (release 0.3) [7] produces event driven C code, which is generally quite fast and compact. However, this compiler cannot handle cyclic dependencies. Thus it can only be applied to the transformed cyclic programs.

CEC-g: The CEC with the option `-g` produces code using computed `goto` targets (an extension to ANSI-C offered by GCC-3.3 [14]) 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 two token ring examples 10,000,000 times and 30,000,000 times for the last (simpler) valued signal example. These iteration counts result in 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, 256KB Cache, 1GB Main Memory).

Table 1(a) compares the execution speed and binary sizes 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.

Table 1 includes the sizes of the compiled binaries, too. 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. While the v5 compiler seems to be competitive with respect to program run times, the binary sizes can reach several times the size of the binaries produced by the other compilers.

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 v5 compiler in automata mode, but only slightly

Variant	Compiler	TR3	TR10	TR10p
cyclic (original)	v5-L	1.55/ 14273	5.39/ 21530	17.19/ 32244
	v5-A	0.90/**13041**	2.58/**16091**	**5.26**/304095
acyclic (transformed)	v5-L	1.40/ 14067	5.07/ 20188	12.16/ 29110
	v5-A	0.89/**13043**	2.58/**16093**	**5.26**/304097
	v7	1.69/ 14526	6.07/ 20255	12.34/ 27353
	v7-O	**0.53**/ 13467	**1.87**/ 16315	5.83/**21033**
	CEC	1.80/ 14244	6.42/ 22020	12.04/ 29579
	CEC-g	1.09/ 13822	3.82/ 20430	5.89/ 25461

Table 1

Run times (in seconds) and binary sizes (in bytes) of cyclic and acyclic Esterel programs compiled with the v5, v7, and CEC compiler.

	TR3	TR10	TR10p
$min(T_{cyclic})$	0.90	2.58	5.26
$min(T_{acyclic})$	0.53	1.87	5.26
reduction	41%	28%	0.0%

Table 2

Relative run time reduction from the fastest cyclic version to the fastest version for the acyclic transformation, with $reduction = 100\% * (1 - min(T_{acyclic})/min(T_{cyclic}))$.

better than the CEC compiler with computed goto optimization. Nevertheless the huge binary produced by the v5 compiler in automaton mode limits its usefulness.

Table 2 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 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 three acyclic test cases. When this compiler is applied to cyclic programs, the compilation times are several times slower but within reasonable limits. The transformation times for the acyclic test programs (Table 4) are not included in Table 3, but even if we add the transformation times to the compilation times of the acyclic programs the picture will not change much.

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.

As an indication of the cost of the transformation algorithm in terms of

Variant	Compiler	TR3	TR10	TR10p
cyclic (original)	v5-L	0.08	0.29	1.38
	v5-A	**0.01**	**0.04**	10.86
acyclic (transformed)	v5-L	**0.01**	0.06	**0.10**
	v5-A	**0.01**	**0.04**	10.54
	v7	0.12	0.20	0.36
	v7-O	0.24	0.54	1.08
	CEC	0.15	0.35	0.76
	CEC-g	0.11	0.37	0.71

Table 3
Compiler run times for Esterel v5, v7, and CEC (in seconds).

Transformation	TR3	TR10	TR50	TR100	TR500	TR1000	TR10p
original size	1565	3705	16348	32159	162959	326470	5765
after module expansion	1370	4391	22031	44092	224892	450903	6995
after cycle transform.	2108	6856	34804	69920	359788	723804	9736
transform. time	0.05	0.07	0.27	0.57	5.18	17.5	0.11

Table 4
Transformation times (in seconds) and resulting program sizes (in bytes) for token ring arbiters with 3 to 1000 nodes.

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 5
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

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, 100, 500, and 1000 nodes. The size of the transformed code is nearly proportional with respect to the arbiter network size. The current transformation times show a sub-quadratic effort for the transformation.

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 5(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 5(b) compares the number of latches needed by the synthesization results. Here the CEC is able to reduce the number of latches considerably.

Table 5(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.

Table 5(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 compilers that can already handle cyclic programs.

Regarding future work, the transformation algorithm spells out only how to handle cycles carried by pure signals. This needs to be generalized to valued signals; one approach is suggested in an extended version of this paper [16].

There are also numerous optimizations possible, some of which presented in Section 5, 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.

We also thank Xin Li for conducting the hardware synthesis experiments.

Finally, we thank the reviewers for their detailed and constructive comments.

References

[1] 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.

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

[3] 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.

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

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

[6] 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.

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

[8] 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.

[9] 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.

[10] 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).

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

[12] 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.

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

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

[15] 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.

[16] Lukoschus, J., and Hanxleden, R. v. Removing cycles in Esterel programs. Technischer Bericht 0502, Christian-Albrechts-Universität Kiel, Institut für Informatik und Praktische Mathematik, Mar. 2005.

[17] Malik, S. Analysis of cyclic combinatorial circuits. *IEEE Transactions on Computer-Aided Design 13*, 7 (July 1994), 950–956.

[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).