Hindawi Publishing Corporation
EURASIP Journal on Embedded Systems
Volume 2007, Article ID 48979, 23 pages
doi:10.1155/2007/48979

Research Article

Removing Cycles in Esterel Programs

Jan Lukoschus and Reinhard von Hanxleden

Department of Computer Science, Christian-Albrechts-Universität zu Kiel, Olshausenstr. 40, 24098 Kiel, Germany

Received 1 June 2006; Revised 14 January 2007; Accepted 6 March 2007

Recommended by Alain Girault

Esterel belongs to the family of synchronous programming languages, which are affected by cyclic signal dependencies. This prohibits a static scheduling, limiting the choice of available compilation techniques for programs with such cycles. This work proposes an algorithm that, given a constructive synchronous Esterel program, performs a semantics-preserving source code level transformation that removes cyclic signal dependencies. The transformation is divided into two parts: detection of cycles and iterative resolution of these cycles. It is based on the replacement of cycle signals by a signal expression involving no other cycle signals, thereby breaking the cycle. This transformation of cyclic Esterel programs enables the use of efficient compilation techniques, which are only available for acyclic programs. Furthermore, experiments indicate that the code transformation can even improve code quality produced by compilers that can already handle cyclic programs.

1. INTRODUCTION

One of the strengths of synchronous languages [1] is their deterministic semantics in the presence of concurrency. Synchronicity implies instantaneous interactions between concurrent threads, which makes it possible to write a synchronous program that 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 all valid cyclic programs. 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 [2]; however, the concepts introduced here are applicable to other synchronous languages as well, such as Lustre [3].

Next we will provide an introduction to Esterel and cyclic programs, followed by an overview of previous work on compiling Esterel programs and handling cycles. Section 2 describes how to find cycles in an Esterel program, based on the computation of signal dependencies. Section 3 introduces the transformation algorithm, which is the main contribution of this paper. Optimization options are presented in Section 4, experimental results follow in Section 5. The paper concludes in Section 6, including an example of how to apply our transformation to Lustre.

1.1. Introduction to the Esterel language

The execution of an Esterel program is divided into discrete temporal *instants*, or *(logical) ticks*. In such an instant, the Esterel program communicates via *signals* with the environment and different parts of the program itself. In each instant, a signal can be in one of two states: *present* or *absent*. If a signal 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. All parts of the program have in each instant, the same view of all signal states emitted anywhere in the entire program.

The Esterel language consists of *kernel statements* and *derived statements*. As the latter are essentially "syntactic sugar" and can be derived from the former, we will restrict our attention here to kernel statements. The draft book [4] by Berry on the Esterel semantics provides a list and reasoning on the selection of kernel statements. Some example Esterel

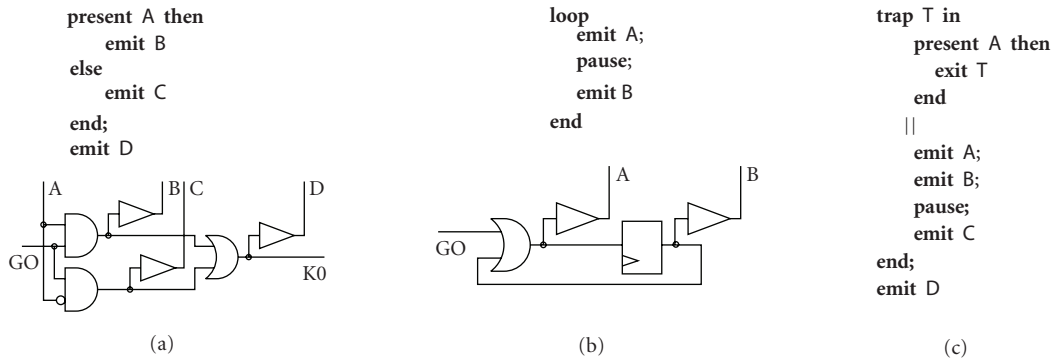


Figure 1: Three Esterel fragments and, for the first two, their translations into circuits.

program fragments are listed in Figure 1. The program in Figure 1(a) starts with a present statement that tests the state of signal A. If A is present, then the then branch is executed, which executes an emit statement that sets the state of signal B to present. Conversely, if A is absent, then the else branch sets C to present. Both branches terminate at the end of the present statement, and in both cases the emit D statement is executed.

Due to its deterministic and synchronous nature, the Esterel language is not only suitable as a programming language, but can also be used for the specification and synthesis of synchronous sequential logic circuits. To give an example, the circuits in Figures 1(a) and 1(b) depict the circuit translations of the respective Esterel codes, according to the circuit semantics [4]. The GO wire starts the circuit and via the K0 wire the circuit signals its termination.

Figure 1(b) contains two further structural parts of Esterel: loop and pause. A loop infinitely restarts its enclosed statement block when it terminates. The pause statement stops the execution for the remainder of the current instant and resumes execution in the following instant. The program fragment in Figure 1(b) has the following behavior: in the first instant, the loop starts its body, which executes the emit A statement. The pause statement stops the execution and nothing more happens for the remainder of the first instant. In the second instant, emit B is executed and the loop body terminates; however, the loop body is instantaneously restarted, emit A is executed again and pause is encountered again. For all the following instants the behavior of the second instant is repeated. Regarding the circuit translation, note how the pause gets translated into a register, which delays the GO input to the emission of B.

Figure 1(c) contains a trap and parallel threads as new elements. The trap statement consists of the definition of a trap signal (here T), which acts as an exception, and a body as a scope of that signal. The body may contain an exit T statement to activate the exception signal T. The control flow does not continue after the exit statement but after the entire trap statement. The behavior of trap and exit thus corresponds to catch and throw, respectively.

The trap body in the example in Figure 1(c) contains two parallel threads, separated by the concurrency operator ||. The parallel thread block terminates when all parallel threads are terminated. In the example in Figure 1(c), signal A is tested in the first thread, while it is emitted in the second thread. Since a signal's status must be consistent in the entire program for an instant, it follows that the emission of A must happen before it is tested in the first thread. Now the first thread is able to test A successfully, the exit T statement is executed. This signals to all other active threads in the associated trap T block to cease execution at reaching the next pause statement this corresponds to a *weak abortion*. As a consequence the second thread executes the emit B statement too. On reaching the pause statement control jumps immediately to the end of the trap block and executes the emit D statement.

Another type of control flow available in Esterel is a temporal delay of execution by a suspend statement:

suspend p **when** S

The execution of code block p is suspended for all those instants when the signal expression S is evaluated to true. An exception is the first instant when entering p, in that instant, S is not evaluated and no suspension takes place.

Figure 2(a) contains an Esterel program including an interface to the environment. Keywords input and output indicate the data direction of interface signals. The states of input signals are read from the environment at the beginning of each instant. Output signals carry their status from the Esterel program to the environment at the end of each instant. Note that input signals may also be emitted internally, and that output signals may be tested.

1.2. Cyclic programs, constructiveness

To illustrate the problem to be solved by the code transformation presented in this paper, we now introduce the concept of *cyclic programs*. We do this rather informally here, and adopt the terminology used in the Esterel primer [5, Chapter 5]. A more detailed treatment follows in Section 2.

As stated in the primer, *the availability of instantaneous broadcasting and control transmission makes it possible to write syntactically correct but semantically nonsensical programs.*

```
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
  ]
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
  ]
end signal
end module
```

(c)

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

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

(d)

FIGURE 2: Resolving a cycle: (a) original program with cycle between A and B, (b) introduction of state signals and shifting the cycle on internal signals, (c) replacement of cycle signal A_ by an expression, (d) optimized version.

A simple example for such a nonsensical program is:

present A **else emit** A **end**

If A is not present at the signal test, then the else part is executed and A is emitted, which invalidates the former signal test. Such a program is considered *nonreactive*, as it does not produce a well-defined output (A can be neither absent nor present). Replacing the else by a then in this example would produce a program that would yield more than one possible output (A could be either absent or present), which would be considered *nondeterministic*. Both variants of this example involve an *instantaneous dependency cycle* between A and itself, and both programs are considered *logically incorrect* and should be rejected by the compiler.

Considering the example above, one might conclude that all programs that contain dependency cycles should be rejected, and the Esterel v4 compiler did just that [6]. However, now consider the program PAUSE_CYC in Figure 2(a). This program also contains a dependency cycle, involving mutual dependencies between A and 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 instant. In such a case, where not all dependencies are active in the same execution instant, this is considered a *false cycle* [5]. Another classic example of a cyclic, yet meaningful program, is the token ring arbiter, shown in Figure 8 [7, 8]. The arbiter also contains a false cycle, and, even more problematic from a compiler's point of view, it does not even allow a static scheduling of the execution order.

We would like to accept programs such as PAUSE_CYC, and the Esterel v5 compiler does so by establishing that unique values for all signals can be determined in all execution contexts. The v5 compiler accepts all programs that are *constructive* [4], meaning that it forbids speculative reasoning, and restricts itself to fact-to-fact propagation according to the imperative nature of Esterel. An analogy in hardware circuitry is that a constructive Esterel program can be directly translated into a circuit where all wires reach unique, predefined voltage levels, irrespective of initial levels and propagation delays.

It is now generally agreed upon that constructive Esterel programs are meaningful, irrespective of whether they are cyclic or not. However, cyclicity poses a particular compilation challenge in that it prevents a compiler from establishing a static ordering for determining the signal values. As elaborated further in Section 1.3, this unfortunately precludes the application of the simulation-based compilation approaches, which are currently the most competitive in terms of execution speed and size. This is where the transformation presented in this paper steps in: it transforms cyclic, constructive programs into equivalent, acyclic programs, such that they are amenable to compilation also by the efficient compilers that require acyclicity.

As discussed further in Section 2.3, different compilers may have different notions of what constitutes a cycle, and whether they are able to handle specific cycles or not. Our code transformation is concerned with transforming constructive programs that are not schedulable by (some) compilers, due to (false) dependency cycles, into equivalent, schedulable programs. To make our transformation widely applicable, we are fairly conservative with respect to what constitutes a dependency and hence may lead to a cycle, and we completely eliminate all cycles even if a smart compiler might be able to determine them to be false.

The key observation, which the transformation presented here builds on, is that if a program is constructive, we can safely *break* a (false) cycle by replacing one of the signals

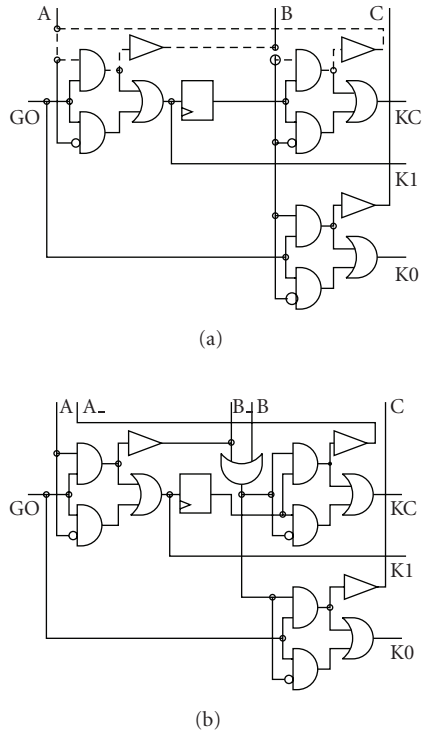


Figure 3: Circuit representation of the program PAUSE_CYC in Figure 2 (simplified without synchronizer): (a) cycle path of the original program PAUSE_CYC, (b) transformed, acyclic program PAUSE_OPT with new local signals A_ and B_.

contained in the cycle by a fresh signal. Consider the program PAUSE_OPT in Figure 2(d). This program is equivalent to PAUSE_CYC, but the dependency cycle is broken by replacing the emission of B within the cycle by a fresh signal B_. This can also be seen at the circuit level, see Figure 3; the netlist for PAUSE_CYC contains a feedback loop, which disappears in the netlist for PAUSE_OPT. To preserve program equivalence, tests for B outside of the cycle must test for B_ as well. However, inside the cycle, it suffices to just test for emissions of B (outside of the cycle), thus breaking the dependency cycle. The transformation mechanism presented here aims to automatically produce such equivalent, acyclic programs.

1.3. Related work

Today there exist three basic compilation techniques to synthesize code for a general purpose processor: automata, circuit, and event-based code synthesis. The automata-based Esterel v3 compiler developed by Berry and Gonthier [2] unfolds all parallel activities in the source program into a finite automaton with a single point of control. This removes all internal signaling (including cyclic dependencies) with the benefit of a very fast execution of the program. The drawback of this method is a possible state explosion for programs with many parallel threads.

The next generation of the Esterel compiler at Berry's group, the v4, implemented a different strategy. It is derived from hardware synthesis and is based on the simulation of circuits as the ones shown in Figure 1 in software [4]. This method translates parallelism in Esterel programs into parallel circuits, thus avoiding the state explosion of the automata code compiler. If the synthesized circuit contains cycles, then a dynamic schedule may be needed for the software simulation of the circuit and is therefore rejected by most Esterel compilers that use this approach.

Malik [9] describes a method to transform these cyclic circuits into acyclic ones. It is based on an iterative algorithm to compute the outputs of cyclic circuits with ternary simulation. Effectively the simulation run is serialized into an unfolding of the cycle path until the remaining inputs have no influence on the outputs. These inputs are replaced by constants, making the circuit acyclic. Shiple et al. [10] have developed Malik's work further by applying optimizations and incorporating cycles including registers into the algorithm. An implementation of this method is available in the Esterel v5 compiler [11].

A third approach to synthesize software is to generate an event-driven simulator, which breaks the program down into a number of small functional blocks that are conditionally executed. The CEC [12, 13] and SAXO-RT [14] compilers are based on this concept. 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 execution schedule. Therefore these efficient simulation-based compilation approaches are generally unable to compile cyclic programs, as already noted in the previous section.

The difficulties in giving a precise differentiation between cyclic and acyclic programs in different compilers (cf. Section 2.3) are also addressed by Potop-Butucaru [15] in his thesis. He identifies different compilation strategies and different internal dependency representations of Esterel programs as the source for discrepancies. Potop-Butucaru proposes to take the circuit synthesis [4] of Esterel programs as a reference and develops changes to his GRC compilation scheme to match the cycle properties of the v5 compiler. That approach is followed here, too, as we use the circuit semantics that underlies the v5 compiler as a reference to identify cycles in Esterel programs.

Besides software synthesis for a general purpose processor and hardware synthesis, the *reactive processing* approach makes use of specialized processors [16, 17]. However, the designs proposed so far also require acyclicity of the given programs, thus they could also benefit from our transformation.

One approach to overcome the limitation to cyclic programs, which is described by Edwards [18], is to unroll the strongly connected components (cycles) of circuits. 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.

The transformation of cyclic Esterel programs presented here builds on the work on cyclic circuits by Malik [9], Shiple et al. [10], and Edwards [18]. The main difference is that these previous approaches work on an internal

representation used by a particular compiler. Our transformation lifts this to the Esterel level, basically by evolving gate duplications into duplications of signal expressions. This makes our approach applicable to Esterel compilers in general, without the need to access their internals.

The key ideas to resolve cycles in Esterel programs were already described previously [8]. However, that earlier work did not cover the identification of cyclic dependencies, the computation of replacement expressions in the context of parallel termination and hierarchic trap blocks. Extensions to the algorithms to cover valued signals and alternative uses for replacement expressions in constructiveness analysis are presented in the dissertation of the first author [19].

The compilation of Esterel cannot only be complicated by cyclic dependencies, but also by *signal reincarnation*, also known as *schizophrenia* [4]. Most compilation schemes are not able to produce correct code for Esterel programs with schizophrenia problems. The code transformation presented in this work is not able to work on schizophrenic programs, either. A simple method to remove schizophrenia from a program involves code duplication, with potentially exponential cost in code size [4]. Several researchers have proposed more efficient cures for schizophrenia in Esterel programs [4, 20, 21]. These approaches work on the source code level, and could thus also be used as a preprocessing step to the transformation presented here.

We generally assume that the input programs that we transform are constructive in Berry's sense [4], as constructiveness is the property exploited by the way our algorithm breaks dependency cycles. However, it should be noted that this constructive semantics is not the only possible semantics for Esterel. More specifically, there have been different proposals as to which types of Esterel programs should be considered valid and which should be rejected. Boussinot has presented a number of alternatives, implemented in the SugarCubes project [22]. Another approach, called *maximal causality analysis*, has been suggested by Schneider et al. [23].

2. DETECTING DEPENDENCY CYCLES

We now refine the concept of program cycles that has been introduced informally in Section 1.2, and present an algorithm to detect such cycles.

2.1. Signal dependencies

We say that a signal P *depends* on signal S if, in some instant, for some sequence of input events, the presence or absence of P can only be decided on (according to the constructive semantics) if the presence or absence of S has been established. We then also say that this is a *dependency* from S to P. For a simulation-based compiler (cf. Section 1.3) a dependency represents a scheduling constraint.

The two most basic elements of a signal dependency are the *test* for a signal state (present S) and the *emission* of a signal (emit P). If both elements are combined in a program fragment

present S **then emit** P **end**

then a signal dependency is created; the presence state of S decides about the emission of P. Therefore the state of S must be known before the state of P can be established. In other words, signal S is a *guard* for signal P, or P *depends* on S.

The simple fact that an emit statement is contained in a subblock of a present statement is not a *sufficient* condition for a signal dependency. Consider this program fragment

present S **then emit** P; **pause**; **emit** Q **end**

The state of S decides over the emission of P in the same instant, which establishes a dependency between S and P. The emission of P is followed by a pause statement and an emission of signal Q inside the same then branch of the present statement. The pause statement defers the emission of Q to the subsequent instant, therefore the emission of Q is not influenced by the state of S in the same instant. Hence, S is a guard for P but not for Q.

While a signal emission being part of a present block is not a sufficient condition for a signal dependency, it is neither a *necessary* condition. Consider the following fragment:

present S **then nothing end**; **emit** P

In this example the emission of P is not part of the present block, but that block must terminate before the emission can take place. To execute the present block, the state of the signal S must be known. The fact that the then and (implicit) else branches both contain just a nothing statement is not relevant here. The constructive semantics of Esterel demands nonspeculative execution of the program, and the execution of either branches of present and subsequently emit P must be stalled until the state of all the tested signals is established. Therefore signal dependencies are established across sequential execution of statements, too, and S is a guard for P in this example.

Other sources for signal dependencies originate in loops connecting dependencies from the end to the start of the body, watcher expressions of suspend statements, or in exceptional control flows triggered by exit statements.

In the presence of reincarnation, this view requires to differentiate between different signal incarnations [4]. However, as we want to separate the reincarnation problem from the cyclicity problem, we will henceforth assume that signal reincarnation has been resolved by one of the known methods (cf. Section 1.3) before we get to analyze the program, and will assume that signals are unique.

2.2. Dependency cycles

We say that a program has a *(dependency) cycle*, or is *cyclic*, if there is a cyclic sequence of signal dependencies; this may also be, for example, a self-dependency. A dependency from S to P is *active* during a specific instant if during that instant the presence of P causally depends on the presence of S.

We say that a cycle is *false* if not all of its constituent dependencies can be active at the same instant. In case we can establish a fixed partial order among the emissions and tests of the signals involved in the cycle, as in the PAUSE_CYC example, we consider this false cycle to be *statically schedulable*.

Otherwise, it is *dynamically schedulable*, as for example in the token ring arbiter. If a program contains a *true cycle*, this cycle is not schedulable, and the program is not constructive and not considered further here.

2.3. The compiler's view

The definition of "signal dependencies" given in Section 2.1 and the subsequent definitions of cyclicity that build on it refer to the constructive semantics of Esterel, as defined, for example, by the Constructive Behavioral Semantics [4]. Ideally, all Esterel compilers we are concerned with would share exactly this view. There is indeed such a direct link between what we consider a true cycle and the v5 compiler: the v5 compiler, when using its built-in constructiveness analysis (option `-causal`), should accept a program if and only if it does not contain a true cycle. We have noticed that there are constructive programs (such as mejia [24]) where the constructiveness analysis of the v5 compiler fails; however, these seem to be implementation imperfections (e.g., regarding cycles on valued signals) and do not represent fundamental compiler limitations. Furthermore, these programs can still be compiled and run using the `-I` option, which generates interpretative code.

The picture becomes less clear when considering cycles in constructive programs, that is, false cycles. For a given program, it is decidable whether the program contains a false cycle or not, according to our definition, again based on Esterel's constructive semantics. Unfortunately, different compilers may have different ideas of what constitutes a (false) cycle, and whether a detected (false) cycle is statically schedulable or not. The Esterel examples discussed so far did not pose any particular problems for a compiler. However, there are other more involved cases that are more difficult for a compiler. Consider the following fragment:

[**present** S **then nothing end** || **pause**]; **emit** A

In this example, A does not depend on S, as the pause in the second thread causes the whole parallel statement to pause, thus separating the present S and the emit A into disjoint instants. The CEC does correctly detect that there is no dependency here; however the v5 compiler is unable to perform such reasoning, and assumes a dependency here. Conversely, there are other examples where the CEC detects dependencies that the v5 compiler does not [19]. The underlying reason for these discrepancies is the difference in the internal program representations used by these compilers [15].

Some compilers, such as the SAXO-RT, accept certain cyclic programs if they can establish that all cycles are false. The compiler tries to determine whether a cycle contains *exclusive* dependencies, meaning that they cannot be activated in the same instant, for example, if they belong to different branches of a present test or if they are separated by a pause statement. Thus, the SAXO-RT is able to compile, for example, the PAUSE_CYC program. However, this fails for cycles which must be dynamically scheduled, such as the aforementioned token ring arbiter.

As compilers in general do not perform an exhaustive causality analysis, they perform conservative approximations when analyzing dependencies, and when analyzing whether dependency cycles are statically schedulable or not. Therefore, to make our transformation applicable as broadly as possible, while at the same time avoiding unnecessary transformations, it would be ideal if we would transform only those cycles that are rejected by the compiler that is actually used. Therefore that compiler would have to provide details on rejected cycles. However, compilers typically do not provide this information, at least not in a standardized fashion. As part of the algorithm, we detect cycles ourselves before resolving them. The algorithm for doing so is presented in the next section.

2.4. Algorithm to detect dependency cycles

The algorithm presented here is divided into two parts: (1) identification of all direct signal dependencies, and (2) searching for cycles in signal dependencies. The first part is specified as a structural induction on Esterel programs with a set of signal pairs as a result. Each signal pair describes a dependency from one signal to another. These pairs can be interpreted as edges in a directed graph with the signal names as nodes. The second part of the algorithm, the detection of cycles in such a graph, is straightforward.

The method to derive signal dependencies here is based on the Esterel circuit semantics, as also used by the v5 compiler. The plain circuit semantics just defines the expansion of Esterel statements into circuits without any optimizations. The v5 compiler actually performs additional circuit minimization steps on the resulting circuit, for example, by propagating constant input values. This reduces the size of the resulting circuit considerably and is sometimes able to remove signal dependencies that would lead to cyclic dependencies otherwise.

Our method to derive signal dependencies uses no optimizations, primarily because we do not explicitly generate a circuit for the full program, making general logic optimization algorithms not applicable. Furthermore we want to provide a conservative solution for cyclic dependencies, which does not depend on specific optimizations implemented in specific compilers. The drawback of this conservative approach is that cycles may be resolved unnecessarily.

All signal dependencies lead from signal emissions to signal tests. This leads to the basic idea of our approach to search for signal dependencies in Esterel programs.

(1) The program is recursively traversed.
(2) Signals in a present condition are added to a set G of *guard signals*.
(3) If an emit statement is encountered, then new dependencies from all signals in G to the emitted signal are collected in a global set D of signal dependencies.
(4) G is emptied when a pause statement is encountered.

The rule set listed in Algorithm 1 implements the computation of signal dependencies for kernel statements of Esterel. The suspend statement is left out for reasons laid out in the discussion of Step (2d) of the transformation

emit S	$\mathcal{G}(P,G,X) = G$ $\mathcal{D}(P,G,X) = \{\langle a,S\rangle \mid a \in G\}$	(1)
present S **then** p **else** q **end**	$\mathcal{G}(P,G,X) = \mathcal{G}(p,G\cup\{S\},X) \cup \mathcal{G}(q,G\cup\{S\},X)$ $\mathcal{D}(P,G,X) = \mathcal{D}(p,G\cup\{S\},X) \cup \mathcal{D}(q,G\cup\{S\},X)$	(2)
pause	$\mathcal{G}(P,G,X) = \varnothing$ $\mathcal{D}(P,G,X) = \{\langle a,s\rangle \mid a \in G,\ s \in X\}$	(3)
nothing	$\mathcal{G}(P,G,X) = G$ $\mathcal{D}(P,G,X) = \varnothing$	(4)
loop p **end**	$\mathcal{G}(P,G,X) = \varnothing$ $\mathcal{D}(P,G,X) = \mathcal{D}(p,G\cup\mathcal{G}(p,G,X),X)$	(5)
$p;q$	$\mathcal{G}(P,G,X) = \mathcal{G}(q,\mathcal{G}(p,G,X),X)$ $\mathcal{D}(P,G,X) = \mathcal{D}(p,G,X) \cup \mathcal{D}(q,\mathcal{G}(p,G,X),X)$	(6)
trap T **in** p **end**	$\mathcal{G}(P,G,X) = \{a \mid \langle a,T\rangle \in \mathcal{D}(p,G,X\cup\{T\})\} \cup \mathcal{G}(p,G,X\cup\{T\})$ $\mathcal{D}(P,G,X) = \{\langle a,s\rangle \in \mathcal{D}(p,G,X\cup\{T\}) \mid s \neq T\}$	(7)
exit T	$\mathcal{G}(P,G,X) = \varnothing$ $\mathcal{D}(P,G,X) = \{\langle a,T\rangle \mid a \in G\}$	(8)
$p \mid\mid q$	$\mathcal{G}(P,G,X) = \mathcal{G}(p,G,X) \cup \mathcal{G}(q,G,X)$ $\mathcal{D}(P,G,X) = \mathcal{D}(p,G,X) \cup \mathcal{D}(q,G,X)$	(9)
signal S **in** p **end**	$\mathcal{G}(P,G,X) = \mathcal{G}(p,G,X)$ $\mathcal{D}(P,G,X) = \mathcal{D}(p,G,X)$	(10)

Algorithm 1: Equations to determine signal dependencies from kernel statements. Suspend is replaced by other kernel statements. $\mathcal{D}$ collects signal dependencies from signal emissions, $\mathcal{G}$ returns the active guard signals from terminating statement blocks. P stands for the corresponding program fragment shown on the left, G is the set of guard signals, and X is the set of trap signals in the current context.

algorithm in Figure 4. While traversing the syntactical elements of a program P with signals Σ, two sets of active guard signals are maintained.

- $G \subset \Sigma$: set of signals (*guards*) comprising the current present conditions.
- $X \subset \Sigma$: set of trap signals (*exceptions*) in the current scope.

Signals in X are kept separate from G because trap signals are not removed by pause statements. The rules to derive signal dependencies from Esterel programs are implemented in two functions (with Π as the set of Esterel programs and Σas the set of signals):

$$\mathcal{D}\colon \Pi \times 2^\Sigma \times 2^\Sigma \longrightarrow 2^{\Sigma\times\Sigma} \quad (P,G,X) \longmapsto \mathcal{D}(P,G,X)$$
$$\mathcal{G} : \Pi \times 2^\Sigma \times 2^\Sigma \longrightarrow 2^\Sigma \quad (P,G,X) \longmapsto \mathcal{G}(P,G,X)$$

$\mathcal{D}$ computes the signal dependencies from the current guard signals. The result is a set of signal pairs describing all signal dependencies in P. It uses $\mathcal{G}$to handle sequences of statements. $\mathcal{G}$returns the set of guard signals that are active when the control flow leaves the program block P. To extract the

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, except for suspend which is handled in Step (2d).
  (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) Add explicit termination handling to ll statements.
  (f) Add emission of individual state signals ST_i to the start of the program and each pause statement.
(3) Identification of cyclic signal dependencies:
  (a) Identify all signal dependencies $D = \mathcal{D}(P, \emptyset, \emptyset)$.
  (b) If Ddoes not contain cycles: **Done**.
  Otherwise: Select a shortest cycle $\vec{C}$, of length l.
(4) Transform P into P'; do for all $\sigma_i \in \vec{C}$, if σ_i is an input signal in the module interface:
  (a) Globally declare a new signal σ_i'. σ_i' replaces σ_i in $\vec{C}$.
  (b) Replace "emit σ_i" by "emit σ_i'."
  (c) Replace tests for σ_i by tests for "(σ_i or σ_i')."
(5) Transform (still cyclic) P' into (possibly acyclic) P'':
  (a) For all $\sigma_i \in \vec{C}$ determine replacement expressions $E_i = \mathcal{E}(P', \text{ST\_0})$ (see Algorithm 2).
  (b) Select some cycle signal $\sigma_p \in \vec{C}$ as the pivot signal to break the cycle.
  (c) Iteratively transform E_p to E_p^* by replacement of all signals $\sigma_j \in (\vec{C} \setminus \sigma_p)$ by their expressions E_j.
  (d) Transform E_p^* into E_p^{**} by replacing σ_p by false (or true) and minimize result. Now E_p^{**} does not involve any cycle signals.
  (e) Replace all tests for σ_p in P' by E_p^{**} resulting in P''.
(6) Goto Step (3), treat P'' now as P.

Figure 4: Transformation algorithm for pure signals.

signal dependencies $D$from an Esterel program P, $\mathcal{D}$ is applied to P with initially empty guard sets: $D = \mathcal{D}(P, \emptyset, \emptyset)$.

As an example for how our cycle detection mimics Esterel's circuit semantics, consider again the example in Figure 1(a). Here signals B and C are emitted under direct control of signal A, therefore B and C depend on A. This is visible in the circuit as the two and gates that depend on A and control the activation of the signal drivers for B and C. Signal D is emitted in sequence of the present statement. The dependency of D on A is visible in the circuit by following the path through the or gate.

These kinds of dependencies are determined by collecting all signals in present conditions as guard signals. The order of the recursive traversal is set up to mirror the (internal, anonymous) wires connecting the different parts of the circuit. If a pause statement is encountered, then the guard set G is emptied because signal dependencies do not reach over pause statements, which correspond to registers at the circuit level.

The body p of loop is evaluated twice in $\mathcal{D}$ to capture dependencies resulting from an instantaneous restart of the loop body. In the following example signal B depends on A because of such an instantaneous restart:

loop emit B; **pause**; **present** A **then nothing end end**

Trap/exit represents a different kind of flow of control than that is followed by the recursive traversal. An exit T statement transfers control to the surrounding trap T environment. Our recursive dependency analysis does not directly follow these exceptional control paths, but instead stores the trap signal as a dependency on the guard set G. When the recursive traversal of a trap T body is completed, all currently stored guards on the trap signal are retrieved from the dependency list and put back into the guard set. Additionally all pause statements within a trap T environment connect the current guard set to T to capture dependencies resulting from parallel threads to an exit statement.

The rules in Algorithm 1 contain no optimizations regarding unreachable code. That simplification may lead to the detection and subsequent transformation of cycles which are not rejected by Esterel compilers in the first place. An extended rule set that detects dead code is given in the thesis [19].

3. PROGRAM TRANSFORMATION

After identifying cyclic dependencies, we are now able to resolve those cycles by transforming the program as described in the following.

3.1. The base transformation algorithm

Figure 4 presents the algorithm for transforming cyclic Esterel programs into acyclic programs. Each transformation step is discussed along with its worst-case increase in code size. The core of the algorithm is Step (5d), which breaks dependency cycles; however, to do this step, we have to first analyze and preprocess the program. The algorithm presented here is applicable to programs with cycles that involve pure signals only. Extensions to support valued signals as well are presented elsewhere [19].

Step (1): the constructiveness of the Esterel program is a precondition for the transformation. It can be performed using, for example, the methods developed by Berry and by Shiple et al. [4, 10]; one available implementation is offered by the v5 compiler [25]. The constructiveness property is exploited in Step (5d) of the algorithm. Note that as acyclicity implies constructiveness, we may first run an acyclicity test, by the compiler, or by Step (3) of our algorithm, which is generally cheaper than a full constructiveness analysis.

Step (2): the core algorithm is only applicable to Esterel programs that have been preprocessed as follows.

Step (2a): the expansion of modules is a straightforward textual replacement of module calls by their respective body.

No dynamic run time structures are needed, since Esterel does not allow recursions.

Step (2b): regarding the statements handling signals, the transformation algorithm is expressed in terms of Esterel kernel statements. Therefore all derived statements must be expanded to kernel statements.

Step (2c): we have to eliminate locally defined signals because replacement expressions for signals computed by the algorithm could carry references to local signals out of their scope. (Note that the programmer may still freely use local signal declarations.) Furthermore, the method of finding replacement expressions assumes that signals are unique, that is, not reincarnated. To address the problem of reincarnation, algorithms with different efficiency are available [4, 21].

Step (2d): the introduction of state signals fails in the context of suspend statements, because state signals emitted as part of a suspend block are suspended too. This constitutes a dependency of the state signals on the—possibly cyclic—suspension condition and may lead to a new cycle, thus preventing a successful reduction on the number of cycles.

The solution proposed here simulates the behavior of suspend blocks by means of other kernel statements (trap, exit, pause, present, loop), which are handled directly. The key difference to the original suspend behavior is the handling of state signals; they are emitted regardless of suspension conditions. This avoids unwanted dependencies for state signals.

Suspension blocks are transformed by removing the suspend envelope:

$$\textbf{suspend}\ \mathrm{p}\ \textbf{when}\ \mathrm{S} \quad \rightsquigarrow \quad \mathrm{p'}$$

Here p denotes the suspended body and S denotes the suspension condition. They are replaced just by the body p' derived from 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:

$$\textbf{await not}\ S \quad \rightsquigarrow \quad \begin{array}{l}\textbf{trap}\ \mathsf{T}\ \textbf{in loop}\\ \quad \textbf{pause}; \textbf{present}\ S\ \textbf{else exit}\ \mathsf{T}\ \textbf{end}\\ \textbf{end end}\end{array}$$

The complexity of this part of the transformation is a constant factor of the number of pause statements inside suspend statements.

When transforming cascaded suspend blocks, then each suspension block can be treated individually. For each suspend definition another layer of trap/loop blocks will be put around included pause statements. The order of transformations of the suspend blocks is not relevant here.

An alternative solution to handle state signals inside suspend blocks, which is based on a small extension to the Esterel language, is proposed in the thesis [19]. It is based on the emission of state signals with no regard for suspension, thus avoiding the introduction of dependencies from the suspend condition to the enclosed state signals introduced in later steps.

Step (2e): another case of hidden program state is present in the termination control of parallel statements. Consider a parallel block with two threads:

$$p \ || \ q$$

This parallel block terminates, if both subblocks p and q terminate. The precise signal state of termination of the whole parallel statement is not directly accessible, because threads that are terminated in earlier execution instants do not emit any signals anymore. Therefore a simple signal expression will generally not describe the termination context of parallel statements.

Figure 5 illustrates the addition of auxiliary signals at the end of each thread in a parallel statement. These signals are continuously emitted, once that thread is terminated. An additional thread tests for the conjunction of all these auxiliary signals. If all auxiliary signals are present, then the entire parallel statement is terminated via a trap exception. This transformation replaces the regular termination mechanism of parallel statements by trap exception handling, which is covered by the regular algorithm.

Step (2f): the execution state of an Esterel program is stored in variables defined inside the synthesized code. Unfortunately there is no provision at the Esterel level to access the state of these variables. The introduction of additional 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 "pause; emit ST_i." The emission of the first state signal "emit ST_0;" is added to the program start. ST_0 corresponds to the boot register in the circuit representation of Esterel programs. Note that many of the signals may be eliminated again by subsequent optimizations, see Section 4.

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 is used to search for cyclic dependencies in the Esterel program. Only signals which are part of the currently resolved cycle are of further interest. More details on the detection of cyclic dependencies are given in Section 2.4. If there is more than one cycle present in the program, then Steps (4) and (5) are performed for each cycle individually. In each cycle resolving step the currently smallest cycle must be selected to be resolved. This ensures the termination of the iterative expression transformation in Step (5c).

The signals comprising the currently selected cycle are called *cycle signals* in the following.

Step (4a/4b): this step splits each cycle input signal σ_i into two signals σ_i and σ_i'. Input signals to/from sub-modules are not addressed here, only the connections to the environment. The motivation of this step is to distinguish between emissions from inside the Esterel program and from the environment. The signal with the original name σ_i is under the control of the environment. All signal emissions in the program itself use the new signal name σ_i'. The aim of

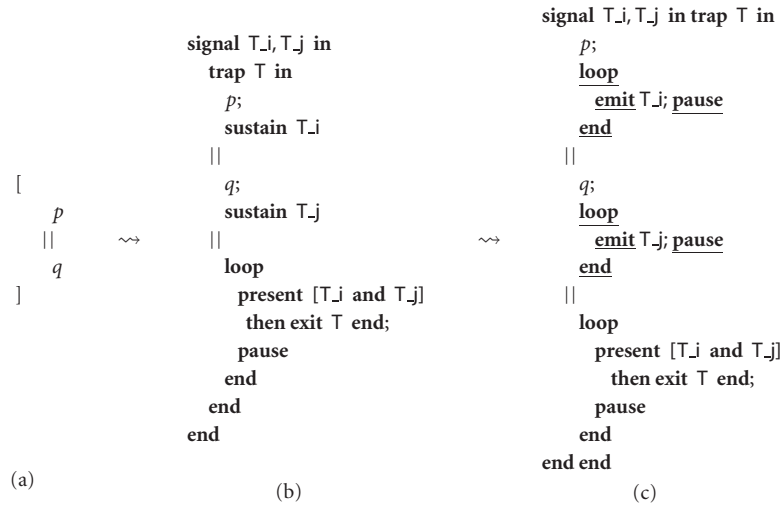


Figure 5: Making the termination state of parallel threads visible to signal expressions by continuous emission of auxiliary signals on terminated subthreads: (a) original parallel block with threads p and q, (b) added termination handling by trap, (c) expansion of sustain into kernel statements.

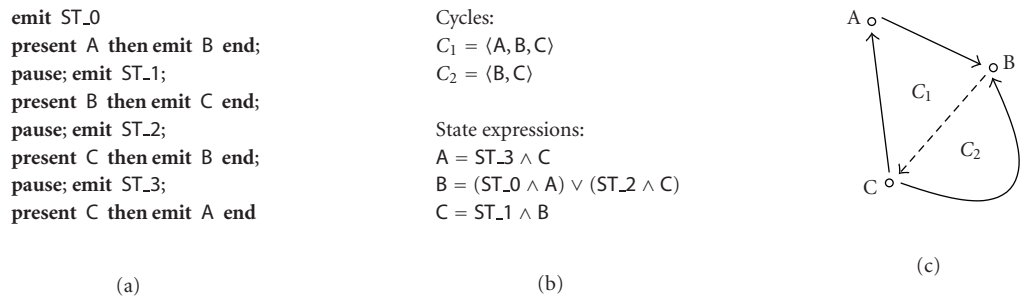


Figure 6: Program with potentially nonterminating iterative signal replacement: (a) cyclic program containing two cycles with a common dependency from B to C, (b) cycles and emission contexts of signals, (c) graphical representation of the two cycles, the common dependency is indicated by a dashed line.

the replacement expressions (see Step (5a)) is to substitute tests on cycle signals by expressions made up of noncycle signals. This is not possible for input signals since their behavior cannot be derived from signals in the program. In a way, this introduction of fresh signals, which are emitted exclusively in the cycle, is akin to static single assignment (SSA) [26].

Step (4c): all tests for cycle input signals in the original program are extended by tests for their replacement signals. Using the SSA analogy, this corresponds to a ϕ-node [26].

Step (5a): the computation of replacement expressions E is described in detail in Section 3.2.

Step (5b): one signal in the set of cycle signals must be selected as a point to break the cyclic dependency. Any signal in the cycle will work; for example, we may select the signal that generates the smallest replacement expression as computed in the next step.

Step (5c): the replacement expression E_p for the selected cycle signal σ_p contains references to other cycle signals σ_j. These are recursively replaced by their respective expressions E_j into E_p^*. This unfolding of expressions is performed until only σ_p and noncycle signals are referenced in E_p^*.

For Esterel programs containing multiple overlapping cycles that unfolding may not terminate. Figure 6 contains an example for such problematic cycles. If C_1 is selected as the first cycle to resolve, then the iteration will not terminate by oscillating between the replacement and reintroduction of B and C. As a simple remedy it suffices to always select the currently *smallest* cycle in Step (3). The following argues the validity of generally solving the termination problem of the iteration by selecting a smallest cycle.

Preconditions 1. Given constructive Esterel program P, including n cycles $\vec{C}_1, \ldots, \vec{C}_n$ involving signals $\sigma_i \in \Sigma$, one shortest cycle $\vec{C}_k$ is selected such that for all $i \in \{1, \ldots, n\}$: $|\vec{C}_k| \leq |\vec{C}_i|$. The emission contexts of all cycle signals $\sigma_i \in \vec{C}_k$ are represented by expressions E_i ($i \in \{1, \ldots, |\vec{C}_k|\}$). One signal $\sigma_p \in \vec{C}_k$ with associated expression E_p is arbitrarily

selected as the pivot element to break the cycle and perform the iteration.

Theorem 1. *The iterative replacement of cycle signals $\sigma_i \in \vec{C}_k$ $(i \neq p)$ in expression E_p by expressions E_i terminates after a finite number of steps.*

Proof. The occurrences of signals in expressions relate to signal dependencies found in the cycle analysis step: if an expression E_i for a signal σ_i contains a signal σ_j, then a dependency $\langle \sigma_j, \sigma_i \rangle$ exists. Dependencies on state signals are omitted here because they are not part of any cycle by design. The iterative replacement of all cycle signals by their respective expressions stops at signal σ_p. This iteration is structurally equivalent to a reverse traversal on the signal dependencies starting from σ_p with the following restrictions: only those signal dependencies are followed where both signals in the dependency are part of the cycle, the traversal stops if σ_p is reached.

The traversal does not terminate if a loop in signal dependencies is encountered. Two cases exist for such loop structures.

A single signal dependency may directly connect an already visited cycle signal to the current cycle signal. Together with the already traversed dependencies this constitutes a cycle with fewer signals than $\vec{C}_k$. That is, a contradiction to the precondition on selecting the shortest cycle to resolve first.

The other case is a chain of two or more dependencies connecting back to the cycle. The signals connected by this chain cannot be part of cycle $\vec{C}_k$ (besides the first and last signals in the chain) because otherwise they would be identical to dependencies already included in $\vec{C}_k$ or are covered by the previous case. Therefore in this case only signals outside the cycle are covered. Since such signals are not traversed in the iteration, no loop in the iteration is present here. □

The complexity of the replacement expressions depends on the length of the cycle, because the length of the cycle governs 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 relationship 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 potential growth in program size for one cycle is of cubic complexity.

Step (5d): this is the central step of the transformation. Since the program is known to be constructive, it follows that σ_p in E_p^* must not have any influence on the evaluation of E_p^*. Therefore we can replace σ_p in E_p^* by any constant value (true or false). The resulting expression E_p^{**} contains only noncycle signals. This replacement of a cycle signal by a constant is described in Malik's work [9] on resolving cycles in cyclic circuits. The following argues the validity of this replacement.

Preconditions 2. Given constructive Esterel program P, including cycle involving signal σ_p, and other signals $\vec{S} = \langle s_0, \ldots, s_n \rangle$, a replacement function $E_p^*(\sigma_p, \vec{S})$ for signal σ_p is derived as of Step (5c) according to the circuit semantics of Esterel.

Theorem 2. *P is constructive $\Rightarrow E_p^*(\mathsf{true}, \vec{S}) = E_p^*(\mathsf{false}, \vec{S})$ for all reachable $\vec{S}$.*

Proof. P is constructive, that is, the state of all signals (including σ_p) can be determined without speculative reasoning for all reachable states of the program. Therefore the status of σ_p can be derived without previous knowledge of the status of σ_p; in other words, σ_p is not allowed to depend on itself. E_p^* computes the status of signal σ_p from signals in P including σ_p itself.

Assuming that E_p^* yields different results for true and false in place of σ_p would make E_p^* dependent on σ_p. This contradicts the constructiveness of P. Therefore E_p^* cannot depend on the status of σ_p. □

Remark 1. The use of constructiveness here implies *strong* constructiveness as defined by Shiple et al. [10], that is, even local nonconstructiveness with no influence on output signals is not allowed.

Step (5e): the last transformation step in the algorithm replaces every occurrence of σ_p in present tests by its replacement expression E_p^{**}. Now we have replaced one signal of the cycle by an expression which does not contain any references to signals of the cycle. Therefore we have *broken* the current cycle $\vec{C}$.

Step (6): the transformation algorithm must be repeated until all cycles are resolved, and the upper limit of cycles to resolve is the number of statements in the program (counting signals, conditionals, emissions, etc.).

It is possible to create an Esterel program with an exponential number of cycles on signals by connecting them in a mesh-like structure. These kinds of cycles share signal dependencies, therefore cutting one signal dependency will resolve multiple cycles, reducing the maximum number of iterations down to the number of signals.

On the other extreme lies a program with signal dependencies connecting all signals to every other signal. In this case each cycle must be resolved individually leading to a quadratic effort with regard to the number of signals. But to establish the net of signal dependencies the program itself must represent each individual dependency at least as a single statement. Therefore the number of cycles to resolve is of linear effort relative to the program size.

Cost of the transformation algorithm

We now analyze the worst-case code size increase that our transformation may induce. We do not consider the constructiveness analysis part of the transformation itself, and, as it is a prerequisite for compilation of cyclic programs in any case, do not consider it in our complexity analysis of the algorithm. (Conversely, our transformation could potentially be used to speed up constructiveness analysis [19].) Similarly, the expansion of modules in Step (2a), which has potentially

exponential cost, and possibly the resolving of reincarnation in Step (2c) must be done by Esterel compilers anyway. Steps (2b) and (2d) to (4c) all introduce a cost of a constant factor to different parts of the Esterel program. Therefore the overall cost of these steps can be summarized to be a constant factor to the size of the entire Esterel input program. In Step (5c) the actual cycle cutting takes place with a cost of cubic complexity.

Overall, a very conservative estimate results in a cost and code size increase 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 5) also demonstrate that the transformation typically results in a code size reduction.

3.2. 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 (5a) of the algorithm). That expression is derived from the control flow contexts of the program where the signal is set by emit statements. This section presents a set of rules to derive these replacement expressions. These rules are based on the logical behavioral semantics rules [4] with the aim of an easy implementation.

The objective of the rules is to obtain replacement expressions for all signals. A 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. These signal contexts are used to derive the replacement expressions. A current signal context expression $S$is modified while traversing the Esterel program P. The context expressions at the point of signal emissions are collected and combined into replacement expressions for all cycle signals. The rules to traverse the Esterel program are implemented in two functions (with Π as set of Esterel programs, Σ as the set of signals, and Ψ as the set of signal expressions):

$$\mathcal{E} : \Pi \times \Psi \longrightarrow 2^{\Sigma \times \Psi} \quad (P \times S) \longmapsto \mathcal{E}(P, S),$$
$$\mathcal{S} : \Pi \times \Psi \longrightarrow \Psi \quad (P \times S) \longmapsto \mathcal{S}(P, S).$$

Function $\mathcal{E}$searches for signal emissions in program P and returns a mapping of signal names to their signal contexts at the point of their emission. The function $\mathcal{S}$takes the signal state context delivered by previous statements, computes the signal state context from substatements, and returns the signal context for evaluation on sequentially following statements. It is used by $\mathcal{E}$as a helper function.

These functions are computed by structural induction over their first argument (an Esterel program); the corresponding definitions for each kernel statement are given in Algorithm 2. To determine the replacement expressions for all signals in a program P, we compute $E := \mathcal{E}(P, \mathsf{ST\_0})$, where ST_0 denotes the boot signal, present only at startup in the very first instant. The result in E 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. Multiple emissions of the same signal result in multiple entries of that signal in E. The expressions for the same signals can now be disjuncted to yield a single replacement expression for the emission of each cycle signal:

$$E_i = \bigvee_{\langle \sigma_i, S_j \rangle \in E} S_j. \tag{11}$$

As an example to illustrate how the definitions of $\mathcal{E}$and $\mathcal{S}$correspond to the behavioral semantics, consider the present statement. The two structurally operational semantics (SOS) rules from the logical behavioral semantics for the present statement, given in Figure 7, select the rule to apply based on the status of the condition signal s and the resulting control flow. The selected rule will add signal emissions and so forth to the resulting context. The corresponding equations for $\mathcal{E}$and $\mathcal{S}$(13) consider both possible control flow paths, and both paths may add signal emissions to E; however, each signal emission is tied to the condition for that part, thus reflecting the original semantics.

Rule (15) handles the pause statement with associated emission of its state signal. Function $\mathcal{E}$does not return a context expression for the emission of a state signal, because state signals are emitted behind pause statements and are therefore not subject to any signal dependencies by construction. This is needed since state signals are used to represent the state of signal emissions. Additional dependencies could lead to new cycles, spoiling the transformation. Function $\mathcal{S}$replaces the previous state with the name of the current state signal. The state signal is replaced by false, if the sequentially previous command returned false too. The case switch with condition "S = false" is a simple way to optimize for an unreachable pause statement. In that case the program state is kept false in that thread. If the condition is not successfully evaluated for an S with a value of, for example, "false or false," then only some efficiency is lost.

Trap signals are treated differently from regular signals: function $\mathcal{E}$in rule (16) (exit) adds the current signal context as an emission context for the trap signal to E. The trap signal name is marked with a prefix "exit" to be able to distinguish it from regular signals. Function $\mathcal{S}$in that rule returns false as a signal context state to indicate that sequentially following code is not reachable.

Function $\mathcal{E}$in rule (17) (trap) removes all references to its own trap signal to not interfere with upper trap definitions. Function $\mathcal{S}$ in rule (17) implements the task to compute the termination context of the trap statement. It consists of the normal termination part with no exception taking place, given by $\mathcal{S}(p, S)$. The signal context states of control flows triggered by exit statements are extracted from the emission context $\mathcal{E}(p, S)$. Those signal context states are limited to exit statements referencing the locally defined trap signal ($\sigma_i = T$). The signal contexts of other trap signals ($\sigma_i \neq T$) are negated, because they reference upper trap statements with higher priorities. In Esterel it is possible to specify hierarchic trap definitions sharing the same trap signal name. In

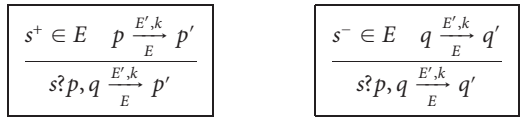
$P =$ **emit** A	$\mathcal{E}(P,S) = \{\langle S, A\rangle\}$ $\mathcal{S}(P,S) = S$	(12)
present A **then** p **else** q **end**	$\mathcal{E}(P,S) = \mathcal{E}(p, S \wedge A) \cup \mathcal{E}(q, S \wedge \overline{A})$ $\mathcal{S}(P,S) = \mathcal{S}(p, S \wedge A) \vee \mathcal{S}(q, S \wedge \overline{A})$	(13)
nothing	$\mathcal{E}(P,S) = \varnothing$ $\mathcal{S}(P,S) = S$	(14)
pause; **emit** ST_i	$\mathcal{E}(P,S) = \varnothing$ $\mathcal{S}(P,S) = \begin{cases} \mathsf{false}: & S = \mathsf{false} \\ \mathsf{ST\_}i: & \textbf{otherwise} \end{cases}$	(15)
exit T	$\mathcal{E}(P,S) = \{\langle \mathsf{exit}\ T, S\rangle\}$ $\mathcal{S}(P,S) = \mathsf{false}$	(16)
trap T **in** p **end**	$\mathcal{E}(P,S) = \{\langle \sigma_i, S_j\rangle \in \mathcal{E}(p,S) \mid \sigma_i \neq \mathsf{exit}\ T\}$ $\mathcal{S}(P,S) = \begin{cases} \mathcal{S}(p,S) \vee \\ (\overline{\bigvee_{\langle \mathsf{exit}\ \sigma_i, S_j\rangle \in \mathcal{E}(p,S) \mid \sigma_i \neq T} S_j} \\ \quad \wedge \bigvee_{\langle \mathsf{exit}\ \sigma_i, S_j\rangle \in \mathcal{E}(p,S) \mid \sigma_i = T} S_j) \end{cases}$	(17)
$p; q$	$\mathcal{E}(P,S) = \mathcal{E}(p,S) \cup \mathcal{E}(q, \mathcal{S}(p,S))$ $\mathcal{S}(P,S) = \mathcal{S}(q, \mathcal{S}(p,S))$	(18)
loop p **end**	$\mathcal{E}(P,S) = \mathcal{E}(p, S \vee \mathcal{S}(p,S))$ $\mathcal{S}(P,S) = \mathsf{false}$	(19)
signal S **in** p **end**	$\mathcal{E}(P,S) = \mathcal{E}(p,S)$ $\mathcal{S}(P,S) = \mathcal{S}(p,S)$	(20)
$p \mid\mid q$	$\mathcal{E}(P,S) = \mathcal{E}(p,S) \cup \mathcal{E}(q,S)$ $\mathcal{S}(P,S) = \mathsf{false}$	(21)

Algorithm 2: Equations to determine replacement expressions for signals: $\mathcal{E}$ collects the signal state context for signal emissions, $\mathcal{S}$ returns the signal state context of terminating statements, P is the given program fragment shown on the left, and S is the state expression in the current program context.

that case the innermost trap masks the outer trap definition, effectively reversing the priorities. This is similar to local signals masking global signals of the same name. Duplicate trap identifiers are not checked explicitly in rule (17). That problem is deferred to the Esterel parser.

Rule (19) (loop) is interesting in that it is the only one to evaluate its body p twice for different context states. This is needed to cover instantaneous restarts of the loop body. The first evaluation is performed by function $\mathcal{S}$ to derive the context state of the terminating body. That expression is added to the current context state to retrieve the signal emissions of p in the second run with function $\mathcal{E}$.

Rule (21) (parallel) returns false for the termination context of all parallel statements, because the termination of



(a) present+ (b) present−

Figure 7: Logical behavioral semantics of the present statement [4].

parallel statements is assumed to be replaced by the scheme proposed in Figure 5. It replaces the implicit termination of parallel statements by explicit trap exception handling.

Rules for suspend statements are not given in Algorithm 2, because they are assumed to be substituted by means of other kernel statements in Step (2d) of the transformation algorithm.

3.3. Example transformations

To illustrate the transformation algorithm we will now apply it to some example Esterel programs.

3.4. Transforming PAUSE_CYC

Applying the algorithm to the example PAUSE_CYC in Figure 2(a) yields the acyclic program PAUSE_ACYC in Figure 2(c).

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

Steps (2a) to (2d) do not apply to PAUSE_CYC. Step (2e) is skipped here for brevity. This is valid here, because the termination of the parallel statement does not influence the cycle.

Step (2f): 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. Signals A_ and B_ are added in Step (4).

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

Step (4): 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 (5a): the computation of replacement expressions for A_ and B_ according to Section 3.2 results in

$$\mathsf{A\_} = \mathsf{ST\_1} \wedge (\mathsf{B} \vee \mathsf{B\_}), \tag{22}$$

$$\mathsf{B\_} = \mathsf{ST\_0} \wedge (\mathsf{A} \vee \mathsf{A\_}). \tag{23}$$

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 equations accordingly.

Step (5b): in PAUSE_PREP, we arbitrarily select A_ as the signal to break the cycle.

Step (5c): to replace B_ in (22), substituting (23) into (22) results in

$$\mathsf{A\_} = \mathsf{ST\_1} \wedge (\mathsf{B} \vee (\mathsf{ST\_0} \wedge (\mathsf{A} \vee \mathsf{A\_}))). \tag{24}$$

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 (5d): replacing the self-reference of signal A_ on the right-hand side of (24) by false (absent) yields:

$$\mathsf{A\_} = \mathsf{ST\_1} \wedge (\mathsf{B} \vee (\mathsf{ST\_0} \wedge \mathsf{A})). \tag{25}$$

Similarly, for A_ = true (present),

$$\mathsf{A\_} = \mathsf{ST\_1} \wedge (\mathsf{B} \vee \mathsf{ST\_0}). \tag{26}$$

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

Step (5e): finally we are ready to break the cycle in PAUSE_PREP. For that, we have to replace the signal selected in Step (5b)—in the cycle—by one of the expressions computed in Step (5d), which does not use any of the cycle signals, without changing the meaning of the program. Substituting (26), the simpler of these expressions, for A_ in PAUSE_PREP yields the now acyclic program PAUSE_ACYC shown in Figure 2(c).

The program PAUSE_OPT in Figure 2(d) is optimized on the insight that ST_1 cannot be present at the point of its test.

3.4.1. Transforming the token ring arbiter

Searching for signal dependencies in the program TR3_CYC from Figure 8(a) according to the algorithm presented in Section 2 yields the following cycle:

$$C = \langle \mathsf{P1}, \mathsf{P2}, \mathsf{P3} \rangle. \tag{27}$$

The computation of replacement expressions yields the following results for the cycle signals:

$$\begin{aligned} \mathsf{P1} &= (\mathsf{ST\_0} \vee \mathsf{ST\_7}) \wedge (\mathsf{T3} \vee \mathsf{P3}) \wedge \overline{\mathsf{R3}}, \\ \mathsf{P2} &= (\mathsf{ST\_0} \vee \mathsf{ST\_1}) \wedge (\mathsf{T1} \vee \mathsf{P1}) \wedge \overline{\mathsf{R1}}, \\ \mathsf{P3} &= (\mathsf{ST\_0} \vee \mathsf{ST\_4}) \wedge (\mathsf{T2} \vee \mathsf{P2}) \wedge \overline{\mathsf{R2}}. \end{aligned} \tag{28}$$

We may select signal P1 to break the cycle. Now the cycle signals P2 and P3 are substituted in the equation for P1:

$$\begin{aligned} \mathsf{P1} &= (\mathsf{ST\_0} \vee \mathsf{ST\_7}) \wedge (\mathsf{T3} \vee \mathsf{P3}) \wedge \overline{\mathsf{R3}} \\ &= (\mathsf{ST\_0} \vee \mathsf{ST\_7}) \\ &\quad \wedge (\mathsf{T3} \vee ((\mathsf{ST\_0} \vee \mathsf{ST\_4}) \wedge (\mathsf{T2} \vee \mathsf{P2}) \wedge \overline{\mathsf{R2}})) \wedge \overline{\mathsf{R3}} \\ &= (\mathsf{ST\_0} \vee \mathsf{ST\_7}) \\ &\quad \wedge (\mathsf{T3} \vee ((\mathsf{ST\_0} \vee \mathsf{ST\_4}) \\ &\qquad \wedge (\mathsf{T2} \vee ((\mathsf{ST\_0} \vee \mathsf{ST\_1}) \\ &\qquad\quad \wedge (\mathsf{T1} \vee \mathsf{P1}) \wedge \overline{\mathsf{R1}})) \wedge \overline{\mathsf{R2}})) \wedge \overline{\mathsf{R3}}. \end{aligned} \tag{29}$$

```
module TR3_CYC:
input R1, R2, R3;
output G1, G2, G3;
signal P1, P2, P3, T1, T2, T3
in
    emit T1
  ||
    loop % Station 1
      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 % Station 2
    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 % Station 3
    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
```

(a)

```
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 in
  emit ST_0;
  signal P1, P2, P3, 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
          emit P1
        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
  ]
  end signal
end signal
end module
```

(b)

FIGURE 8: Token ring arbiter with three stations: (a) Esterel implementation [5] with expanded run modules, (b) acyclic transformation by replacing testing of signal P1.

```
module TrapPause:
signal A, B in
  loop
    trap pausais in
      loop
          present A then
            emit B
          end present;
          exit pausais
        ||
          pause;
          present B then
            emit A
          end present
      end loop
    end trap;
    pause
  end loop
end signal
end module
```

(a)

```
module TrapPause:

signal ST_0, ST_1, ST_2, ST_3,
                ST_4, ST_5 in
  emit ST_0;
  signal P_i, P_j in
    signal A, B in
      loop
        trap pausais in
          loop
            trap P in
              [
                present False then
                  emit B
                end present;
                exit pausais;
                loop
                  emit P_i;
                  pause; emit ST_1
                end loop
              ||
                pause; emit ST_2;
                present False then
                  emit A
                end present;
              ||
                loop
                  emit P_j;
                  pause; emit ST_3
                end loop
              ||
                loop
                  present
            [P_i and (ST_2 or ST_3)] then
                    exit P
                  end present;
                  pause; emit ST_4
                end loop
              ]
            end trap
          end loop
        end trap;
        pause; emit ST_5
      end loop
    end signal
  end signal
end signal

end module
```

(b)

FIGURE 9: TrapPause: resolving two cycles between A and B and B on itself: (a) original program, (b) cycles on A and B resolved.

Equation (29) now 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 true yields

$$\begin{aligned} \mathsf{P1} = {} & (\mathsf{ST\_0} \vee \mathsf{ST\_7}) \\ & \wedge (\mathsf{T3} \vee (\mathsf{ST\_0} \vee \mathsf{ST\_4}) \\ & \quad \wedge (\mathsf{T2} \vee (\mathsf{ST\_0} \vee \mathsf{ST\_1}) \wedge \overline{\mathsf{R1}}) \wedge \overline{\mathsf{R2}}) \\ & \wedge \overline{\mathsf{R3}}. \end{aligned} \tag{30}$$

Equation (30) is applied when transforming TR3_CYC into the acyclic program TR3_ACYC shown in Figure 8(b).

3.4.2. Other example transformations

Figures 9 to 13 list further example transformations that illustrate interesting cases and some subtleties of the transformation problem. Figure 9 lists a program that contains one cycle between signals A and B and another, more complex one involving B across the termination of parallel threads and an additional restart by a loop. Therefore the cycle includes the helper signal P_j which is introduced to handle the termination of parallel threads.

The program in Figure 10 consists of two parallel threads with a dependency from A to B and vice versa. Several pause statements ensure constructiveness of the program by executing the two dependencies in different instants.

Figure 11 contains program where a cycle is present between signals S and T. Both are contained in a pair of present/emit statements with reversed roles. Constructiveness is ensured by mutually exclusive execution of both present statements.

Figure 12 contains a program that features the suspend statement and an input signal A which is part of the cycle. The suspend statement is replaced by a trap/loop construction and the cycle is moved from the input signal to a new internal signal A_2. The parallel statement is not addressed here because it is not part of the cycle.

In the program in Figure 13, a cycle on signal A is controlled by the termination of two parallel threads. Constructiveness is ensured by separation of emission and test of A in separate instants. The entire parallel part is enclosed in an additional loop structure. This results in an additional cyclic dependency in the termination control of the parallel part. Here constructiveness is ensured by executing a pause statement in at least one parallel thread and therefore separating start and termination of the parallel part into different instants.

4. OPTIMIZATIONS

So far, we have presented the transformation algorithm and equations for replacement expressions in its basic form without any additional improvements. This section outlines some possible optimizations. For space considerations, we here limit the presentation to short descriptions of the most essential optimizations. Further explanations, including examples, and additional optimizations are given elsewhere [19].

Expression for present

The most important optimization refines the treatment of the present statement in (13). Consider the following

```
module PausePause:

signal A, B in
    pause;
    pause;
    present A then
      emit B
    end present
  ||
    pause;
    present B then
      emit A
    end present
end signal

end module
```

(a)

```
module PausePause:

signal ST_0, ST_1, ST_2, ST_3 in
  emit ST_0;
  signal A, B in
    [
      pause; emit ST_1;
      pause; emit ST_2;
      present False then
        emit B
      end present
    ||
      pause; emit ST_3;
      present B then
        emit A
      end present
    ]
  end signal
end signal

end module
```

(b)

FIGURE 10: PausePause: resolving a cycle between A and B: (a) original program, (b) resolved cycle.

```
module TrapParallel:
input A, B;

signal S, T in
  trap X in
    [
      present A then
        exit X
      end present
    ||
      present B then
        pause
      end present
    ];
    present S then
      emit T
    end present;
    halt
  end trap;
  present T then
    emit S
  end present
end signal

end module
```

(a)

```
module TrapParallel:
input A, B;

signal ST_0, ST_1, ST_2,
       ST_3, ST_4, ST_5 in
  emit ST_0;
  signal P_i, P_j in
    signal S, T in
      trap X in
        trap P in
          [
            present A then
              exit X
            end present;
            loop
              emit P_i;
              pause; emit ST_1
            end loop
          ||
            present B then
              pause; emit ST_2
            end present;
            loop
              emit P_j;
              pause; emit ST_3
            end loop
          ||
            loop
              present [P_i and P_j] then
                exit P
              end present;
              pause; emit ST_4
            end loop
          ]
        end trap;
        present False then
          emit T
        end present;
        loop
          pause; emit ST_5
        end loop
      end trap;
      present False then
        emit S
      end present
    end signal
  end signal
end signal

end module
```

(b)

FIGURE 11: TrapParallel: resolving two cycles on S and T: (a) original program, (b) resolved cycles.

program fragment:

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

The application of the rules listed in Algorithm 2 would result in a replacement expression for signal $C = (ST\_3 \wedge S) \vee (ST\_3 \wedge \overline{S})$. It is obvious that this can be simplified to $C = ST\_3$, since neither present branch has an influence on the emission of C. Or more generally, see Figure 9.

Equation (13)

$$\mathcal{S}((S?p,q), C) = \mathcal{S}(p, C \wedge S) \vee \mathcal{S}(q, C \wedge \overline{S}) \tag{31}$$

can be simplified to

$$\mathcal{S}((S?p,q), C) = C \tag{32}$$

```
module SuspendPause:
input A;
output B;

    suspend
      emit A;
      pause
    when A;
    emit B
  ||
    present B then
      emit A
    end present

end module
```

(a)

```
module SuspendPause:
input A;
output B;

    emit A;
    trap new_trap_0 in
      loop
        pause;
        present A else
          exit new_trap_0
        end
      end
    end;
    emit B
  ||
    present B then
      emit A
    end present

end module
```

(b)

```
module SuspendPause:
input A;
output B;

signal ST_0, ST_1, A_2 in
  emit ST_0;
  [
    emit A;
    trap new_trap_0 in
      loop
        pause; emit ST_1;
        present [not (A or A_2)] then
          exit new_trap_0
        end present
      end loop
    end trap;
    emit B
  ||
    present [ST_1 and not A] then
      emit A_2
    end present
  ]
end signal

end module
```

(c)

Figure 12: SuspendPause: resolving a cycle between A and B: (a) original program, (b) after substituting suspend, (c) resolved cycle.

if

$$(\mathcal{S}(p, C \wedge S) = C \wedge S) \wedge (\mathcal{S}(q, C \wedge \overline{S}) = C \wedge \overline{S}) \qquad (33)$$

holds.

Termination of parallel statements

The general transformation algorithm for parallel statements calls for instrumentation to detect the termination of parallel statements at run time. It involves additional signals, sustain statements, and a test for those signals (see Figure 5). These additions are not needed if the parallel statement cannot terminate at all at run time, for example, one thread contains a loop statement on the top level. This is the case for the token ring arbiter (Figure 8(a)).

Parallel termination and exceptions

The interaction of parallel threads with exception handling leaves significant room for optimization. Currently each pause statement contained in a trap block is considered as a point where control is potentially handed over to the end of the trap block. As an optimization this could be limited to actually (statically) reachable exceptions at pause statements.

Substitution of suspend

Step (2d) of the algorithm calls for a replacement of all suspend statements, for the reasons outlined in Section 3.1. However, the transformation algorithm does not necessarily fail on all programs containing suspend statements. If the suspend statements are not part of the cycle, then they pose no problem. Furthermore, participation in the cycle can be tolerated if the suspend guard predicates can be replaced by noncyclic expressions. Figure 14 contains such a successful example. Signal B′ in the suspend guard is replaced by a noncyclic expression not depending on signals emitted inside a suspend environment.

Replacing state signal tests by constants

Another optimization is 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 2(d) by taking the reachable presence status of the signals ST_0 and ST_1 into account.

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 tests of cycle breaking signal

If the signal σ_p that is selected in Step (5b) to break the cycle is not tested outside of the cycle, this means that after

```
module ParallelPause:
inputoutput A;
loop
  [
      present [A] then
        pause
      end present
    ||
      pause
  ];
  emit A;
  pause
end loop
end module
```

(a)

```
module ParallelPause:
input A_in;
output A_out;
loop
  [
    signal P_i, P_j in
      trap P in
          present [A_in or A_out] then
            pause
          end present;
          loop emit P_i; pause; end
        ||
          pause;
          loop emit P_j; pause; end
        ||
          loop
            present [P_i and P_j] then
              exit P
            end present;
            pause
          end loop
        end trap
    end signal;
  ];
  emit A_out;
  pause
end loop
end module
```

(b)

```
module ParallelPause:
input A_in;
output A_out;

signal ST_0, ST_1, ST_2, ST_3, ST_4, ST_5, ST_6 in
    emit ST_0;
    loop
      signal P_i, P_j in
        trap P in
          [
            present [A_in or (ST_0 or ST_5) and (ST_1 or ST_0
and not A_in or ST_2 or ST_1 or (ST_0 or ST_6) and not A_in
or ST_2) and P_j or (ST_0 or ST_6 or ST_5) and (ST_1 or ST_0
and not A_in or ST_2 or ST_1 or (ST_0 or ST_6) and not A_in
or ST_2) and P_j] then
              pause; emit ST_1
            end present;
            loop
              emit P_i;
              pause; emit ST_2
            end loop
          ||
            pause;
            emit ST_3;
            loop
              emit P_j;
              pause; emit ST_4
            end loop
          ||
            loop
              present [(ST_1 or ST_0 and not A_in or ST_2
or ST_1 or (ST_0 or ST_6) and not A_in or ST_2) and P_j] then
                exit P
              end present;
              pause; emit ST_5
            end loop
          ]
        end trap
    end signal;
    emit A_out;
    pause; emit ST_6
  end loop
end signal

end module
```

(c)

FIGURE 13: ParallelPause: resolving two cycles with A_out on itself and P_i on itself: (a) original program, (b) preprocessed program adding parallel termination control and separating "inputoutput A" into "input A_in" and "output A_out," (c) resolved cycles.

replacing the tests for σ_p within the cycle (Step 5e) by E_p^{**}, the signal σ_p is not tested anywhere in the program. One can therefore eliminate its emission. This is the case for signal P1 in the token ring arbiter (Figure 8(b)). P1 is emitted but not tested anymore in the program. P1 and its emission can be removed. Emissions of output signals must not be removed, because emissions of them must reach the outside interface.

Lifting of locally defined signals

The transformation algorithm (Figure 4) states in Step (2c) the relocation of local signal definitions up to the top level. The reason for this step lies in the introduction of replacement expressions. These expressions may transport references to local signals out of their respective scope. Relocation of *all* local signals is certainly not strictly necessary. Only those signals which are referenced in a replacement expressions must be relocated. It would be more efficient to detect possible conflicts with replacement expressions first and then to relocate the problematic signals.

5. IMPLEMENTATION AND EXPERIMENTAL RESULTS

To validate the transformation and to measure its effectiveness, we have implemented the transformation presented in Section 3 as an extension to the Columbia Esterel compiler

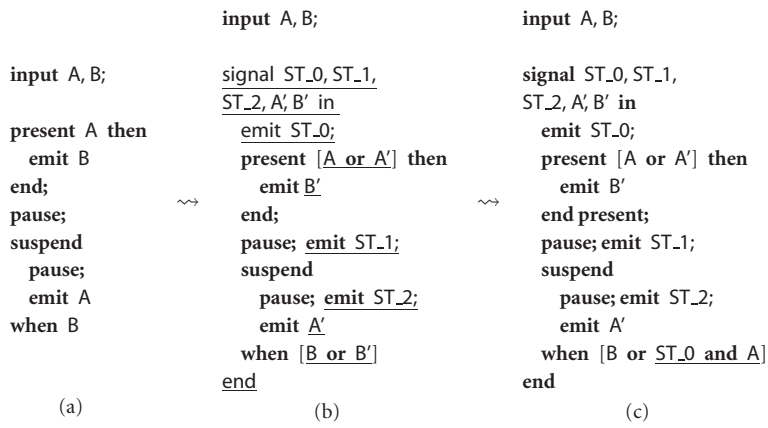


FIGURE 14: Successful resolving of a cyclic dependency involving present and suspend statements: (a) original program, (b) preprocessing by introduction of state signals and signal renaming, (c) replacement of tests for B′ in the suspend guard by an expression.

(CEC). The CEC is used for file access, parsing, and partial dismantling into kernel statements. The implementation handles cycle detection and the transformation algorithm for pure signals. Certain parts of the preprocessing stage like handling of suspend, termination of parallel, and lifting of local signals must still be performed manually. The first optimization explained in Section 4 regarding present is implemented, but not the other ones.

For an experimental evaluation, we have tested several variants of the token ring arbiter. They are named TR3 to TR1000 to indicate the number of network stations in each example.

Program TR10p is a special case; 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 original cyclic and in the transformed acyclic version with the following six different compilation techniques.

v5-L: the publicly available Esterel compiler v5.92 [11, 25] is used 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.

The v5 compiler is able to handle constructive Esterel programs with cyclic dependencies. For such programs the full causality analysis must be activated by the option `-causal`. The cyclic parts of the program are resynthesized resulting in a growth in program size compared to noncyclic programs.

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 from Esterel Technologies [27]) is used here in version v7_10i8 to compile acyclic code based on sorted equations, as the v5 compiler. v7 is not able to handle cyclic programs. Thus it can only be applied to the transformed cyclic programs.

v7-O: the former compiler, but with option `-O`, applies some circuit optimizations to reduce program size and run time.

CEC: the Columbia Esterel compiler (release 0.3) [12] produces event-driven C code, which is generally quite fast and compact. The CEC is not able to handle cyclic programs either.

CEC-g: the CEC with the option `-g` produces code using computed goto targets (an extension to ANSI-C offered by GCC-3.3 [28]) to reduce the run time 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 provides an iteration over 10 000 000 times for the reaction function. 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, 256 KB cache, 1 GB main memory).

Table 1 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 run time 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 run time 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 run time and binary sizes. And in fact the v5 compiler produces functionally identical code for the original and transformed programs. Only the different file names result in a small difference in binary sizes.

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. The best values across the compilers are shown bold.

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 includes the sizes of the compiled binaries too. All compilers produce code of similar sizes, 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." 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 better than the CEC compiler with computed goto optimization. Nevertheless the huge binary produced by the v5 compiler in automaton mode limits its usefulness.

A condensed presentation of the measurements from Table 1 is given in Table 2, which compares the fastest code for the cyclic programs compiled by the v5 compiler to the fastest code for the transformed acyclic programs. For each test program the relative reduction in run time is listed. In no case does the transformation slow down execution, and in two of the three cases considered here, there is a considerable gain in execution speed by enabling the use of the v7 and CEC compilers on cyclic programs.

As an indication of the cost of the transformation algorithm in terms of source code increase, Table 3 lists 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 a constant factor with respect to the arbiter network size. The current transformation times show a sub-quadratic effort for the transformation.

Table 2: Relative run time reduction from the fastest cyclic version to the fastest version for the acyclic transformation, with reduction = $100\% * (1 - \min(T_{\text{acyclic}})/\min(T_{\text{cyclic}}))$.

	TR3	TR10	TR10p
$\min(T_{\text{cyclic}})$	0.90	2.58	5.26
$\min(T_{\text{acyclic}})$	0.53	1.87	5.26
reduction	41%	28%	0%

6. CONCLUSIONS AND FUTURE WORK

This paper has 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 achieved by the compilers that can handle cyclic by themselves. Furthermore, the experiments have shown that the code transformation proposed here can even improve code quality produced by compilers that can already handle cyclic programs. As noted in Section 2, it depends on the compiler what cycles it actually does detect. It therefore may be possible that a compiler that does a very weak analysis detects cycles that we do not detect; however, our approach is fairly conservative already, and we do not expect to miss a significant fraction of cycles for any compiler.

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). Nevertheless the transformation does not depend on a specific implementation of constructiveness. The assurance of constructiveness itself is sufficient for the transformation to work.

The transformation introduces new signals and expands the original program. However, most of this disappears again after optimizations. In fact, the net effect of the transformation is often a reduction of code size, as the static analysis may remove some unreachable code. In a certain way, the transformation performs a partial evaluation of the given program.

The synchronous data flow oriented programming language Lustre is affected by cyclic dependencies too. In Figure 15(a) a Lustre implementation of the token ring arbiter is shown as an example. This program is rejected by Lustre compilers because of cyclic dependencies on streams pass1, pass2, and pass3. The replacement expression (30) (without state signals) is used in Figure 15(b) to manually produce an acyclic derivation of the original program which is accepted by Lustre compilers. This example indicates a possible application of the cycle transformation algorithm

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

	TR3	TR10	TR50	TR100	TR500	TR1000	TR10p
Original program size	1565	3705	16348	32159	162959	326470	5765
After module expansion	1370	4391	22031	44092	224892	450903	6995
After cycle transformation	2108	6856	34804	69920	359788	723804	9736
Ratio after/before transformation	1.53	1.56	1.58	1.59	1.60	1.61	1.39
Transformation time (seconds)	0.05	0.07	0.27	0.57	5.18	17.5	0.11

```
node three_stations( request1 : bool;
      request2: bool; request3 : bool)
   returns (grant1 : bool;
      grant2 : bool; grant3 : bool);

var
  pass1 : bool; pass2 : bool; pass3 : bool;
  token1 : bool; token2 : bool; token3 : bool;
  token1_or_pass1 : bool;
  token2_or_pass2 : bool;
  token3_or_pass3 : bool;

let
  /* Station 1 */
  token1_or_pass1 = token1 or pass1;
  grant1 = request1 and token1_or_pass1;
  pass2 = not(request1) and token1_or_pass1;
  token2 = pre ((true) -> (token1));

  /* Station 2 */
  token2_or_pass2 = token2 or pass2;
  grant2 = request2 and token2_or_pass2;
  pass3 = not(request2) and token2_or_pass2;
  token3 = pre (( false ) -> (token2));

  /* Station 3 */
  token3_or_pass3 = token3 or pass3;
  grant3 = request3 and token3_or_pass3;
  pass1 = not(request3) and token3_or_pass3;
  token1 = pre (( false ) -> (token3));
tel;
```

(a)

```
node three_stations( request1 : bool;
      request2 : bool; request3 : bool)
   returns (grant1 : bool;
      grant2 : bool; grant3 : bool);

var
  pass1 : bool; pass2 : bool; pass3 : bool;
  token1 : bool; token2 : bool; token3 : bool;
  token1_or_pass1 : bool;
  token2_or_pass2 : bool;
  token3_or_pass3 : bool;

let
  /* Station 1 */
  token1_or_pass1 = token1 or
  (token3 or (token2 or not request1) and
  not request2) and not request3;
  grant1 = request1 and token1_or_pass1;
  pass2 = not(request1) and token1_or_pass1;
  token2 = pre ((true) -> (token1));

  /* Station 2 */
  token2_or_pass2 = token2 or pass2;
  grant2 = request2 and token2_or_pass2;
  pass3 = not(request2) and token2_or_pass2;
  token3 = pre (( false ) -> (token2));

  /* Station 3 */
  token3_or_pass3 = token3 or pass3;
  grant3 = request3 and token3_or_pass3;
  pass1 = not(request3) and token3_or_pass3;
  token1 = pre (( false ) -> (token3));
tel;
```

(b)

Figure 15: Lustre implementation of the token ring arbiter with three stations: (a) cyclic implementation, (b) acyclic transformation by replacing testing of signal pass1.

to Lustre programs. Unfortunately a notion of constructive Lustre programs has not been established yet. The transformation needs the assurance of constructiveness to be able to produce programs with the same behavior as the original programs.

Regarding future work, there are numerous optimizations possible, some of which were presented in Section 4. Some of these might be helpful for Esterel programs in general, not just as a post-processing step to the transformation proposed here.

The transformation algorithm as described here needs the constructiveness of input programs as a precondition. The algorithm exploits the constructiveness while replacing self-referencing signal names in replacement expressions. However, it appears that the computation of replacement expressions presented here may also be used to facilitate constructiveness analysis in the first place [19].

ACKNOWLEDGMENTS

We would like to thank Stephen Edwards for providing his CEC compiler as a solid foundation to build upon, and we would like to thank him and Dumitru Potop-Butucaru for fruitful discussions about different types of cyclic dependencies. Claus Traulsen has provided helpful feedback on earlier versions of this paper. We also thank the anonymous reviewers for their detailed and constructive comments, including suggesting the examples presented in Section 3.4.2.

REFERENCES

[1] A. Benveniste, P. Caspi, S. A. Edwards, N. Halbwachs, P. Le Guernic, and R. de Simone, "The synchronous languages 12 years later," *Proceedings of the IEEE*, vol. 91, no. 1, pp. 64–83, 2003, special issue on Embedded Systems.

[2] G. Berry and G. Gonthier, "The Esterel synchronous programming language: design, semantics, implementation," *Science of Computer Programming*, vol. 19, no. 2, pp. 87–152, 1992.

[3] N. Halbwachs, P. Caspi, P. Raymond, and D. Pilaud, "The synchronous data flow programming language LUSTRE," *Proceedings of the IEEE*, vol. 79, no. 9, pp. 1305–1320, 1991.

[4] G. Berry, "The Constructive Semantics of Pure Esterel," draft Book, 1999, ftp://ftp-sop.inria.fr/esterel/pub/papers/constructiveness3.ps.

[5] G. Berry, "The Esterel v5 Language Primer, Version v5_91," Centre de Mathématiques Appliquées Ecole des Mines and INRIA, 06565 Sophia-Antipolis, 2000, ftp://ftp-sop.inria.fr/esterel/pub/papers/primer.pdf.

[6] Esterel.org, Esterel history, http://www-sop.inria.fr/esterel.org/Html/History/History.htm.

[7] P. Pandya, "The saga of synchronous bus arbiter: on model checking quantitative timing properties of synchronous programs," *Electronic Notes in Theoretical Computer Science*, vol. 65, no. 5, pp. 894–908, 2002.

[8] J. Lukoschus and R. von Hanxleden, "Removing cycles in Esterel programs," in *Proceedings of International Workshop on Synchronous Languages, Applications and Programming (SLAP '05)*, F. Maraninchi, M. Pouzet, and V. Roy, Eds., Edinburgh, Scotland, UK, April 2005.

[9] S. Malik, "Analysis of cyclic combinational circuits," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 13, no. 7, pp. 950–956, 1994.

[10] T. R. Shiple, G. Berry, and H. Touati, "Constructive analysis of cyclic circuits," in *Proceedings of European Design and Test Conference (ED&TC '96)*, pp. 328–333, Paris, France, March 1996.

[11] G. Berry, A. Bouali, Y. Bres, et al., *The Esterel v5_91 System Manual*, INRIA, June 2000, http://www-sop.inria.fr/esterel.org/.

[12] "CEC: The Columbia Esterel Compiler," http://www1.cs.columbia.edu/~sedwards/cec/.

[13] S. A. Edwards, "An Esterel compiler for large control-dominated systems," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 21, no. 2, pp. 169–183, 2002.

[14] E. Closse, M. Poize, J. Pulou, P. Venier, and D. Weil, "SAXO-RT: interpreting Esterel semantic on a sequential execution structure," *Electronic Notes in Theoretical Computer Science*, vol. 65, no. 5, pp. 864–878, 2002.

[15] D. Potop-Butucaru, *Optimizations for faster simulation of Esterel programs*, Ph.D. thesis, Ecole des Mines de Paris, Paris, France, November 2002.

[16] L. H. Yoong, P. Roop, and Z. Salcic, "Compiling Esterel for distributed execution," in *Proceedings of International Workshop on Synchronous Languages, Applications, and Programming (SLAP '06)*, Vienna, Austria, March-April 2006.

[17] X. Li, M. Boldt, and R. von Hanxleden, "Mapping Esterel onto a multi-threaded embedded processor," in *Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '06)*, pp. 303–314, San Jose, Calif, USA, October 2006.

[18] S. A. Edwards, "Making cyclic circuits acyclic," in *Proceedings of the 40th Design Automation Conference (DAC '03)*, pp. 159–162, ACM Press, Anaheim, Calif, USA, June 2003.

[19] J. Lukoschus, *Removing cycles in the Esterel programs*, Ph.D. thesis, Department of Computer Science, Christian-Albrechts-Universität Kiel, Kiel, Germany, July 2006, http://e-diss.uni-kiel.de/tech-fak.html.

[20] K. Schneider and M. Wenz, "A new method for compiling schizophrenic synchronous programs," in *Proceedings of International Conference on Compilers, Architecture, and Synthesis for Embedded Systems (CASES '01)*, pp. 49–58, ACM, Atlanta, Ga, USA, November 2001.

[21] O. Tardieu, "Goto and concurrency introducing safe jumps in Esterel," in *Proceedings of the 3rd International Workshop on Synchronous Languages, Applications, and Programs (SLAP '04)*, Barcelona, Spain, March 2004.

[22] F. Boussinot, "SugarCubes implementation of causality," Research Report RR-3487, INRIA, Le Chesnay Cedex, France, September 1998.

[23] K. Schneider, J. Brandt, T. Schüele, and T. Tüerk, "Improving constructiveness in code generators," in *Proceedings of International Workshop on Synchronous Languages, Applications and Programming (SLAP '05)*, Edinburgh, Scotland, UK, April 2005.

[24] "Estbench Esterel Benchmark Suite," http://www1.cs.columbia.edu/~sedwards/software/estbench-1.0.tar.gz.

[25] Esterel web, http://www-sop.inria.fr/esterel.org/.

[26] R. Cytron, J. Ferrante, B. K. Rosen, M. N. Wegman, and F. K. Zadeck, "Efficiently computing static single assignment form and the control dependence graph," *ACM Transactions on Programming Languages and Systems*, vol. 13, no. 4, pp. 451–490, 1991.

[27] Esterel Technologies, Company homepage, http://www.esterel-technologies.com/.

[28] Free Software Foundation, "GCC—The GNU Compiler Collection," http://gcc.gnu.org/.