Removing Cycles in Esterel Programs

Dissertation

zur Erlangung des akademischen Grades
Doktor der Naturwissenschaften
(Dr. rer. nat.)
der Technischen Fakultät
der Christian-Albrechts-Universität zu Kiel

Jan Lukoschus

Kiel
2007

1. Gutachter	Reinhard von Hanxleden
2. Gutachter	Stephen A. Edwards
Datum der mündlichen Prüfung	20. Juli 2006

Abstract

Programming embedded and real-time systems demands different methodologies and programming languages than conventional applications focused mostly on generic computations. Restrictions on reaction times and deterministic behavior that are hard to implement in common programming languages like C or Java, motivate to use specialized programming languages. Esterel is such a specialized programming language. It natively supports exceptions, suspension, abortion, and concurrency. Communication between threads and the environment is performed by signals.

The execution is divided into temporal steps, called *instants*, which conceptually take no time. Communication with the environment takes place at the start of each instant for input signals and at the end for output signals. From this execution model follows that interface signals can either be set or not set in an instant. This is extended to the handling of internal signals, and if a signal is set in the course of an instant, it is considered set from the beginning of that instant. Therefore any tests on the signal status must be scheduled after all possible settings of that signal. Programming languages which are based on such a handling of signals are called *synchronous* languages. The formalization of such a schedule is the *constructive* derivation of signal states, which leads to a deterministic behavior even in the context of multiple parallel threads.

Synchronous programs may contain cyclic signal interdependencies. This prohibits a static scheduling, which limits the choice of available compilation techniques for such programs. This thesis proposes an algorithm which, given a constructive synchronous program, performs a semantics-preserving source-level code transformation that removes cyclic signal dependencies. This makes it possible to compile originally cyclic programs using for example the existing efficient compilers that implement event-driven simulators.

The transformation is divided into two major parts: detection of cycles and iterative resolving of the cycles. This thesis provides details of both parts and suggests possible further optimizations. The detection of cycles is based on the derivation of signal dependencies in the Esterel program. Cycles are defined as cyclic paths in the set of signal dependencies. Esterel compilers differ in some details in what they consider cyclic. Here the v5 compiler is taken as a reference for cyclic dependencies. Nevertheless the current implementation makes conservative assumptions on signal dependencies to cover the cycles found by other Esterel compilers as well.

The transformation itself is independent of the Esterel compiler actually used; it replaces

a cycle signal by a signal expression involving no other cycle signals. This expression is computed by deriving the context state expressions for the cycle signal and by the iterative replacement of other cycle signals by their respective state expressions. The resulting expression may contain a dependency from the signal to itself, however the constructiveness of the source program rules out any influence of the signal on itself, and therefore the signal can be removed from the expression by replacing it with an arbitrary, constant truth value.

The transformation algorithm is implemented as an additional module to the Columbia Esterel Compiler for validation of the correctness of the transformation, quantification of the cost in code size growth, and evaluation of different transformation variants. 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 codes as well. Furthermore, experiments showed that the code transformation proposed here can even improve code quality produced by compilers that can already handle cyclic programs, such as the net-list approach employed by the v5 compiler. It also turns out that the compilation itself can be sped up by transforming cyclic programs into acyclic ones first.

To make the transformation algorithm more efficient, an addition to the Esterel language is suggested making the internal state signals of the runtime system visible to the Esterel level. This extension would eliminate the need to introduce synthetic state signals as part of the transformation.

As a byproduct of the transformation method proposed here, the analysis on constructiveness itself can be improved by using parts of the transformation algorithm.

Contents

Contents v

List of Figures ix

1 Introduction 1
- 1.1 Contribution of this Thesis . . . 1
- 1.2 Related Work . . . 2
- 1.3 Overview . . . 5
- 1.4 Acknowledgments . . . 6

2 The Esterel Language 7
- 2.1 Programming Reactive Systems . . . 8
- 2.2 Basic Structure of Esterel . . . 10
- 2.3 Overview of Esterel Statements . . . 12
  - 2.3.1 Kernel Statements . . . 12
  - 2.3.2 Derived Statements . . . 16
- 2.4 Reactivity, Determinism, Constructiveness . . . 19
  - 2.4.1 Logical Behavioral Semantics . . . 19
  - 2.4.2 Constructive Behavioral Semantics . . . 23
  - 2.4.3 Constructive Circuit Semantics . . . 25

3 Cyclic Dependencies 29
- 3.1 Non-Constructive Cycles . . . 29
- 3.2 Constructive Cycles . . . 31
- 3.3 Variants of Cyclic Dependencies . . . 34
- 3.4 Finding Cyclic Dependencies . . . 37

3.4.1 Algorithm to Identify Signal Dependencies 41
3.4.2 Searching for Cycles in Signal Dependencies 48

4 Program Transformation 51
4.1 The Base Transformation Algorithm 51
4.1.1 Cost of the Transformation Algorithm 63
4.2 Computing the Replacement Expressions 64
4.2.1 Relation to the Circuit Transformation 67
4.3 Extending Esterel to Handle suspend/abort 68
4.4 Example Transformations 70
4.4.1 Transforming PAUSE_CYC 70
4.4.2 Transforming the Token Ring Arbiter 72
4.4.3 Transforming Cycles Over Parallel Termination 77
4.4.4 Multiple pause Statements 78
4.4.5 Suspend 78
4.5 Proposals for Constructiveness Analysis 80
4.5.1 Substitution of Fixpoint Iteration 82
4.5.2 Temporal Induction 83

5 Cycles on Valued Signals 87
5.1 Introduction to Valued Signals in Esterel 87
5.2 Signal Dependencies on Valued Signals 88
5.3 Replacement Expressions for Valued Signals 90
5.4 Cycles on Pure Signals Broken by Valued Signals 91
5.5 Cycles on Internal Valued Signals 91
5.6 Preprocessing of Cyclic Valued input Signals 98
5.7 Transforming mejia.strl from Estbench 101

6 Optimizations 105
6.1 Replacement Expression for present 105
6.2 Termination of Parallel Statements 107
6.3 Interaction of Parallel Termination with Exceptions 107
6.4 Substitution of suspend and abort 108

6.5 Signal Renamings for Locally Defined Signals . . . 109
6.6 Replacing State Signal Tests by Constants . . . 109
6.7 Eliminating Emission of State Signals . . . 110
6.8 Absence of External Tests of Cycle Breaking Signal . . . 110
6.9 Simplification of External Tests . . . 111
6.10 Compiler-Specific Signal Dependencies . . . 111
6.11 Lifting of Locally Defined Signals . . . 111

7 Experimental Results 113
7.1 Synthesizing Software . . . 113
7.2 Synthesizing Hardware . . . 116

8 Assessment 119
8.1 Scope of the Cycle Identification . . . 119
8.2 Scope of the Transformation Algorithm . . . 120
8.3 Transformation of Non-Constructive Programs . . . 120
8.4 Accessing the Program State . . . 122

9 Conclusions and Future Work 125

A Example Transformations 127
A.1 present, pause . . . 128
A.2 Termination of parallel Threads . . . 130
A.3 Implementing the Token Ring Arbiter in Lustre . . . 132

Bibliography 133

List of Figures

2.1 Finite state machine implementation of the ABRO specification. 10

3.1 Invalid cyclic Esterel programs 30
3.2 Resolving a cycle 32
3.3 Circuit representation of the program PAUSE_CYC 33
3.4 False cyclic dependencies 33
3.5 Token Ring Arbiter with three stations 35
3.6 Ambiguous cyclic dependencies 36
3.7 Equations to determine signal dependencies, first part 38
3.8 Equations to determine signal dependencies, second part 39
3.9 Simplified equations for signal dependencies 40
3.10 Dependencies introduces by exception handling 46
3.11 Signal dependency extending outside the scope of a local signal 47
3.12 Finding (cyclic) signal dependencies 48
3.13 Algorithm to find a shortest cycle in signal dependencies 49

4.1 Notation summary. 52
4.2 Transformation algorithm, for pure signals. 53
4.3 Preserving the priorities between cascaded abort statements 56
4.4 Making the termination state of parallel threads visible to expressions . . 57
4.5 Program with potentially non-terminating iterative signal replacement . . 59
4.6 Replacing the signal test for A by its emission context. 64
4.7 Equations to determine replacement expressions for signals 65
4.8 Circuit translation of the present statement 67
4.9 Failed resolving of a cyclic dependency involving two suspend statements 68

4.10 Equations for replacement expressions in context of suspend/abort 71
4.11 Transformed non-cyclic Token Ring Arbiter 73
4.12 Treatment of cyclic dependencies crossing parallel termination 75
4.13 Continuation of Figure 4.12 76
4.14 Example requiring state signals which cannot be eliminated 79
4.15 Resolving a cycle in a cyclic program with suspend 80
4.16 Continuation of Figure 4.15 81
4.17 Algorithm to decide on the constructiveness of an Esterel program 84

5.1 Signal dependencies of emissions and tests on valued signals 89
5.2 Replacement expressions for valued signals 89
5.3 Example for emissions on valued signals 90
5.4 Cycle on pure signals broken by a valued signal 92
5.5 Esterel program with a cycle on internal valued signals 93
5.6 Acyclic transformation of the cyclic Esterel program in Figure 5.5. . . . 94
5.7 Optimized version of the transformation in Figure 5.6. 95
5.8 Esterel program with a cycle on valued input signals 99
5.9 Application of the transformation on a cutdown version of mejia.strl . . . 102
5.10 Acyclic transformation of the program in Figure 5.9. 103

6.1 Cyclic dependency resolved without iterative replacement 106
6.2 Resolving a cyclic dependency involving present and abort statements . . 107

Chapter 1

Introduction

One of the strengths of synchronous languages [1] is their deterministic semantics in the presence of concurrency. It is possible to write a synchronous program 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 cyclic programs. Hence, cyclic programs are currently only translatable by techniques that are relatively inefficient with respect to execution time, code size, or both. This thesis 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 thesis is on the synchronous language Esterel [7]; however, the concepts introduced here should be applicable to other synchronous languages as well, such as Lustre [20].

1.1 Contribution of this Thesis

The main contribution of this thesis is a method to specify and implement resolving of constructive cyclic dependencies as an Esterel source code transformation.

The proposed transformation makes use of the property of constructiveness to resolve cycles; however, unlike the approaches suggested earlier by Edwards [15, 16], it works on the source code level. Hence this makes it possible to compile originally cyclic programs using for example the existing efficient compilers that implement event-driven simulators. Furthermore, experimental results indicate that this transformation can also improve the code resulting from the techniques that can already handle cyclic programs, such as the net-list approach employed by the v5 compiler. It also turns out that the compilation itself can be sped up by transforming cyclic programs into acyclic ones first.

The basic transformation is defined on the basic kernel statements of Esterel to make the resulting programs compatible to existing Esterel compilers. Nevertheless certain aspects of the transformation would be considerably more efficient if internal state encoding registers of the compiled program would be accessible on the Esterel level. This improvement would require changes to Esterel compilers. These changes would aid further developments of the transformation algorithm as it would make it possible to access the state of `pause` statements embedded in non-kernel statements.

The transformation algorithm is worked out in detail for Esterel kernel statements and pure signals. Extensions to support valued signals are laid out by application on some representative examples. Cycles involving variables are not addressed here. Handling cyclic dependencies on valued signals needs to consider additional dependencies contained in value expressions, multiple emissions on the same signal coordinated by a combination function, and the inheritance of the value from previous instants if no current emission is executed for that signal.

As a byproduct of the transformation method proposed here, the analysis on constructiveness itself can be improved by using parts of the transformation algorithm. Classic constructiveness analysis consists of state space exploration by three-valued fixpoint iteration to derive reachable signal states from the initial state. If the fixpoint iteration converges without any unknown signal values left for all reachable states, then the program is considered constructive. The approach presented here tries to eliminate the need for a three-valued fixpoint iteration on all reachable signal states. The fixpoint iteration is used only in a preprocessing step to derive expressions for all signals describing their constructiveness with regard to the current signal state. The main benefit of the proposal presented here lies in the much simpler evaluation of a binary expression instead of a fixpoint iteration on three-valued signals for all program states.

1.2 Related Work

A number of different approaches for compiling Esterel programs into either software or hardware have been proposed and implemented. Historically the first Esterel compilers were based on automata as execution models.

The v2 compiler by Berry and Cosserat [6] used a LISP program to perform program transformations according to the operational semantics of Esterel. This literal implementation of the Esterel semantics proved to be much too slow and resource hungry to be of much practical use in embedded systems.

Gonthier [7] developed the v3 compiler, which avoids costly textual transformations at runtime. The Esterel program is compiled by transforming it into an *IC graph* (intermediate code graph). This graph is used to extract an automaton code which can be interpreted efficiently at runtime. The drawback of this approach is a possible code size explosion for the compiled program, because the automaton synthesis is based on the exploration of all program states.

The next generation of the Esterel compiler at Berry's group implemented a different strategy. While working on controller specifications in Esterel for programmable hardware, a scheme was developed to translate a subset of Esterel into logic gates [2]. This translation avoided the state space explosion by implementing concurrent activities in parallel logic gates. The ideas to synthesize hardware from Esterel got generalized into software synthesis by simulation of netlists of logic gates. This compilation strategy was the basis for the v4 compiler, and it resulted in a problem which is addressed in this thesis: cyclic dependencies. The automata synthesis explores all reachable program states, thereby abstracting away any internal dependencies of the program. Therefore cyclic dependencies pose no problem for the automata synthesis. The netlists fully reflect all dependencies of the program, and for a static schedule no cyclic dependencies must be contained in the program.

In the field of logic circuits these cyclic dependencies are known as feedbacks of outputs to the inputs in non-combinational circuits. Not all feedback loops result in unstable logic circuits, there exists the class of cyclic combinational circuits. Malik [28] 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.* [36] developed Malik's work further by applying optimizations and incorporating cycles including registers into the algorithm. An implementation into the v4 Esterel compiler lead to the v5 Esterel compiler, which is able to compile constructive cyclic Esterel programs into compact netlist code. However, this software simulation of circuits tends to be rather slow, as it simulates the entire circuit during each instant, irrespective of which parts of the circuit are currently active.

A third approach to synthesize software is to generate an event-driven simulator, which breaks the simulated circuit into a number of small functions that are conditionally executed [12, 14, 9]. These compilers tend to produce code that is compact and yet almost as fast as automata-based code. The drawback of these techniques is that so far, they rely on the existence of a static schedule and hence are limited to acyclic programs. One approach to overcome this limitation, which is described by Edwards [15], 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 basic synthesis approaches for Esterel are software synthesis for execution on a general purpose processor and hardware synthesis. A middle way is the use of a specialized processor to execute suitably tokenized Esterel programs. Such a processor is called *Reactive Processor* because of the reactive nature of the executed Esterel programs. One implementation of this approach is the *Kiel Esterel Processor* (KEP) [25, 24]. It supports a subset of the Esterel language natively including valued signals (integer), exceptions, and suspension/abortion. Parallel blocks are supported by interleaving the

parallel statements in a static way. To produce this interleaving schedule a preprocessor is used [23]. It analyses the dependencies between the different threads and orders the parallel Esterel statement blocks into a sequential list of statements for the KEP. Therefore to make such a sequential list of statements feasible, the input programs must generally be free of cyclic dependencies. The transformation presented in this thesis can be used to fulfill this condition for cyclic constructive programs.

The compilation of Esterel cannot only be complicated by cyclic dependencies, but also by *signal reincarnation*, also known as *schizophrenia* [3]. An efficient cure for schizophrenia in Esterel has been proposed by Tardieu [37]. It is based on source code transformation and therefore suited to be sequentially applied to Esterel programs together with other transformations. The work on schizophrenia is related to this thesis, because the transformation presented here applies only to input programs which are free of any schizophrenia problems.

Another kind of source code transformation is addressed by Tardieu and Edwards [38] in elimination of dead code in programs of an extended Esterel version called Esterel*. The search for signal dependencies in this work does recognize dead code, too. But such unreachable code is not removed here, it is just not considered the source of signal dependencies.

Potop-Butucaru presents in his thesis [31] a new representation model (GRC) for Esterel. It is used to optimize the generation of efficient sequential code from Esterel programs. This scheme is restricted to programs without cyclic dependencies, too. On discovering subtle differences in rejecting programs as cyclic compared to the transformation of Esterel programs into circuits, he proposes a refinement of the GRC scheme to be able to accept the same class of non-cyclic programs as the circuit transformation.

The opposite direction with regard to cycles is taken by Riedel [33, 32]. He proposes an algorithm to deliberately introducing cyclic dependencies in combinational circuits to reduce the circuit size. In benchmarks he shows that the reduction can be a significant. This result is taken as a strong argument to support cyclic combinational circuits in future circuit synthesis tools.

The Esterel source code transformation to resolve constructive cyclic dependencies presented in this thesis has already been partly published [26, 27]. This work presents significant progress in the following fields: the identification of cyclic dependencies, treatment of multiple cycles, replacement expressions in 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.

1.3 Overview

The remainder of this thesis is organized as follows.

Chapter 2 introduces the syntax of Esterel and the basics of constructiveness and code synthesis.

Chapter 3 lays out dependency relationships between signals in Esterel and how cycles in these dependencies can result in non-constructiveness. An algorithm is provided to decide if an Esterel program contains cyclic dependencies and which signals are part of that cycle.

Chapter 4 introduces a transformation algorithm to resolve cyclic dependencies on pure signals, which do not carry a value. Additionally some extensions of the transformation algorithm for constructiveness analysis are presented.

Possible extensions to handle valued signals are addressed in Chapter 5. It involves additional variants on signal dependencies and necessary refinements in renaming valued input signals.

The algorithm in its pure form does not contain any optimizations, therefore the transformed programs will contain redundant structures. Chapter 6 lists approaches on optimizing transformed programs in a post-processing way.

Chapter 7 provides experimental results on applications of different Esterel compilers on cyclic and transformed non-cyclic programs.

Chapter 8 discusses aspects of the coverage of the transformation algorithm.

The conclusions of this thesis are presented in Chapter 9, along with possible future work.

1.4 Acknowledgments

This thesis would not have been possible without the support of many people. I would like to thank them here.

At first I thank my doctoral advisor Professor Reinhard von Hanxleden for providing me a pleasant research environment, for many ideas, discussions, suggestions, and invaluable support for this work.

I thank Professor Stephen Edwards for helpful hints on problems and for providing his Esterel compiler as a solid foundation to build upon.

Professor Klaus Schneider noted an alternative application to be helpful for constructiveness analysis.

Further thanks go to Xin Li for conducting the hardware synthesis experiments.

Claus Traulsen had been very helpful with fruitful discussions and proof reading of this work.

Hauke Fuhrmann prepared the SCADE/Lustre implementation of the Token Ring Arbiter example.

Our secretaries Gerti Rosenfeld and Gesa Walsdorf along with the computer administration staff Corinna Dort, Peter Pichol, and Willi Burmeister provided dependable support on administrative and technical tasks.

Finally, I thank my parents Waltraud and Peter for their continuous support.

Chapter 2

The Esterel Language

Computer systems are not entities without any outside connections. They almost always accept input data from an environment and produce output data handed back to the environment. However, systems vary in a great degree in the tightness of integration with the environment. One can differentiate three basic types of systems [4]:

- *Transformational systems* read input data at startup, perform some computations, and deliver output data at termination. Examples: Batch processing, simulations, compilers.
- *Interactive systems* run continuously waiting for input data, perform computations and produce output data. Examples: Databases, word processors, operating systems.
- *Reactive systems* are tightly integrated into a physical environment and receive input data from sensors and produce output values for actuators.

  They typically implement control algorithms to manage the physical system they are embedded in and have to obey constraints on the answer time dictated by the environment. Optimizations typically try to minimize the worst case answer time, hardware cost, and/or electric power demands. Examples: Engine controllers, traffic control, plant automation.

These three systems differ in further properties:

The computational results of transformational systems typically depend only on their input values, no internal state is kept at termination. Interactive and reactive systems manage an internal state, on which the computations depend.

Performance considerations differ for these systems, too. Transformational and interactive systems are usually optimized for the delivery of results in a minimum of time in the *average case*. This is in contrast to reactive systems, where the implemented control algorithms must not be analyzed for the average but for the *worst case*. The implemented

algorithms on reactive systems have to obey maximum answer times which are derived from the demands of the environment. Being considerably faster than those demands is not of much use. Further optimization targets for reactive systems are hardware cost and/or electric power demands which are not of such a high priority for transformational and interactive systems.

The degree of synchronization of concurrent activities in transformational and interactive systems is typically limited to few communication points a runtime. The scheduling is not necessarily fixed and changes dynamically depending mostly on the computational needs of each process. Reactive systems typically consist of several concurrent threads which are tightly coupled in exchanging state information. The execution order is therefore mostly limited by these communication dependencies.

Since each system has its own demands on optimizations, different programming languages are differently specialized for use in each domain. This thesis is concerned with reactive systems, therefore the following part will give a short introduction on programming principles of reactive systems.

2.1 Programming Reactive Systems

The purpose of *reactive systems* is the interaction of a computer system with its environment to control some kind of physical process. Basically two kinds of information flows are need to interact with the the environment: *Input* signals sense the current state of the environment, *Output* signals perform certain actions in the environment. To ensure adequately continuous reactions on changes in the environment, the input and output signals are typically handled in a control loop:

1. Read input values
2. Compute reaction
3. Write output values
4. Repeat.

The frequency of repetition is determined by the physical process in the environment the system is embedded in. Most reactive systems are considered *real-time systems* with additional requirements of guaranteed bounds on the reaction time between changes on the input values and output reactions.

The computation of the reaction function itself depends generally not only on the inputs alone, an internal state of the system is commonly used, too. This naturally leads to the programming model of *Moore/Mealy finite state machines* [21].

The main drawback of state machines lies in its limitation to fairly small models:

- The specification of the transition function becomes quite complex for big systems.
- There are no provisions for grouping subsystems into modules.
- An efficient solution for parallel activity is missing.
- Local variables for internal signaling are not available.

Some of these issues can be dealt with by using development tools which provide a way to graphically specify the state machine. State machines are usually visualized by graphical nodes representing the states S and edges between nodes representing the transition function δ. The input Σ and output Γ signals are contained in labels at transitions as conditions and actions. Grouping some states into a hierarchy of some kind provides modularization of components. All these features are easily translated back into the original flat state machine.

The remaining desirable features — parallelism and internal signals — poses some structural problems. A state machine does provide only one point of control: The execution of the transition function δ. Multiple points of control in multiple threads can be mapped to a single δ function by computing the cross product of reachable state spaces between all parallel threads. The drawback of this method is a potentially exponential growth of states and transition rules in the target state machine. This growth (called *state explosion*) can make it infeasible to implement even moderately sized programs.

The remaining problem is how to implement communication between the parallel threads. Internal signals similar to the input/output signals may be seen as an obvious solution. These signals can be implemented by adding them as additional input signals which are driven internally and not by the environment. When adding these internal signals, the order of setting and testing these signals must be clarified. For conventional input and output signals connected to the environment the order is defined: In each execution run the input signals are read from the environment first, then state/signal computations are performed, and the output signals are fed back to the environment last. The setting of internal signals is not defined in such a simple way since the value of internal signals is determined as part of the computation step.

Huizing and Gerth [22] formulated three desirable aspects on treating signals in parallel activities:

Responsiveness: The System reacts with conceptually no delay on external or internal events. This abstracts the temporal characteristics of the system from the logical behavior by assuming instantaneous reactions from input events. The actual specification of the reaction delay is deferred to later stages of the system implementation.

Modularity: The interface to the environment and distinct modules of the system are of the same nature. The input/output behavior of the entire system can be expressed as a composition of the input/output behavior of each module. This enables the

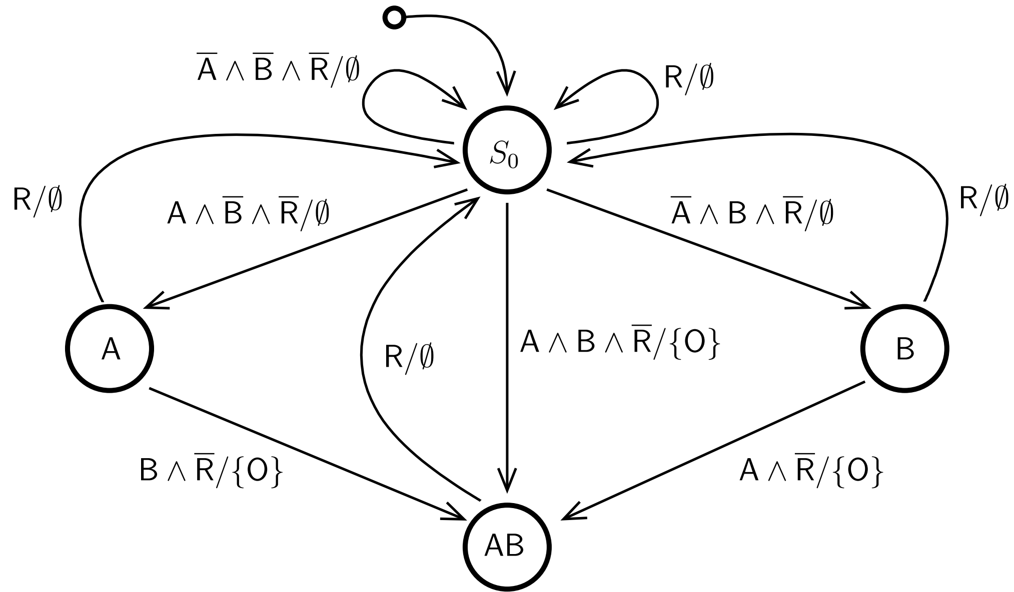


Figure 2.1: Finite state machine implementation of the ABRO specification.

composition of a complex system from separately developed subsystems without detailed insight into the internal structure of each subsystem.

Causality: All actions in the system can be derived from input events in a causal chain without any non-determinism. Actions executed under a condition must not invalidate that condition. This property ensures deterministic behavior of the system.

As Huizing and Gerth proved [22] these properties cannot be unified into a single semantics of a reactive system. Nevertheless Esterel aims to fulfill all three properties. It does so by explicitly rejecting those programs where fulfilling all three properties leads to ambiguous behavior. Responsiveness and modularity are fulfilled by the way I/O and internal events are treated. Causality is ensured by rejecting all programs which are not *constructive*. This is done at the compilation stage to avoid run time errors which are not acceptable in a reactive system which may be even safety critical.

The details of constructiveness are laid out in the following sections.

2.2 Basic Structure of Esterel

Esterel is an imperative textual language intended to implement reactive control programs. The main advantages of Esterel over plain finite state machines are support for:

- Hierarchy,
- Parallel activities, and
- Exception handling.

These points greatly aid in developing big programs. The canonic example to motivate the usefulness of these features is the *ABRO Example*. It is taken from Berry's Esterel language primer [4]. Its specification is as follows:

> Emit an output O as soon as two inputs A and B have occurred.
> Reset this behavior each time the input R occurs.

What this specification misses is how the conflict of the occurrence of A and B at the same time with R is resolved. The actual implementations assigns R priority over A and B. Therefore A or B must have no effect if R occurs at the same time.

The ABRO specification translates into a state machine which is depicted in Figure 2.1. The complexity of the FSM results from the parallel activity of waiting for two different signals A and B. The arrival of each signal combination is preserved by distinct states. The transitions between states explicitly have to cover all combinations of input signals making the program hard to read and error prone when constructed manually.

The extension of the specification to the arrival of three input signals A, B and C will double the number of states to eight with more than twenty transitions between states. Such an exponential growth of complexity is known as a *state explosion*.

The Esterel implementation of ABRO (again taken from [4]) is much simpler as the plain FSM:

```
module ABRO:

input A, B, R;
output O;

loop
  [ await A || await B ];
  emit O
each R

end module
```

The outer "loop...each R" structure restarts its inner part whenever signal R is sent. This repetition implements the "Reset this behavior each time the input R occurs" part of the specification of ABRO. The first activity inside the loop consists of two await statements which are executed in parallel. The parallelism is indicated by the || bars. Each await statement stops its execution until the indicated signal occurs. If both signals A and B occurred then the entire parallel construction terminates and the following emit

statement sets the output signal O. Now the execution halts until the signal R is received and the loop restarts.

The benefit of Esterel over a plain FSM lies in the much simpler structure of the program, which follows the *Write Things Once* (WTO) principle [4]. The reset signal R is only tested at one point, the loop...each condition. In the FSM implementation R is contained in all transition labels. Furthermore the waiting on the input signals A and B does involve just one await statement for each signal. Additional signals could be added by just inserting "|| await C", "|| await D". No state explosion happens here.

2.3 Overview of Esterel Statements

The following section will give a short introduction into the statements and control structures of Esterel and can be skipped by readers familiar with Esterel. The level of detail will suffice to be able to follow this thesis without previous knowledge of Esterel. For a more detailed introduction refer to the informal introduction to Esterel [4], the description of Esterel semantics [3], or the manual of the v5 compiler [8].

An Esterel program is made up of two main parts: The interface declaration and the body. In the interface the names of interface signals are listed and their respective data direction defined by either input or output:

```
module MAIN:

input A, B, R;
output O;

p

end module
```

The body p of the program consists of a hierarchical layer of control and signal handling statements. The set of Esterel statements is divided into the small set of basic *kernel* statements and the much larger set of *derived* statements. All derived statements can be expressed by means of kernel statements.

2.3.1 Kernel Statements

Besides the input and output signals, internal signals can be defined by the signal block:

```
signal S in
  p
end
```

Here the signal S is defined in the block p. Access to S outside of p is not possible. If multiple signals with the same name are nested, then the innermost definition in the

current context is visible.

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

In each instant all threads of the programs execute their tasks until they encounter a pause statement. At that point all threads are synchronized and a new instant is started. The signal emissions in the previous instant have no direct influence on the signal statuses of the new instant. Nevertheless the *program state* (*i. e.*, which pause statements got executed) is preserved, which is typically implemented by *state registers*.

The only way to set a signal's presence state with a kernel statement is via the emit statement:

```
emit S
```

If a signal S is emitted, then it is considered *present* for the current instant. Multiple emissions on the same signal in one instant are permitted but only the first one has any effect.

There exists no statement to unset the presence state of a signal, because this would contradict a previously executed emit statement. A signal can only implicitly be set to *absent* by not executing any emit statements on it.

Multiple statements can be concatenated with the with the ";" operator:

```
emit S;
emit T
```

Both emit statements are executed sequentially in the same instant.

The most basic control structure of Esterel is the present statement:

```
present S then
  p
else
  q
end
```

It evaluates a signal expression S and executes one of two branches with code blocks p or q. The signal expression S may be comprised of signal names and boolean operators. If one of the then or else branches contains no code, then it can be omitted, *e. g.*,

```
present S else
  q
end
```

The pause statement halts the execution in the current instant and resumes in the next instant:

```
emit S;
pause;
emit T
```

Signal S is emitted in the first instant and signal T in the following instant.

Repetitive behavior can be implemented by use of the loop structure:

```
loop
  p
end
```

The body p is immediately restarted whenever p terminates. It follows that p must execute at least one pause statement to limit the amount of work being done in an instant to a finite amount. Esterel compilers check for this property, therefore the execution of at least one pause statement must be detectable statically.

A key feature of Esterel is its capability to execute parallel threads in a deterministic way:

```
  p
||
  q
```

The code blocks p and q are started concurrently. The entire parallel terminates when all parallel threads are terminated.

The parallel operator "||" has a lower priority than the sequence operator ";". If a parallel block needs to be prepended or appended by another statement block, then squared brackets can be used to group these blocks:

```
emit A;
[
  p
||
  q
];
emit B
```

Here an emission on A is executed first in a single thread, then the control flow splits into the two parallel blocks p and q. If p and q have both terminated, then B is emitted again in a single thread. The syntactical order of parallel threads (p first, q second) has no meaning, all threads have the same execution priority.

As mentioned above, the loop statement does not terminate, it repeats its body infinitely. This is not universally useful, some systems need the repetition of some behavior for a certain time and eventually stop that behavior and start another. Such a limited repetition is not supported by the loop statement alone. In Esterel such a behavior can be implemented with exception handling by the trap statement:

```
trap T in
  p
end
```

The trap statement consists of the definition of a trap signal (here T) and a body (p) as a scope of that signal. The body may contain an exit T statement to activate ("throw") the exception signal T. The control flow does not continue after the exit statement but after the entire trap statement.

As an example, the following code block implements waiting for the arrival of a signal S:

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

The activation of the trap is called *weak* because other parallel threads inside the trap body continue execution until they reach a pause statement or terminate anyway:

```
trap T in
    exit T
  ||
    emit A;
    pause
end;
emit B
```

In this example, the first thread inside the trap body throws the exception. But nevertheless the emission of signal A is executed in the second thread. When executing the second thread ceases for the instant at the pause statement, then the entire trap body is terminated and B is emitted. As a result signals A and B are emitted in the same instant.

Nested traps have a defined meaning in Esterel, too. If multiple trap signals are activated in a trap hierarchy, then the outermost defined trap signal takes priority:

```
trap T1 in
  trap T2 in
      exit T1
    ||
      exit T2;
  end;
  emit A
end;
emit B
```

Here the conflict between T1 and T2 is won by T1 as the outermost trap signal. As a

consequence the signal A is not emitted, only B.

The handling of exceptions in Esterel is fully integrated into the synchronous behavior without any non-determinism. This applies also for exceptions in the presence of concurrent activities.

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.

It appears that the suspend statement is rarely used directly in an application program. It is mostly part of the expansion of derived statements into kernel statements.

The last command in this list of kernel statements is the nothing command:

```
nothing
```

It does nothing indeed and has no effect while being executed. This command is unneeded in a strict sense, but it simplifies reasoning on program transformations. Empty statement blocks are not permitted in Esterel. Therefore it is much easier to define the deletion of a code block by substitution with nothing than to ensure the validity of a cascade of block structures.

2.3.2 Derived Statements

The following section does not contain a complete list of all Esterel commands that are not part of the kernel. Only those commands are explained that are frequently used in the remainder of this thesis. All these command are listed along with their expansion into kernel statements. These expansions are mostly based on other derived statements which are defined before.

A very simple derived command is halt. It is meant to stop execution of a thread. The implementation in kernel statements involves just an (infinite) loop around a pause statement:

```
                loop
halt      ⇝       pause
                end
```

Exceptions via the kernel trap statement are a means of self termination of code blocks by explicitly executing an exit statement. The block containing that exit statement is terminated and the execution continues on an upper level. In some programs the need

arises to terminate an active code block when a signal external to the current block is set. This functionality is supported by the non-kernel abort statement:

```
                        trap T in
                          suspend
                            p
                          when S;
                          exit T
abort                    ||
  p          ~>           loop
when S                      pause;
                            present S then
                              exit T
                            end
                          end
                        end
```

The code block p is terminated whenever the signal expression S is evaluated to true. This termination is different from the trap termination method. trap implements a *weak* termination method by executing the body until the end of the instant before handing control over to the outside level. In contrast to that abort implements a *strong* termination control. The abort condition is tested at the beginning of the instant. If the condition applies then the control does not start in the body of the abort for that instant but at the end of the abort statement.

A weak termination method is available for abort by adding the attribute weak:

```
                        trap T in
                          p;
                          exit T
                         ||
weak abort                loop
  p          ~>             pause;
when S                      present S then
                              exit T
                            end
                          end
                        end
```

An additional constraint exists for the evaluation of the abort condition: In the first instant on entering the abort statement the condition is not evaluated. Therefore the body p is executed even if the condition applies. The condition is started to be evaluated when at least one pause statement got executed in the body.

The probably most used derived command is the await command:

```
                        abort
await S      ~>           halt
                        when S
```

Its purpose is to stop the execution at the current execution point until a signal expres-

sion S evaluates to true. The signal state in the first instant is ignored, this command waits at least for one instant.

The statements suspend, abort, and await have the delayed execution in the first instant in common. In some program contexts this behavior is not wanted. For this case the attribute immediate can be applied to the respective signal condition. The use of that attribute has slightly different expansions into kernel statements as a consequence:

```
suspend                      suspend
  p                    ⤳       present S then pause end;
when immediate S                p
                             when S

                             trap T in
                                 suspend
                                   p
                                 when immediate S;
abort                            exit T
  p                    ⤳        ||
when immediate S                 loop
                                   present S then exit T end;
                                   pause
                                 end
                             end

                             abort
await immediate S      ⤳       halt
                             when immediate S
```

These expansion rules are quite compact in size, but contain references to other non-kernel statements which must be expanded in turn. The end result in kernel statements will be quite voluminous, while a direct translation into kernel statements would be more space efficient. For example applying the former rules to **await immediate** S:

```
                           trap T in                      trap T in
                               suspend                        suspend
                                 loop                           loop
                                   pause                          present S then pause end;
                                 end                              pause
await immediate S                when immediate S;              end
                                 exit T                       when S;
⤳                                                              exit T
                       ⤳       ||                      ⤳      ||
abort                            loop                         loop
  halt                             present S then               present S then
when immediate S                     exit T                       exit T
                                   end;                         end;
                                   pause                        pause
                                 end                          end
                           end                            end
```

A much more efficient expansion with the same functionality is:

```
                              trap T in
                                loop
                                  present S then
await immediate S     ⇝             exit T
                                  end;
                                  pause
                                end
                              end
```

2.4 Reactivity, Determinism, Constructiveness

Syntactical soundness is not sufficient for a valid Esterel program. As mentioned in Section 2.1 (page 9) on Responsiveness/Modularity/Causality, those programs are rejected which are not able to fulfill all three criterions. Responsiveness and modularity are fulfilled by design of the reaction characteristics of the signal signal interface of Esterel. Therefore causality is the critical point which decides on the validity of an Esterel program.

The following three sections will give a short overview on the main three semantics used to describe the meaning of an Esterel program. Full details can be found in Berry's draft book [3] on the semantics of Esterel.

The first one — the logical behavioral semantics — is the historically oldest of the three. It differs from the other two because it is not based on causality but on reactivity/determinism, which leads to a bigger set of accepted Esterel programs. The second (constructive behavioral) and third (constructive circuit) semantics are considered equivalent in their set of accepted programs.

2.4.1 Logical Behavioral Semantics

This section follows Chapter 6 of Berry's book on the Esterel semantics [3] without reproducing every detail given in that book.

Despite being refined by the constructive semantics, the logical semantics is useful because it formally defines the behavior of Esterel statements. A set of derivation rules is given for the kernel statements of Esterel in the form of textual transformation rules based on the Structural Operational Semantics [30].

The reaction of a program P in an instant is written as the transition:

$$P \xrightarrow[I]{O} P'$$

The transition takes place under control of the set of input signals I and produces the output signals O. The execution state of the program is reflected in the rewriting of P into P'.

The reaction of the program in an instant is expressed by means of a hierarchy of reactions of single statements and sub-blocks. These individual reactions are written in the following form:

$$\frac{\textit{condition}}{p \xrightarrow[E]{E',k} p'}$$

This derivation describes the transition of statement p into p'. E denotes the current signal environment and E' the signals emitted as part of the transition. It must be taken into account that the synchronous coherence property of Esterel implies $E' \subset E$. The signal state in E is stored as an element s^+ for a currently present signal s and s^- for the absent case.

The termination of sub-blocks is controlled by numerical *completion codes* (symbolized here by k): A 0 means normal sequential execution of statements, 1 denotes the encounter of a pause statement, and 2 and following are reserved for trap signals with ascending priority. The processing of a reaction in an instant concludes therefore always with a completion code of 1.

To be able to complete this transition, the *condition* must hold. This condition may be comprised of other transitions or expressions on signals and completion codes. The actual derivation of p into p' is formulated as a recursive hierarchy of rule applications inside further sub-conditions. For some kernel commands the condition may be empty.

To be able to write the derivation rules in a compact form, an abbreviated syntax of Esterel is introduced called the *Esterel Process Calculus Syntax* or shorter the *terse* Syntax:

nothing	0	
pause	1	
emit S	!S	
present S **then** p **else** q **end**	S?p, q	
suspend p **when** S	S $\supset p$	
p;q	$p; q$	
loop p **end**	$p*$	
p || q	$p\|q$	
trap T **in** p **end**	$\{p\}$	using $\uparrow p$ and $\downarrow p$
exit T	k	with $k \geq 2$
signal S **in** p **end**	$p\backslash$S	
[p]	(p)	

Most statements have a direct equivalence in the terse syntax, but some exceptions apply: Empty then or else branches are always explicitly included by adding nothing.

The trap signals are anonymized into completion codes starting from 2. To preserve the relationship between trap environments and exit statements, auxiliary operators $\uparrow$ and $\downarrow$ are introduced. $\uparrow p$ increases the completion codes of all exit statements inside p by one. It is used as an intermediate operator to describe the translation into the terse syntax, it is not part of the final program. The $\downarrow k$ operator is added to completion codes of

trap environments to select the matching trap. If a completion passes a $\downarrow k$ operator it is decremented. If the result equals two then the current trap environment is selected.

Applying this terse syntax to a short example yields:

```
trap T in
  loop
    present A then
      emit B
    else
      exit T
    end
    pause
  end
end
```

$\rightsquigarrow \quad \{(\mathsf{A?!B}, 2; 1)*\}$

The following behavioral rules define the semantics of the kernel statements of Esterel. They are not all listed here, just some to give a basic insight into their architecture.

$$k \xrightarrow[E]{\emptyset,k} 0$$

(compl)

This rule handles the termination of statements. The completion code k represents 0 for the instantaneously terminating nothing statement, 1 represents the pause statement, and 2 and higher numbers represent trap signals.

$$!s \xrightarrow[E]{\{s^+\},0} 0$$

(emit)

The emission of a signal s is returned as a new element s^+ in the environment. This command has no condition limiting its execution.

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

(present+) *(present-)*

The present command is covered by two rules, which are selected depending on the tested signal s. If s is present (*i. e.*, $s^+ \in E$) then the left rule *(present+)* applies, otherwise the right one *(present-)*. The execution of the sub-blocks p and q is added to the condition to derive p' and q' respectively.

Behavioral rules for the remaining Esterel statements can be found in Berry's book on the Esterel semantics [3].

Using the former behavioral rules the *logical correctness* of an Esterel program P with regard to a set of inputs I can be defined by arguing on the existence of derivations $P \xrightarrow[I]{O} P'$ leading to some program P' and a set of output signal states O:

- P is *reactive* w.r.t. I if at least one derivation $P \xrightarrow[I]{O} P'$ exists.
- P is *deterministic* w.r.t. I if at most one derivation $P \xrightarrow[I]{O} P'$ exists.
- P is *logically correct w.r.t.* I if it is reactive and deterministic w.r.t. I. That is there exists exactly one derivation for P and I.
- P is *logically correct* if it is logically correct for all possible input sets I.

It must be noted that the application of derivation rules to a program P covers only the very *first* temporal instant of the program execution. After that first instant P is transformed into P' for the next instant. Derivations obtained for P do generally not apply to P', each instant must be treated individually.

The derivation rules correctly reflect the behavior of the Esterel statements and the concept of logical correctness is useful in rejecting invalid Esterel programs.

```
present A then
  emit A
end
```

This program is reactive but not deterministic, because it has two different solutions: A absent and present. Therefore this program is not logically correct. This is reflected by the following derivations:

$$\frac{\mathsf{A}^- \in E \quad 0 \xrightarrow[E]{\emptyset, 0} 0}{\mathsf{A?!A,0} \xrightarrow[E = \{\mathsf{A}^-\}]{\emptyset, 0} 0} \qquad \frac{\mathsf{A}^+ \in E \quad \mathsf{!A} \xrightarrow[E]{\{\mathsf{A}^+\}, 0} 0}{\mathsf{A?!A,0} \xrightarrow[E = \{\mathsf{A}^+\}]{\{\mathsf{A}^+\}, 0} 0}$$

(A absent) (A present)

The following program has no solution for A at all, A present and absent both lead to contradictions.

```
present A else emit A
```

The logical correctness is rejected as the main criterion on the validity of Esterel programs, because it is not able to reject certain programs with counterintuitive behavior. In the following program the emission of signal A depends on the signal itself. Nevertheless it is logically correct, since only for A present a derivation free of contradictions can be made.

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

$$\frac{\mathsf{A}^+ \in E \quad \mathsf{!A} \xrightarrow[E]{\{\mathsf{A}^+\}, 0} 0}{\mathsf{A?!A,!A} \xrightarrow[E = \{\mathsf{A}^+\}]{\{\mathsf{A}^+\}, 0} 0}$$

The fundamental reason for this problem lies in the direction of the information flow, when the program is analyzed. The emission of signal A in both branches must be known before the outer present test is evaluated. This speculative reasoning on signal statuses is dealt with by testing all possible input combinations and testing which lead to derivations free of contradictions. Such an approach is obviously computationally very demanding and does scale poorly for large programs.

2.4.2 Constructive Behavioral Semantics

The motivation to develop another approach for testing the validity of Esterel programs besides the logical correctness is founded in the non-intuitive flow of information in programs according to the logical correctness. This section presents the *constructive behavioral semantics*. It is based on strictly following *cause and effect* to determine the behavior of a program. When deriving the state of a signal no speculative reasoning must take place, only known facts on signal statuses are used to derive further signal statuses (the *constructive* approach).

"Constructiveness" is defined by Berry [3] as a property of signal emissions:

- A signal is declared *present* iff it must be emitted.
- A signal is declared *absent* iff it cannot be emitted.

If a signal cannot be declared either present or absent with regard to some reachable program state, then the program is considered not constructive.

The actual constructiveness analysis is defined by two functions $Must$ and Can which iterate on the Esterel program and derive iteratively the emission status of all signals. The $Must$ function detects which signals are emitted derived from the status of already known signal statuses. The Can function searches for potential emissions of signals. That information is not directly useful, but the complement of the result ($Cannot = \overline{Can}$) describes, which signals are absent. The direct computation of $Cannot$ would be possible, but Can is technically easier to handle.

Both functions are applied iteratively on the Esterel program. In each iteration run the signal environments delivered by $Must$ and Can are enriched with further knowledge on signal emissions. The set of signals delivered by $Must$ describes the signals which are set to present in the current instant. Can produces all signals with a possible emission, including the signals delivered by $Must$. The element-wise inverse of the Can set delivers those signals which cannot be emitted, therefore these are the signals which are not emitted in the current instant and considered absent. If after several iterations no further progress on signal states can be derived the the iteration is terminated. If there are signals remaining, which are not either present or absent then the program is considered non-constructive.

Both functions collect information on the completion codes returned by the program, too. The result of each function application is returned as a pair with sets of emitted signals as the first element, and the completion codes as the second element.

Each Esterel statement is handled by a separate set of $Must/Can$ functions. The first of these functions manages completion codes.

$$Must(k, E) = \langle \emptyset, \{k\} \rangle$$
$$Can^m(k, E) = \langle \emptyset, \{k\} \rangle$$

Both functions return an empty set as the first element, since completion code statements like nothing, pause, or exit do not change the state of signals. In the second element the respective completion code is returned.

$$Must(\mathsf{!s}, E) = \langle \{s\}, \{0\} \rangle$$
$$Can^m(\mathsf{!s}, E) = \langle \{s\}, \{0\} \rangle$$

Signal emissions are returned in the first entry and a completion code of value zero because the emit statement terminates immediately.

$$Must((\mathsf{s?p:q}), E) = \begin{cases} Must(p, E) & : s^+ \in E \\ Must(q, E) & : s^- \in E \\ \langle \emptyset, \emptyset \rangle & : s^\perp \in E \end{cases}$$
$$Can^m(\mathsf{s?p:q}, E) = \begin{cases} Can^m(p, E) & : s^+ \in E \\ Can^m(q, E) & : s^- \in E \\ Can^\perp(p, E) \cup Can^\perp(q, E) & : s^\perp \in E \end{cases}$$

The handling of the present statement is significantly more complicated. Both Can and $Must$ are applied in the same way if the status of the tested signal s is known. If s is present then p is evaluated, if s is absent then q is evaluated. Can and $Must$ differ fundamentally for an unknown s. Can evaluates both p and q and unifies the result. This implements the detection whether no emissions for signals are contained in p or q. $Must$ cannot continue on an unknown signal s because this would enable speculative computation.

Further details on $Must/Can$ rules can be found in [3].

The $Must/Can$ analysis covers only one instant of the Esterel program. Analysis of the entire program must be repeated for each derivative program covering all reachable execution states of the program. Additionally for each program state all combinations of states of input signals must be covered. Therefore a constructiveness analysis of a program is considered very costly, leading to approaches to avoid a full constructiveness analysis. These approaches typically put more restrictions on the validity of an Esterel program. The most prominent restriction is the *acyclic* property. Acyclic programs are a superset of constructive programs.

A test for the absence of cyclic dependencies is efficient and therefore implemented by all Esterel compilers. In fact among the actual available Esterel compilers, only the Esterel v5 compiler is optionally able to perform a constructiveness analysis. The Esterel v7 compiler, the CEC and others are limited to acyclic Esterel programs. Details on cyclic

dependencies can be found in Chapter 3.

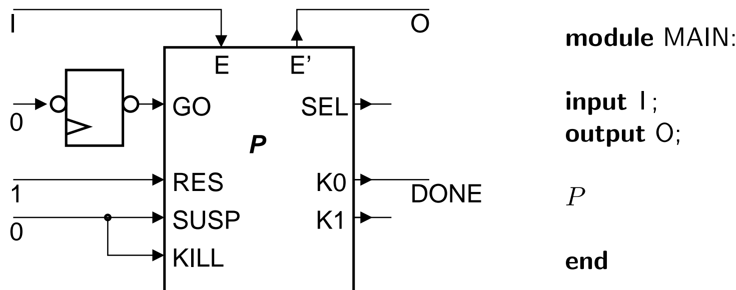
2.4.3 Constructive Circuit Semantics

As already mentioned in Section 1.2 there exists a scheme to translate an Esterel program into a circuit of logic gates implementing the same behavior. This scheme is efficient as it is not affected by state explosion, because concurrent activities are synthesized as parallel sub-circuits. However some logically correct Esterel programs showed some anomalies in their respective circuit representation. Since these anomalies where rooted in speculative execution of program parts, a refinement of the logical correctness into the constructiveness became the reference semantics of Esterel. Constructiveness forbids speculative execution and makes it feasible to synthesize efficient circuits from Esterel programs.

The translation of an Esterel program into a circuit as defined by Berry [3] is specified as a structural translation of the program hierarchy. For each individual Esterel statement a translation rule is given. By recursively replacing the Esterel statements from the outer to the inner hierarchy the whole Esterel program is replaced by its circuit representation.

The following paragraphs lists the translation of a selection of Esterel statements extended by a logical correct program with an unstable circuit representation.

The topmost layer connects the Esterel program P to the environment:



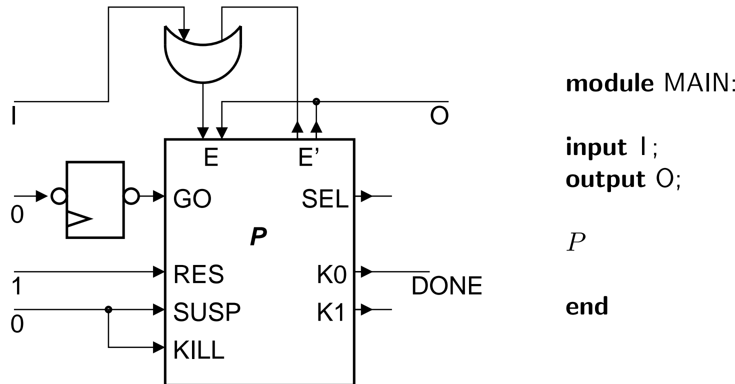
The input signals are connected to the E ports, the output signals to the E' ports. All tests of input signals can only evaluate signals in the E set, and all emissions of output signals emissions are connected to the E' set.

The entire circuit is driven by a single clock signal which is not explicitly shown here. The clock is connected to every register in the circuit, therefore the entire circuit runs synchronously on the same clock. One clock cycle in the circuit corresponds to one instant in the Esterel program. The GO input starts the circuit. It is connected to a register to produce a single 1 signal at power up of the circuit. This register is also known as the *boot register*.

The K0 and K1 outputs reflect the completion codes of P. At the topmost level the program P will set K1 while it is running and K0 when it terminates completely. Translations of inner statements can have further outputs for completion codes greater than 1 for trap exceptions.

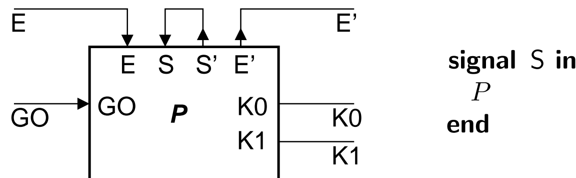
The RES, SUSP, KILL, and SEL signals are related to suspension, exceptions, and parallel termination and of no further interest in this context. They will be left out in the following translations.

A small problem is present with this connection to the environment. Common Esterel compilers allow emissions of input signals and tests of output signals. This is not reflected by the former circuit, the following circuit is a proposal to enable the emission/testing of input/output signals:



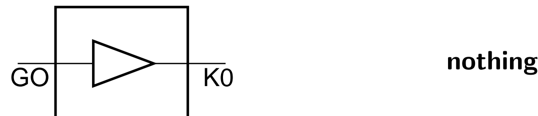
The output signals are made available in the input set and emissions of input signals are combined with the signals from the environment.

This feedback of signals from the output to the input is similar to the handling of local signals:



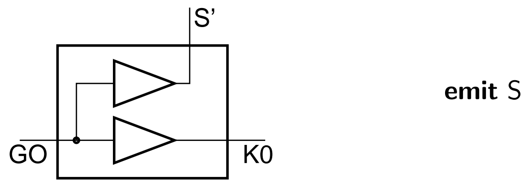
All emissions on the signal S are carried via S' out of the block *P* and back into *P* on the S inputs where they are available for tests.

The translation for nothing is the most simple one. It just connects the GO input to the K0 output indicating the immediate termination.



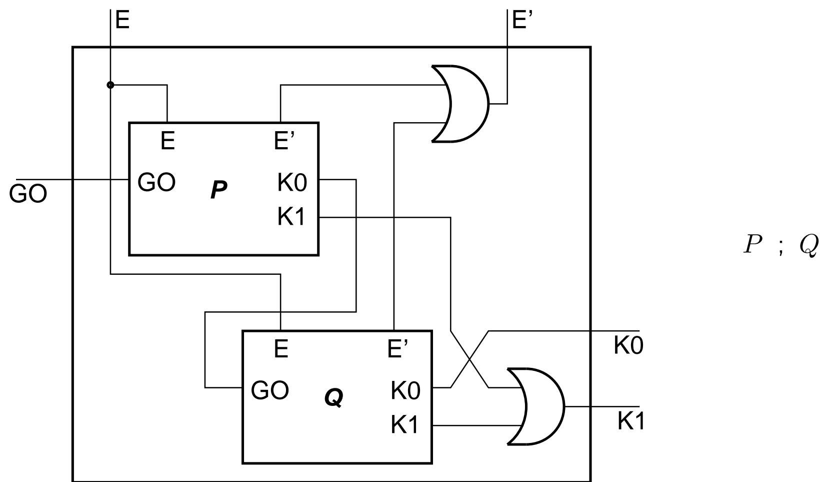
The driver gate is redundant here, it just helps to visually connect the nothing statement to an actual part of the circuit and to indicate the signal flow.

An emit S statement terminates instantly, too. In addition to nothing it sets the output part S' of the signal S. S' may be connected to a local or output signal.



emit S

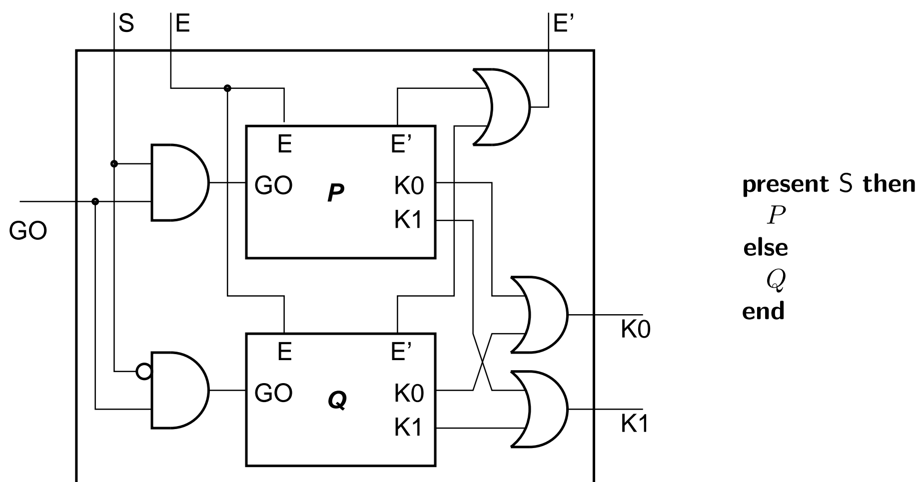
Sequential execution of two statement blocks like "P;Q" is implemented by starting P on its GO input. The termination of P is indicated by its K0 output, this signal is connected to the GO input of Q. This enables the sequential execution of P and Q:



$P \; ; \; Q$

Some additional effort must be made to propagate the completion codes up to the upper layers of the program: The entire block terminates when Q terminates, therefore the output K0 of Q is connected to the outer K0. Higher order completion codes from K1 on are simply or-combined from both blocks P and Q. Signal emissions on E' are treated likewise.

The basic control flow statement is the present statement. It structurally similar to the statement sequence:

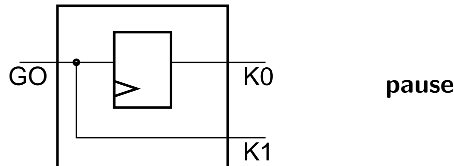


present S **then**
  P
else
  Q
end

The difference lies in the start and termination of P and Q: Signal S is and-combined with the GO signal, with the result starting one of P or Q. As a termination criterion

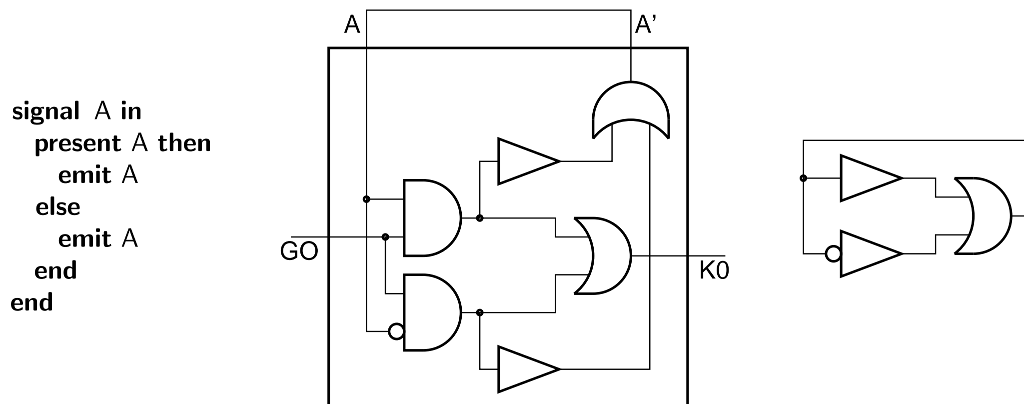
for the whole block the or-combination of K0 from P and Q is used as the outer K0.

The pause statement terminates the execution in the current instant and starts sequentially following statements in the next instant. This behavior is influenced by suspensions and exceptions. Neglecting these the following circuit represents the pause statement:



A simplified implementation of pause is used here, because all cyclic dependencies in this thesis which are illustrated as circuits do not involve suspension or exceptions.

The following circuit is a translation of the logical correct program in Section 2.4.1 on page 22. An explicit local signal definition is added to make the instantaneous feedback loop from the output to the input of the block visible.



The small circuit to the right is a reduced version of the full circuit. It shows the same behavior as the original circuit when the GO signal is assumed to be set continuously. This circuit does not stabilize for certain delays on the logic gates.

The reason for this kind of unstable behavior lies in outputs of a gates which are derived from signals which itself are not stabilized yet. Such a configuration must be rejected to achieve stable circuits. This restriction on the circuit level is implemented by the constructiveness of Esterel programs, because the test of constructiveness is based on the derivation of known (stable) signal states from already known signal states.

Chapter 3

Cyclic Dependencies

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

This chapter deals with examples of different kinds of cycles in Esterel programs and an algorithm to find these cycles automatically. The identification of cycles is a preparation step in resolving them which is described in the next chapter.

3.1 Non-Constructive Cycles

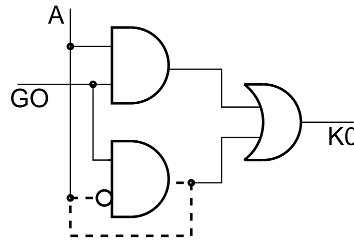
Consider the four short Esterel programs shown in Figure 3.1. The first program NREACT involves the signal A, which is an input signal, meaning that it can be emitted by the environment, and also an output signal, meaning that it can be tested by the environment. Here the environment may be either the external environment of the program, or it may consist of other Esterel modules. The body of NREACT states that if A is present (emitted by the environment), then nothing is done, which is not problematic. However, if A is absent, then the else part is activated: A is emitted, which invalidates the former presence test for A. Such a contradiction is an invalid behavior of an Esterel program; such a program is over-constrained, or *not reactive*, and should be rejected by the compiler. This problem also becomes apparent when synthesizing this program into hardware, as the gate representation of this program is an inverter with its output directly fed back to the input. This is obviously not a stable circuit and hence forbidden in Esterel.

The program NDET in Figure 3.1(b) is similar to NREACT, but with else changed to then. Here a presence of A will result in an emission of A in the then branch of the present statement, which would justify taking the then branch. Conversely, an absent A will skip

```
module NREACT:
inputoutput A;

present A else
  emit A
end

end module
```

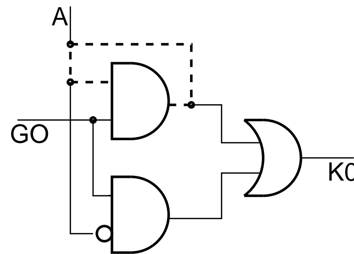


(a)

```
module NDET:
inputoutput A;

present A then
  emit A
end

end module
```

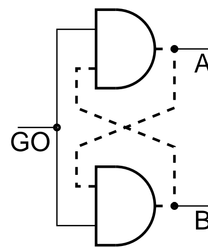


(b)

```
module CYCLE:
inputoutput A, B;

  present A then
    emit B
  end
||
  present B then
    emit A
  end

end module
```

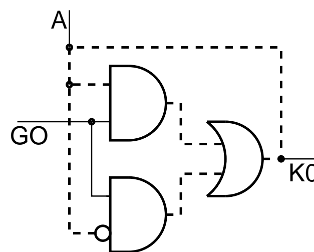


(c)

```
module SEQ:
inputoutput A;

present A then
  nothing
end;
emit A

end module
```



(d)

Figure 3.1: Invalid cyclic Esterel programs. The wires shown as dashed lines indicate the cyclic dependencies.

the emission of A. Hence, this program is under-constrained, or *not deterministic*. A compiler should reject NDET. This also becomes apparent at the gate representation of NDET, which is a driver gate that transmits the input value to the output. Now the output is fed back to the input to map the behavior of the program. As a certain gate delay is inevitable, this circuit may be an oscillator instead of providing stable outputs.

Programs NREACT and NDET have the same underlying problem: They involve a signal that is *self dependent*. In both programs the emission of A depends on a guard containing A. In these two examples, we have a *direct* self dependence, where the emission of a signal immediately depends on the presence of a signal. However, we may also have *indirect* self dependencies, in which a signal depends on itself via some other, intermediate signals. Consider program CYCLE in Figure 3.1(c), which contains two parallel threads, both testing for the signal emitted by the other one. However, the signals are emitted only if the other one has been emitted already; the emission of A depends on the presence of B and vice versa. In this case, we have a *cyclic dependency*, or *cycle* for short, and the program should again be rejected.

The former examples all involved signal emissions guarded by predicates over signal tests. Another kind of dependency is linked to the termination of a statement block. Figure 3.1(d) contains such a scenario. The emission of A depends on the termination of the preceeding present block. But that block can only be executed if the the presence status of the tested signal A is determined for that instant. That status cannot be determined at the point of testing, since the sequential order of statements puts the emission of A behind the test for that signal. In fact the circuit representation of the program contains a self dependency of signal A to itself. Therefore the program in Figure 3.1(d) is considered non-constructive.

3.2 Constructive Cycles

All four programs shown in Figure 3.1 involve non-constructive cyclic signal dependencies and are therefore invalid, and hence of no further interest to us.

However, there are programs that contain dependency cycles and yet are valid. A program is considered valid, or *constructive*, if we can establish the presence or absence of each signal without speculative reasoning, which may be possible even if the program contains cycles. The equivalent formulation in hardware is that there are circuits that contain cycles and yet are self-stabilizing, irrespective of delays [5].

Consider the program PAUSE_CYC in Figure 3.2(a): the cyclic dependency consists of an emission of B guarded by a test for A and an emission of A guarded by a test for B. At run time, however, the dependencies are separated by a pause statement into separate execution instants. The emission of B in the first instant has no effect on the test for B in the second instant.

In such a case, where not all dependencies are active in the same execution instant,

```
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 3.2: 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.

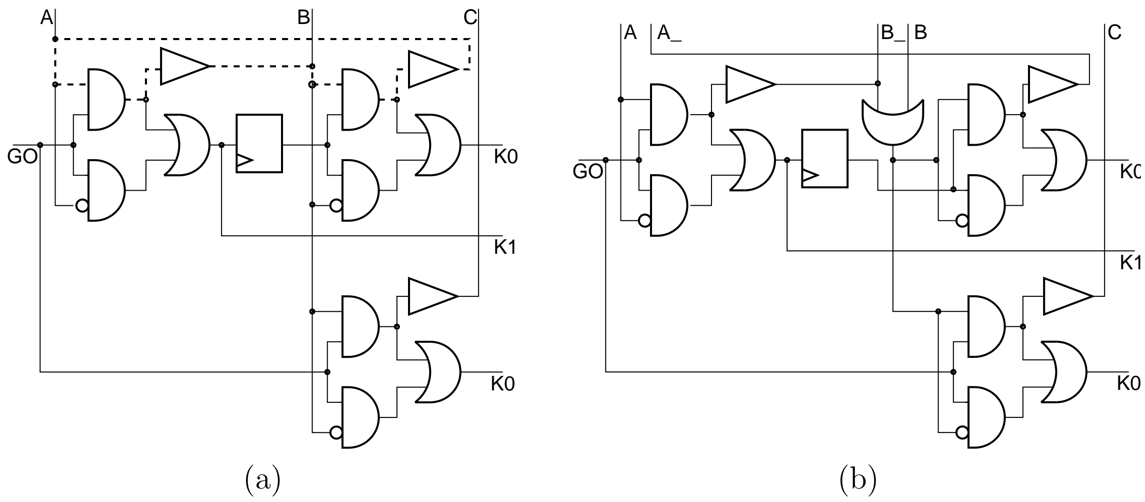


Figure 3.3: Circuit representation of the program PAUSE_CYC in Figure 3.2 (simplified without synchronizer): (a) Cycle path of the original program PAUSE_CYC, (b) Transformed, acyclic, and optimized program PAUSE_OPT with new signals A_ and B_.

```
module DRIVER_CYC:
input D;
input  Ain,  Bin;
output Aout, Bout;

loop
  present D then
    present
      [Ain or Aout]
    then
      emit Bout
    end
  else
    present
      [Bin or Bout]
    then
      emit Aout
    end
  end;
  pause
end
end module
```

(a)

```
module DRIVER_ACYC:
input D;
input  Ain,  Bin;
output Aout, Bout;

loop
  present D then
    present
[Ain and not D and (Bin or D)]
    then
      emit Bout
    end present
  else
    present
      [Bin or Bout]
    then
      emit Aout
    end
  end
  pause
end
end module
```

(b)

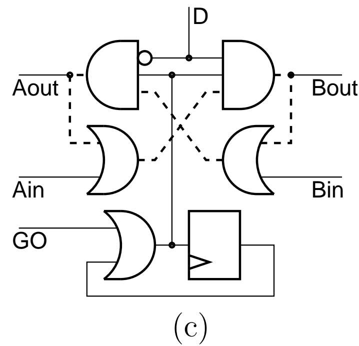


Figure 3.4: False cyclic dependencies in a bidirectional bus driver. The wires shown as dashed lines indicate the cyclic dependency.

we will call the cyclic dependency a *false cycle*. In contrast, the programs shown in Figure 3.1 all contained *true cycles*, where all dependencies involved were present at the same instant.

A cycle may be false because it is broken by a register, as is the case in PAUSE_CYC, or because it is broken by a guard, as is the case in program DRIVER_CYC shown in Figure 3.4(a). Programs that only contain false cycles are still constructive and hence are valid programs that should be accepted by a compiler.

So far, we have considered only programs that contained true cycles and were invalid (NREACT, NDET, CYCLE) or that contained false cycles and were valid (PAUSE_CYC, DRIVER_CYC). However, there also exist programs that contain true cycles, with all dependencies evaluated at the same instant, and yet are valid programs. A classic example of a truly cyclic, yet constructive program is the Token Ring Arbiter [29]; Figure 3.5 shows a version with three stations (slightly modified version from Berry [4]). Each network station consists of two parallel threads: one computes the arbitration signals, the other passes a single token from one station to the next in each instant. An inspection of the Arbiter reveals that there is a true cycle involving signals P1, P2, and P3. However, the program is still constructive as there is always one token present that breaks the cycle. Hence, a compiler should accept this program. Note that the same program, but without the first thread that emits T1 in the first instant, should be rejected. This illustrates that determining constructiveness of a program is a non-trivial process.

3.3 Variants of Cyclic Dependencies

Besides syntactical soundness, Esterel programs must be *constructive* for being considered a *valid* Esterel program. This involves the exploration of all internal program states which are reachable from the initial state [36]. For each of such states the successor states must be reachable with constructive reasoning. This test for constructiveness is complex and computationally demanding. Therefore most Esterel compilers put a stronger requirement on valid Esterel programs: Absence of cyclic dependencies. This property is much easier to check and yields another benefit: Esterel programs without cyclic dependencies are statically schedulable. This simplifies code generation, results in smaller code, faster execution, and easier runtime analysis.

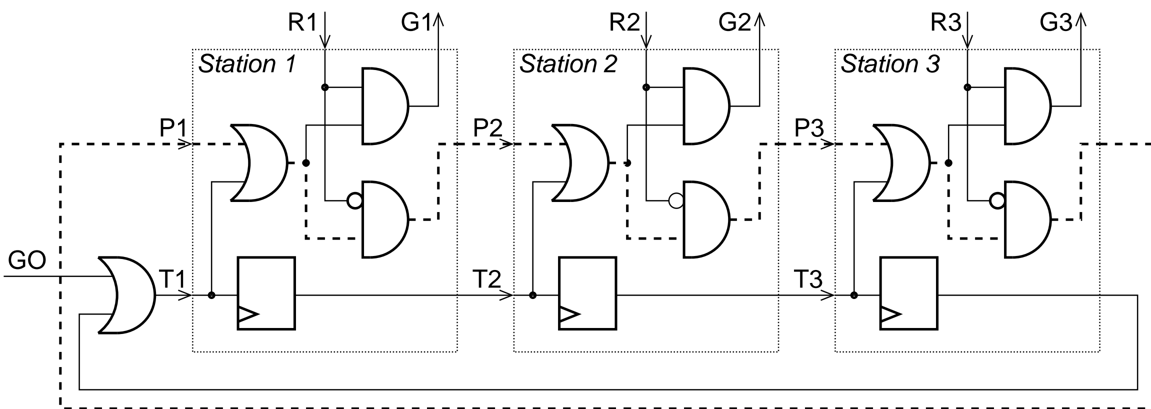
Unfortunately, different Esterel compilers can have different ideas of cyclicity of signal dependencies. Additionally there is no clear hierarchy of compilers with regard to subsets of accepted programs. And to make things worse, for no compiler a precise definition of what constitutes cyclic dependencies on the Esterel level is given. The detection of cycles presented in this work is derived from the behavior of the v5 Esterel compiler on signal dependencies in the circuit translation of Esterel programs. For an adaption of the cycle detection algorithm to other Esterel compilers the signal dependency rules as formulated in Figure 3.7 on page 38 must be checked and implemented in a modified

```
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)



(b)

Figure 3.5: Token Ring Arbiter with three stations: (a) Esterel implementation [4] with expanded run modules, (b) Simplified circuit representation, dashed lines indicate the path of cyclic dependency.

```
module CYCLE_CEC:
input I,S;

[
    present I then
      present S then
        pause
      else
        pause
      end
    end
  ||
    nothing
];
emit S

end module
```

(a)

```
module CYCLE_v5:
input A;

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

end module
```

(b)

```
module CYCLE_TRAP:
input A, B;

trap T1 in
  trap T2 in
    present A then
      nothing
    end;
    [ exit T1 || exit T2 ]
  end;
  emit B
end;
pause;
present B then
  emit A
end

end module
```

(c)

Figure 3.6: Ambiguous cyclic dependencies identified differently by CEC and v5: (a) cyclic for CEC but acyclic for Esterel v5 compiler, (b) acyclic for CEC but cyclic for Esterel v5 compiler, (c) acyclic for CEC and v5 compiler, but considered cyclic by transformation.

version accordingly. Probably the greatest potential for differences in signal dependencies will pose the termination of parallel threads, because the interaction of exception priorities with parallel termination can break cyclic dependencies.

The example program CYCLE_CEC in Figure 3.6(a) (taken from D. Potop-Butucaru's thesis [31]) is considered cyclic by GRC (graph code) based compilers like the CEC 0.3 compiler and acyclic by circuit code based compilers like the Esterel v5_92 compiler.

Potop-Butucaru shows that the problem originates in the termination control block (*synchronizer*) of parallel threads. The synchronizer is treated as opaque, causing dependency connections from otherwise unrelated inputs and outputs. As a solution, Potop-Butucaru proposes a refinement of the synchronizer by splitting it up into separate entities each handling an unrelated subset of inputs and outputs. The CEC 0.3 compiler is based on GRC code synthesis, but does not implement this modification. Therefore the program CYCLE_CEC in Figure 3.6(a) is considered cyclic by the CEC.

The parallel synchronizer for circuit representations [3] is not an opaque module, it is based on logic gates and is therefore the Esterel v5_92 compiler is able to avoid cyclic dependencies in CYCLE_CEC.

The example program CYCLE_v5 in Figure 3.6(b) is treated differently than CYCLE_CEC: The CEC is able to compile the program without any complaints on cyclic dependencies, but the Esterel v5_92 compiler rejects it as statically cyclic and needs a causality analysis to compile it. The v5_92 compiler does not recognize that the emission of A is executed

following a pause statement and therefore not dependent on the test for A.

The third example in Figure 3.6(c) contains a cyclic dependency between signals A and B involving parallel termination and trap priorities. Both the CEC and v5_92 compiler recognize the unreachability of the "emit B" statement. Therefore no cycle is detected by these compilers.

The cycle analysis algorithm described in the next section does not make use of such optimizations. It is based on the propagation of a set of active guard signals while traversing the Esterel program. Joining a parallel block is basically the union of two such sets. Applied to CYCLE_CEC in Figure 3.6(a) returns just the signal I, not including S as a guard, because both branches of the test on S contain a pause statement. As a result just I is returned as a guard for the appended "emit S" statement. Therefore no cycle is detected in this example.

If one would want to make the cycle detection scheme presented here compliant to the CEC, one possibility is the relaxing of cycle detection in the transformation. This would involve adding the signals in a present predicate to the guard set returned by that block if it is connected to a synchronizer. Another possibility would be the addition of Potop-Butucaru's proposal for a modified synchronizer to the CEC code synthesis. The latter approach has bigger potential for a more efficient code generation.

The second example CYCLE_v5 in Figure 3.6(b) returns the signal A in the first thread and no signal in the second thread containing the pause statement. The result is a guard of A for the "emit A" statement which constitutes a cycle.

The cycle searching algorithm does not take trap priorities into account. Therefore the third example CYCLE_TRAP in Figure 3.6(c) returns a dependency from A to B in the first instant before the pause statement. Together with the dependency from B to A in the second part behind the pause statement, a cycle is detected.

This additional cycle will be resolved by the actual transformation step presented in Chapter 4. The behavior of the program will be conserved, but a certain amount of overhead is introduced by this needless transformation.

3.4 Finding Cyclic Dependencies

The previous sections gave some informal introduction into different kinds of cyclic signal dependencies in Esterel programs on some examples. Before being able to resolve such cycles algorithmically, details on a method to identify such cycles are needed. The algorithm presented here is divided into two parts:

1. Identification of all direct dependencies between signals.

2. Searching for cycles in signal dependencies.

$P =$		
emit S	$\mathcal{G}(P,G,W,X) = G$ $\mathcal{D}(P,G,W,X) = \{\langle a,S\rangle \mid a \in G\}$	(3.1)
present S **then** p **else** q **end**	$\mathcal{G}_p \equiv \mathcal{G}(p, G \cup \{S\}, W, X) \qquad \mathcal{G}_q \equiv \mathcal{G}(q, G \cup \{S\}, W, X)$ $\mathcal{G}(P,G,W,X) = \begin{cases} \bot & : (\mathcal{G}_p = \bot) \wedge (\mathcal{G}_q = \bot) \\ \mathcal{G}_p & : (\mathcal{G}_p \neq \bot) \wedge (\mathcal{G}_q = \bot) \\ \mathcal{G}_q & : (\mathcal{G}_p = \bot) \wedge (\mathcal{G}_q \neq \bot) \\ \mathcal{G}_p \cup \mathcal{G}_q & : \text{otherwise} \end{cases}$ $\mathcal{D}(P,G,W,X) = \mathcal{D}(p, G \cup \{S\}, W, X) \cup \mathcal{D}(q, G \cup \{S\}, W, X)$	(3.2)
pause	$\mathcal{G}(P,G,W,X) = W$ $\mathcal{D}(P,G,W,X) = \{\langle a,s\rangle \mid a \in G, s \in X\}$	(3.3)
halt	$\mathcal{G}(P,G,W,X) = \bot$ $\mathcal{D}(P,G,W,X) = \{\langle a,s\rangle \mid a \in G, s \in X\}$	(3.4)
nothing	$\mathcal{G}(P,G,W,X) = G$ $\mathcal{D}(P,G,W,X) = \emptyset$	(3.5)
loop p **end**	$\mathcal{G}_p \equiv \mathcal{G}(p,G,W,X)$ $\mathcal{G}(P,G,W,X) = \bot$ $\mathcal{D}(P,G,W,X) = \begin{cases} \mathcal{D}(p,G,W,X) & : \mathcal{G}_p = \bot \\ \mathcal{D}(p, G \cup \mathcal{G}_p, W, X) & : \text{otherwise} \end{cases}$	(3.6)
p ; q	$\mathcal{G}_p \equiv \mathcal{G}(p,G,W,X)$ $\mathcal{G}(P,G,W,X) = \begin{cases} \bot & : \mathcal{G}_p = \bot \\ \mathcal{G}(q, \mathcal{G}_p, W, X) & : \text{otherwise} \end{cases}$ $\mathcal{D}(P,G,W,X) = \mathcal{D}(p,G,W,X) \cup \begin{cases} \emptyset & : \mathcal{G}_p = \bot \\ \mathcal{D}(q, \mathcal{G}_p, W, X) & : \text{otherwise} \end{cases}$	(3.7)
trap T **in** p **end**	$\mathcal{G}_p \equiv \mathcal{G}(p, G, W, X \cup \{T\})$ $\mathcal{G}(P,G,W,X) = \{a \mid \langle a,T\rangle \in \mathcal{D}(p,G,W,X \cup \{T\})\} \cup \begin{cases} \emptyset & : \mathcal{G}_p = \bot \\ \mathcal{G}_p & : \text{otherwise} \end{cases}$ $\mathcal{D}(P,G,W,X) = \{\langle a,s\rangle \in \mathcal{D}(p,G,W,X \cup \{T\}) \mid s \neq T\}$	(3.8)
exit T	$\mathcal{G}(P,G,W,X) = \bot$ $\mathcal{D}(P,G,W,X) = \{\langle a,T\rangle \mid a \in G\}$	(3.9)
p || q	$\mathcal{G}(P,G,W,X) = \begin{cases} \bot & : (\mathcal{G}(p,G,W,X) = \bot) \vee (\mathcal{G}(q,G,W,X) = \bot) \\ \mathcal{G}(p,G,W,X) \cup \mathcal{G}(q,G,W,X) & : \text{otherwise} \end{cases}$ $\mathcal{D}(P,G,W,X) = \mathcal{D}(p,G,W,X) \cup \mathcal{D}(q,G,W,X)$	(3.10)

Figure 3.7: First part of the equations to determine signal dependencies. See Figure 3.8 for the second part. $\mathcal{D}$ collects signal dependencies from signal emissions, $\mathcal{G}$ returns the active guard signals from terminating statement blocks. Parameter P is the given program fragment shown on the left, G the set of guard signals, W the set of signals in suspend/abort expressions, and X the set of trap signals in the current context.

$P =$

signal S **in**
  p
end

$$\mathcal{G}(P,G,W,X) = \begin{cases} \bot & : \mathcal{G}(p,G,W,X) = \bot \\ \mathcal{G}(p,G,W,X) & : \text{otherwise} \end{cases} \qquad (3.11)$$
$$\mathcal{D}(P,G,W,X) = \mathcal{D}(p,G,W,X)$$

---

suspend
  p
when S

$$\mathcal{G}_p \equiv \mathcal{G}(p,G,W \cup \{S\},X)$$
$$\mathcal{G}(P,G,W,X) = \begin{cases} \bot & : \mathcal{G}_p = \bot \\ \mathcal{G}_p \cup \{S\} & : \text{otherwise} \end{cases} \qquad (3.12)$$
$$\mathcal{D}(P,G,W,X) = \mathcal{D}(p,G,W \cup \{S\},X)$$

---

abort
  p
when S

$$\mathcal{G}_p \equiv \mathcal{G}(p,G,W \cup \{S\},X)$$
$$\mathcal{G}(P,G,W,X) = \begin{cases} \bot & : \mathcal{G}_p = \bot \\ \mathcal{G}_p \cup \{S\} & : \text{otherwise} \end{cases} \qquad (3.13)$$
$$\mathcal{D}(P,G,W,X) = \mathcal{D}(p,G,W \cup \{S\},X)$$

Figure 3.8: Second part of the equations to determine 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 are interpreted as a directed graph with the signal names as nodes and the set of signal pairs defining directed edges between the nodes. The task of the second part of the algorithm is to identify those nodes which are cyclically connected by directed edges. Those nodes comprise the cyclic dependency.

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 S). 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 P statement is contained in a sub-block 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
```

$P =$		
emit S	$\mathcal{G}(P,G,W,X) = G$ $\mathcal{D}(P,G,W,X) = \{\langle a,S\rangle \mid a \in G\}$	(3.14)
present S **then** p **else** q **end**	$\mathcal{G}(P,G,W,X) = \mathcal{G}(p,G \cup \{S\},W,X) \cup \mathcal{G}(q,G \cup \{S\},W,X)$ $\mathcal{D}(P,G,W,X) = \mathcal{D}(p,G \cup \{S\},W,X) \cup \mathcal{D}(q,G \cup \{S\},W,X)$	(3.15)
pause	$\mathcal{G}(P,G,W,X) = W$ $\mathcal{D}(P,G,W,X) = \{\langle a,s\rangle \mid a \in G, s \in X\}$	(3.16)
halt	$\mathcal{G}(P,G,W,X) = \emptyset$ $\mathcal{D}(P,G,W,X) = \{\langle a,s\rangle \mid a \in G, s \in X\}$	(3.17)
nothing	$\mathcal{G}(P,G,W,X) = G$ $\mathcal{D}(P,G,W,X) = \emptyset$	(3.18)
loop p **end**	$\mathcal{G}(P,G,W,X) = \emptyset$ $\mathcal{D}(P,G,W,X) = \mathcal{D}(p,G \cup \mathcal{G}(p,G,W,X),W,X)$	(3.19)
p ; q	$\mathcal{G}(P,G,W,X) = \mathcal{G}(q,\mathcal{G}(p,G,W,X),W,X)$ $\mathcal{D}(P,G,W,X) = \mathcal{D}(p,G,W,X) \cup \mathcal{D}(q,\mathcal{G}(p,G,W,X),W,X)$	(3.20)
trap T **in** p **end**	$\mathcal{G}(P,G,W,X) = \{a \mid \langle a,T\rangle \in \mathcal{D}(p,G,W,X \cup \{T\})\} \cup \mathcal{G}(p,G,W,X \cup \{T\})$ $\mathcal{D}(P,G,W,X) = \{\langle a,s\rangle \in \mathcal{D}(p,G,W,X \cup \{T\}) \mid s \neq T\}$	(3.21)
exit T	$\mathcal{G}(P,G,W,X) = \emptyset$ $\mathcal{D}(P,G,W,X) = \{\langle a,T\rangle \mid a \in G\}$	(3.22)
p || q	$\mathcal{G}(P,G,W,X) = \mathcal{G}(p,G,W,X) \cup \mathcal{G}(q,G,W,X)$ $\mathcal{D}(P,G,W,X) = \mathcal{D}(p,G,W,X) \cup \mathcal{D}(q,G,W,X)$	(3.23)
signal S **in** p **end**	$\mathcal{G}(P,G,W,X) = \mathcal{G}(p,G,W,X)$ $\mathcal{D}(P,G,W,X) = \mathcal{D}(p,G,W,X)$	(3.24)
suspend p **when** S	$\mathcal{G}(P,G,W,X) = \mathcal{G}(p,G,W \cup \{S\},X)$ $\mathcal{D}(P,G,W,X) = \mathcal{D}(p,G,W \cup \{S\},X)$	(3.25)
abort p **when** S	$\mathcal{G}(P,G,W,X) = \mathcal{G}(p,G,W \cup \{S\},X)$ $\mathcal{D}(P,G,W,X) = \mathcal{D}(p,G,W \cup \{S\},X)$	(3.26)

Figure 3.9: Simplified equations from Figure 3.7/3.8 to determine signal dependencies without skipping of unreachable code. If the input program does not contain any suspend/abort statements then rules (3.25) and (3.25) can be omitted and any reference to W replaced by $\emptyset$.

The emission of P is followed by a pause statement and an emission of signal emit Q. The pause statement defers the emission of Q to the next instant from the instant where S is tested. Therefore the emission of Q is not influenced by the state of S in their respective instants and so 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 program 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 non-speculative execution of the program and the execution of present must be held back until the state of all the tested signals is established. Therefore signal dependencies can be established across sequential execution of statements, too.

Further details of deriving signal dependencies off Esterel programs are given in the next section on the dependency algorithm.

3.4.1 Algorithm to Identify Signal Dependencies

The concept of signal guards is used to identify all signal dependencies in an Esterel program. While traversing the syntactical elements of the program, three sets of currently active guard signals are maintained and signal dependencies are derived from signal emissions.

Given an an Esterel Program P with signals Σ, the rules to implement both tasks operate on three signal sets:

- $G \subset \Sigma$: Set of signals (*Guards*) comprising the current present conditions.
- $W \subset \Sigma$: Set of signals (*Watchers*) contained in the currently active suspend and abort conditions.
- $X \subset \Sigma$: Set of trap signals (*Exceptions*) in the current scope.

Signals in G (the present conditions) do not reach across pause statements and are therefore deleted when traversing such statements. The signals added to W inside suspend and abort blocks are tested for the entire runtime of those blocks, and they must not be removed while traversing pause statements. Effectively the content of G is initialized

with the content of W at every pause statement. This implements the delayed nature of suspend and abort conditions.

The signals in X are not regular signals but trap signals. This set is used to express the extension of range for guards in exceptions in the context of parallel threads.

The rules to derive signal dependencies from Esterel programs are implemented in two functions (with Π as set of Esterel programs and Σ the set of signals):

- $\mathcal{D} : \Pi \times 2^{\Sigma} \times 2^{\Sigma} \times 2^{\Sigma} \rightarrow 2^{\Sigma \times \Sigma} \qquad (P, G, W, X) \mapsto \mathcal{D}(P, G, W, X)$
  This function 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.
- $\mathcal{G} : \Pi \times 2^{\Sigma} \times 2^{\Sigma} \times 2^{\Sigma} \rightarrow 2^{\Sigma} \qquad (P, G, W, X) \mapsto \mathcal{G}(P, G, W, X)$
  $\mathcal{G}$ returns the set of guard signals that are active when the program block P terminates. If control cannot leave P, then $\perp$ is returned. This is used to identify and efficiently handle unreachable code.

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

The actual rules for all Esterel kernel statements are listed in Figure 3.7 on page 38 and Figure 3.8 on page 39. Some remarks on the application of the rules are stated in the following.

Figure 3.9 on page 40 contains a simplified variant: The handling of dead code is missing. This rule set is useful for an overview on the principle of finding signal dependencies without the complications introduced by dead code. If dead code in input programs is not present in the first place, is removed by other means, or known to not contain dependencies as part of a cycle then the rule set in Figure 3.9 instead of Figures 3.7/3.8 can be used without loss of efficiency. Otherwise the following transformation would potentially resolve cycles that are not rejected by Esterel compilers.

Rule 3.1 (emit) applies to signal emissions. $\mathcal{G}$ returns the current guard unchanged. $\mathcal{D}$ takes the current guard set G and returns a dependency pair for each signal to the emitted signal.

Rule 3.2 (present) handles the conditional execution of sub-blocks (*e. g.*, either p or q). $\mathcal{D}$ returns the union of the independent results on p and q. For the evaluation of p and q, the same current signal guard set is used, extended by the signals from the present condition. The signal guard set at the point of termination consists of the union of guard sets returned by either terminating branch. If neither branch terminates, then the entire present block is considered non-terminating and $\mathcal{G}$ returns $\perp$.

The following code gives an example for the kind of dependencies found by Rules 3.1 and 3.2. Rule 3.2 extracts A as the dependency source used in Rule 3.1 to add a dependency to B and C.

```
    present A then
      emit B
P = else                 ⇝    D(P,∅,∅,∅) = {⟨A,B⟩,⟨A,C⟩}
      emit C
    end;
```

Rule 3.3 (pause) handles the separation of instants at program execution, therefore the currently valid guards are invalid after execution of the pause statement. $\mathcal{G}$ deletes the guard set by resetting it to the content of W. Details on the purpose of the W set are given in the remarks on Rule 3.12/3.13.

In a variant of the previous example a pause statement is added to the else branch of the present statement. This breaks the dependency from A to C and only the dependency to B remains:

```
    present A then
      emit B
P = else                 ⇝    D(P,∅,∅,∅) = {⟨A,B⟩}
      pause;
      emit C
    end;
```

Function $\mathcal{D}$ for pause relates to the use of traps to terminate parallel threads. The remarks on Rule 3.10 address the handling of trap signals in this context.

Rule 3.5 (nothing) does nothing. $\mathcal{G}$ just returns the current guard set and $\mathcal{D}$ produces no dependency pairs.

Rule 3.6 (loop) introduces a difficulty: Unreachable or *dead* code. The body of a loop is indefinitely repeatedly executed, unless an included exit statement hands control over to a surrounding trap exception handler. However, the code located immediately in sequence of a loop statement is not reachable. Therefore any statements sequentially following a loop statement do not produce any signal dependencies. This is not correctly reflected by, *e. g.*, returning an empty guard set for the pause statement. Unreachable code is marked here with a $\perp$ symbol, instead.

The body of a loop statement needs to be evaluated twice, to capture signal dependencies which are wrapped around the end of the loop body. Rule 3.6 implements this for the first iteration as an application of $\mathcal{D}$ on the loop body p to obtain the guard set active at termination of p. That guard set is added to the guard set active at the start of the loop statement. The resulting guard set is used to derive the signal dependencies in p in the second iteration.

The following example contains a dependency from A to B. The dependency wraps around the end of the loop statement:

```
      loop
        emit B;
        pause;
P =     present A then        ⇝    D(P,∅,∅,∅) = {⟨A,B⟩}
          nothing
        end
      end
```

The emission of B is not added as a dependency to A in the first iteration, because the test on A is not yet evaluated. The second iteration will register B as dependent on A.

Rule 3.7 (;) handles signal dependencies introduced by sequential execution of code blocks (*e. g.*, p before q). The analysis of p is the easy part: Function $\mathcal{D}$ is applied to the code of p with the current guard set G. The problematic part is the analysis of q: Beforehand it must be known if p cannot terminate. In that case q is unreachable and is considered to produce no signal dependencies. If p is able to terminate, then the active guard set at the termination of p is needed for the application of $\mathcal{D}$ on q. Both questions are answered by the function $\mathcal{G}$. Incidentally the hypothetical abandonment of the sequence operator would eliminate the need for the function $\mathcal{G}$ altogether.

The following code contains a present statement with empty then and else branches. Nevertheless the present can only be executed when the status of A is known. After the termination of present the emit statement is executed in sequence and therefore adds a dependency from A to B:

```
      present A then
        nothing
P =   end;                    ⇝    D(P,∅,∅,∅) = {⟨A,B⟩}
      emit B
```

If both branches of a present execute pause statements, then the sequential dependency is broken. This behavior is implemented in the definition of $\mathcal{G}$ in Rules 3.2 and 3.3:

```
      present A then
        pause
      else
P =     pause                 ⇝    D(P,∅,∅,∅) = ∅
      end;
      emit B
```

Rule 3.8 (trap) and **Rule 3.9 (exit)** are tightly coupled. The basic problem encountered when formulating these rules was the handling of *exceptional* control flow for exit statements in p. The regular recursive traversal of the trap body would only catch the guard set for the *regular* termination of p. But when an exit statement is encountered inside p, then the guard set present at the point of exit must be added to the guard set at termination of the entire trap statement.

The solution proposed here adds the trap signal T to the list of signal dependencies. This connects each signal in the current guard set G as a guard for the trap signal, and

"tunnels" the guard set from the exit statement to the termination of the trap body. At that point the exit guard is extracted back from the signal dependency set and added to the guard set computed for the regular termination.

At the end of the evaluation of a trap statement, all references to the trap signal T are removed from the dependency set. But this is just cosmetical, because the trap signal T is never tested like a regular signal and therefore will not interfere in the search for cycles in signal dependencies.

The following example contains two signal dependencies. The dependency from B to C is introduced by regular termination of the trap body. Another dependency connects A via the exit statement to B:

```
      trap T in
        present A then
          exit T
        end;
P =     pause;                  ⇝    D(P, ∅, ∅, ∅) = {⟨A, C⟩, ⟨B, C⟩}
        present B then
          nothing
        end
      end;
      emit C
```

Rule 3.10 (||) handles the parallel execution of threads (*e. g.*, p and q). It is similar to Rule 3.2 on the present statement. Both threads are evaluated independently and the results unified. The main difference is in the handling of non-terminating p or q. For present it is sufficient if either one terminates to consider the whole present block to potentially terminate. The parallel block terminates when all threads are terminated, therefore if one thread does not terminate, the whole parallel block must be considered non-terminating.

This treatment of the termination of parallel blocks is a conservative simplification of the synchronizer circuit in the circuit semantics of Esterel. Figure 3.6 on page 36 contains examples where this makes a difference.

Signal dependencies must be recognized across the termination of parallel threads. The following (simple) example contains two dependencies to C:

```
      [
          present A then nothing end
P =     ||                                ⇝    D(P, ∅, ∅, ∅) = {⟨A, C⟩, ⟨B, C⟩}
          present B then nothing end
      ];
      emit C
```

Complications are introduced by termination of parallel threads by activating a trap exception. In a regular control flow the set of guard signals is emptied on encountering a pause statement because the execution of the instant ends there. This limits the influence

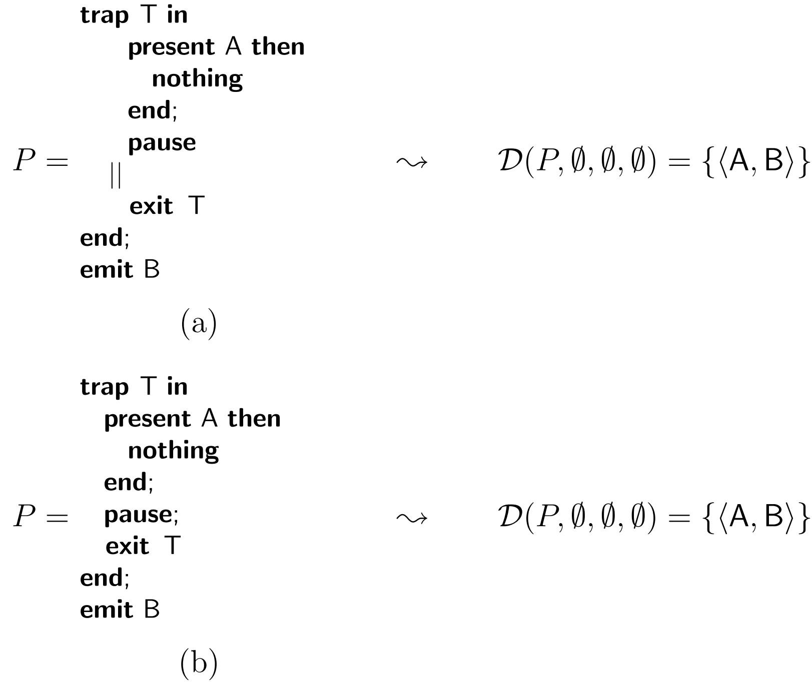


Figure 3.10: Dependencies introduces by exception handling: (a) Dependency from A to B because of an exit in a parallel thread, (b) statically unreachable dependency.

of a guard set up to the next pause statement. But if in a parallel thread an exit statement is executed, then the control point is moved to the end of the corresponding trap block. This extends the reach of a guard beyond the pause statement.

Such a case is pictured in the example in Figure 3.10(a). The present test in the first thread adds A to the guard and the execution stops at the pause statement. But the exit statement in the second thread moves the control to the end of the trap block, which adds a dependency to B.

The solution to the problem which is proposed here is to not simply discard the guard set at a pause statement. Instead of this the signals contained in the guard are stored as dependencies to all trap signals in the current scope. This is implemented in Rule 3.3 on function $\mathcal{D}$ for the pause statement.

The additional signal set X ("eXception") is used to store the set of trap signals in the current scope. It is set up in Rule 3.8 on entering the body of a trap statement.

A drawback of this simple method is that trap dependencies are added even for clearly unreachable control flows. That is this method does not return the minimal set of dependencies regarding the cycles which are rejected by Esterel compilers. The dependency returned for the example in Figure 3.10(b) is just an artefact of the simplicity of the algorithm. Future refinements of the dependency detection algorithm may improve on this aspect.

Rule 3.11 (signal) does not remove the locally defined signal S from the guard set,

```
signal S in
  ...
  present S then
    pause
  end;
end;
emit B;
```

(a)

```
signal S in
  present A then
    emit S
  end;
  present S then
    pause
  end
end;
emit B;
pause;
present B then
  emit A
end
```

(b)

Figure 3.11: Signal dependency extending outside the scope of a local signal: (a) Dependency from local signal S to B leaving scope of S, (b) constructive cyclic dependency from A over local signal S to B.

because the local signal may carry a signal dependency outside the scope of the signal.

Consider the examples in Figure 3.11: The termination of the signal block in the program fragment in Figure 3.11(a) depends on the status of the local signal S. Therefore a dependency exists from S to the signal B emitted following the signal block. If Rule 3.11 would remove S from the guard set when leaving the signal block, then the dependency from S to B would be missed. The example in Figure 3.11(b) contains a complete (constructive) cyclic dependency from A over S to B and back to A. Part of that cycle is routed outside the scope of the locally defined signal S.

This "leaking" of signal names out of their respective scopes is acceptable to detect the presence of cyclic dependencies, but it is not for resolving those cycles. Therefore the transformation algorithm in Chapter 4 suggests in Step 2c (Figure 4.2, page 53) the removal of locally defined signals. This treatment of local signal definitions is definitely not needed in all cases, and removal of local signals can be quite expensive in code size. Therefore an optimization can be applied to this rule, see Section 6.11.

Rule 3.12 (suspend) and **Rule 3.13 (abort)** are handled in the same way. The suspend/abort signal is added to the watcher signal set W to analyze the sub-block. As noted before, a distinct watcher set W is used for the signals in the suspend/abort condition beside the regular guard set G. The dependencies to signals in G will be removed by pause statements, but the dependencies to suspend/abort signals are present for the entire run of the sub-block and therefore must not be deleted. This is implemented by copying the content of W over into G at every evaluation of pause in Rule 3.3.

The condition of suspend/abort is not evaluated in the very first instant entering the body. Therefore no dependencies to the suspend/abort condition are produced for the first instant. This restriction is honored, since the copying of W took not yet place in the first instant entering the body of suspend/abort.

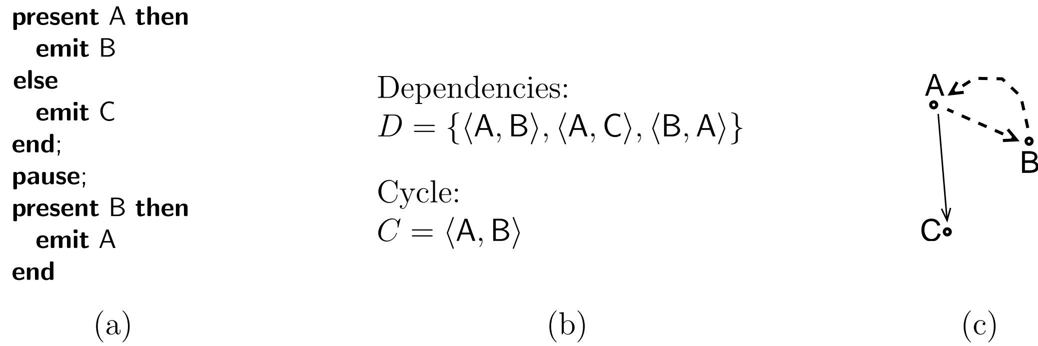


Figure 3.12: Finding (cyclic) signal dependencies: (a) Example program, (b) Application of functions $\mathcal{D}()$ and algorithm find_shortest_cycle(), (c) Graphical representation of signal dependencies.

Such a configuration is presented in the following example. The watcher signal C is a guard for B even though B is located behind a pause statement:

```
      suspend
        emit A;
P =     pause;      ⇝      D(P, ∅, ∅, ∅) = {⟨C, B⟩}
        emit B;
      when C
```

Signal A does not depend on C because of the delayed nature of suspend/abort conditions.

The rule for abort is listed here despite of abort not being a kernel statement of Esterel. But abort is frequently used (much more so than suspend) and therefore worth handled directly without previous expansion into kernel statements.

The algorithm presented in Chapter 4 (Figure 4.2, page 53) on the actual resolving of cyclic dependencies substitutes in Step (3a) all suspend statements by means of other kernel statements. Nevertheless the search for signal dependencies supports the suspend statement directly because in the absence of cyclic dependencies the substitution of suspend is not needed.

3.4.2 Searching for Cycles in Signal Dependencies

The algorithm described in Section 3.4.1 computes all signal dependencies in a given Esterel program. The result is a set of pairs of signal names describing all dependencies between signals. Figure 3.12 contains an example for such an analysis.

Formally: Given the set of signals Σ in the Esterel program P, the signal dependencies are returned as a set D of pairs by function $\mathcal{D}(p, \emptyset, \emptyset, \emptyset)$:

$$D = \{\langle a_0, b_0\rangle, ..., \langle a_n, b_n\rangle\}, \quad a_0, ..., a_n, b_0, ..., b_n \in \Sigma$$

A cycle of length $l \in \mathbf{N}$ is defined as a chain $\vec{C}$ of signals connected by dependencies:

```
1  type Signal = struct {
2      string     name,
3      SignalSet  depend,
4      boolean    tag
5    };
6
7  type SignalSet = set Signal;
```

```
1  SignalSet
2    find_shortest_cycle (SignalSet Σ )
3  {
4    SignalSet C_min := ∅;
5    int n := |Σ|;
6
7    forall s ∈ Σ do {
8      SignalSet C :=
9                find_connection (n,s,s);
10     if (C ≠ ∅ and |C| <n) then {
11       C_min := C;
12       n := |C|;
13     }
14   }
15
16   return C_min;
17 }
```

```
1  SignalSet
2    find_connection (int n, Signal s, Signal t)
3  {
4    SignalSet C_tail := ∅;
5
6    s.tag := true;
7    forall d ∈ s.depend {
8      if (d = t) then {
9        C_tail := {d};
10       n := 1;
11     } else {
12       if (not d.tag and n>1) then {
13         SignalSet C :=
14                   find_connnection (n−1,d,t);
15         if (C ≠ ∅ and |C| <n) then {
16           C_tail := C;
17           n := |C|;
18         }
19       }
20     }
21   }
22   if ( C_tail ≠ ∅) then {
23     C_tail := C_tail ∪ {s};
24   }
25   s.tag := false;
26
27   return C_tail ;
28 }
```

Figure 3.13: Algorithm to find a shortest cycle in signal dependencies by following recursively all connections. If no cycle is found ∅ is returned.

$$\vec{C} = \langle c_0, ..., c_{l-1} \rangle, \quad c_0, ..., c_{l-1} \in \Sigma$$
$$\forall i \in \{0, ..., l-1\} : \langle c_i, c_{(i+1) \bmod l} \rangle \in D$$

If multiple cycles are present in D then a *smallest* cycle must be found for reasons laid out in Step (6c) on page 59. The search for such cycles in the signal dependencies is fairly easily done with standard algorithms [34]. Nevertheless a specialized algorithm is presented here in Figure 3.13 on page 49 in a pseudo code notation.

The data structure used to store the signal dependencies for the algorithm is a little bit different than the one used in the previous sections. There is one structure defined for each signal s storing several attributes. The attributes used in the algorithm in Figure 3.13 are:

- s.depend: Set of references to other signals depending on s.
  s.depend $= \{t \in \Sigma \mid \langle s, t \rangle \in D\}$.
- s.tag: Boolean flag to indicate already visited paths.

The function find_shortest_cycle(Σ) is the entry point of the algorithm. It iterates over all signals Σ and searches for the shortest connection over dependencies back to that signal. To improve on the efficiency of the algorithm an integer variable n stores the size of the currently shortest cycle found. It makes no sense to follow longer paths than this size, because cycles found on those paths would be longer than the one already found.

The actual recursive search for cycles is implemented in the function find_connection(n,s,t). It receives as parameters n to limit the search depth, the current signal under test s, and checks recursively if the signal s is connected to t:

- Line 4: The signal set C_tail stores the shortest connection from s to t found so far.
- Line 6 and 25: Each visited signal is marked to avoid multiple iterations on the same signal. That mark is removed in Line 25 when the recursion unfolds not to interfere with iterations on different cycles.
- Line 7: All signals depending on s are tested whether they connect back to t.
- Line 8-10: These lines decide over the presence of a cycle. If t is encountered while following all dependencies from s, then a cycle has been found. This connection is stored in C_tail. n is updated to reflect the new shortest connection to t.
- Line 12: Reaching an already tagged signal does indicate a cycle, but the current tags are not limited to the cycle at this point. Therefore the recursion skips already marked signals. Additionally further recursions are inhibited if a direct connection to t with length one had already been found on this recursion level.
- Line 13/14: A connection from d to t is searched for recursively with a search limit decreased by one. The result is stored in a local signal set C.
- Line 15-17: If d provides a shorter connection from s to t then it is stored as a new minimal connection in C_tail.
- Line 22/23/27: If a connection from s to t had been found then s is added to C_tail, too. The resulting connection set is returned to the upper recursion levels.

The result returned by the function find_shortest_cycle(Σ) is a set of signals contained in the shortest cycle found in the signal dependencies. That set is empty if no cycle is present. The actual order of signals in the cycle is not needed in further transformation processing and therefore not preserved in the set returned by find_shortest_cycle(Σ).

Chapter 4

Program Transformation

After identifying cyclic dependencies in the previous chapter we are now able to resolve those cycles by application of the algorithm presented in this chapter.

4.1 The Base Transformation Algorithm

Figure 4.1 introduces the notations which will be used for the transformation. Figure 4.2 presents the algorithm for transforming cyclic Esterel programs into acyclic programs. The algorithm is applicable to programs with cycles that involve pure signals only. This section discusses each transformation step along with it's worst-case increase in code size.

Step (1): Constructiveness
The constructiveness of the Esterel program is a precondition for the transformation. This analysis can be performed using the methods developed by Shiple, Berry *et al.* [36, 3]; one available implementation is offered by the v5 compiler [18]. The key property of constructiveness is that no causality cycle makes the state of a signal dependent on itself. All signals are only dependent on the current program state, other signals, or inputs from the environment. It is sufficient for this property to hold only for program states which are *reachable* at runtime.

This absence of self dependencies is exploited in Step (6d) of the algorithm.

Conversely, the fact that the transformation is valid if and only if the transformed program is constructive can be exploited to employ this transformation to aid in constructiveness analysis in the first place; see also the comments on Step (6d) on page 62, and in Section 4.5 on page 80.

Step (2): Preprocessing
The core algorithm is only applicable to Esterel programs restricted in certain ways,

Basics

$\mathbf{N}$: Set of natural numbers (including zero)

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

P: Given Esterel Program

Σ: Set of signals used in P

Signals

$\Sigma_{\vec{C}}$: Set of original cycle signals

σ_i': Fresh signal used to replace emission of input signal σ_i in P

$\Sigma_{\vec{C}'}$: Set of cycle signals derived from $\Sigma_{\vec{C}}$ with input signals renamed

ST_i: State signals used to access the program state in guards

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

Dependencies

G: Set of signals ("Guard") involving signals $\Sigma_i \subseteq (\Sigma \cup ST)$.

W: Set of signals off suspend/abort conditions ("Watcher") involving signals $\Sigma_i \subseteq \Sigma$.

X: Set of trap signals in the current scope.

$\mathcal{G} : P \times G \times W \rightarrow G'$: Computes active guard signals at program termination.

$D = \{\langle \sigma_i, \sigma_j \rangle \mid i, j \in \mathbf{N}\}$: Dependencies of signals σ_i to σ_j.

$\mathcal{D} : P \times G \times W \rightarrow D$: Function to compute signal dependencies from P.

Cycles

$\sigma_i \in \Sigma_{\vec{C}}$: Signal part of a cyclic dependency

$\vec{C} = \langle \sigma_i \mid i \in \mathbf{N}_l \rangle$, with $l \in \mathbf{N}$: Cycle of length l

$\forall \sigma_i \in \vec{C} : \langle \sigma_i, \sigma_{(i+1) \bmod l} \rangle \in D$: Cycle property

$\sigma_p \in \Sigma_{\vec{C}}$: Pivot signal selected to break the cyle

Emission context

S_i: Boolean context expression ("signal state context") involving signals $\Sigma_i \subseteq (\Sigma \cup ST)$.

$\mathcal{S} : P \times S \rightarrow S$: Function to compute context expressions at program termination.

$E = \{\langle \sigma_i, S_j \rangle \mid \sigma_i \in \vec{C}\}$: All emissions of cycle signals in their respective guard context.

$\mathcal{E} : P \times S \rightarrow E$: Function to compute context expressions for signal emissions.

$E_i = \bigvee_{\langle \sigma_i, S_j \rangle \in E} S_j$: or'ed guards of signal σ_i, describing its complete emission context.

E_i^*: Derived from E_i by iterative replacement of $\sigma_j \in (\Sigma_{\vec{C}} \setminus \sigma_i)$ by E_j.

E_i^{**}: Derived from E_i^* by replacement of σ_i by false (or true).

Figure 4.1: Notation summary.

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 abort which is handled in Step (3a).
    (c) Rename locally defined signals to make them unique and lift the definitions up to the top level. Furthermore, eliminate signal reincarnation. (See Section 6.11)
3. Introduce state signals:
    (a) Transform suspend and abort into equivalent present/trap statements.
    (b) Add explicit termination handling to || statements. (See Fig. 4.4)
    (c) Add boot register as a new global signal ST_0 and add "emit ST_0;" to the start of the program body.
    (d) Enumerate all pause statements starting from 1 and do for all pause_i:
        i. Globally declare a new signal ST_i.
        ii. Replace pause_i by "pause; emit ST_i."
4. Identification of cyclic signal dependencies:
    (a) Identify all signal dependencies: Compute $D = \mathcal{D}(P, \emptyset, \emptyset, \emptyset)$ (see Fig. 3.7/3.8).
    (b) Search for cycles in D: Compute find_shortest_cycle() (see Fig. 3.13).
    (c) If P does not contain cycles: **Done**.
    Otherwise: Select a shortest cycle $\vec{C}$, of length l.
5. 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')."
6. Transform (still cyclic) P' into (acyclic) P'':
    (a) For all $\sigma_i \in \vec{C}$ determine replacement expressions $E_i = \mathcal{E}(P', \mathsf{ST_0})$ (see Fig. 4.7).
    (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 cyclic signals.
    (e) Replace all tests for σ_p in P' by E_p^{**}.
7. Goto Step (4), treat P'' now as P.

Figure 4.2: Transformation algorithm, for pure signals.

requiring the following preprocessing steps:

Step (2a): Module expansion
The expansion of modules is a straightforward textual replacement of module calls by their respective body. No dynamic runtime structures are needed, since Esterel does not allow recursions.

The complexity of this module expansion can reach exponential growth of code size, but this expansion is done by every Esterel compiler and not a special requirement of this transformation algorithm. Hence the baseline for an analysis of the code increase introduced by the transformation presented here is the size of the original program after module expansion.

Step (2b): Non-kernel Statements
Regarding the statements handling signals, the transformation algorithm is expressed in terms of Esterel kernel statements. Therefore statements that are derived from emit, present, or suspend must be reduced to these statements.

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

Step (2c): Local Signals
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, *i. e.*, not re-incarnated. A simple approach to eliminate reincarnation is based on loop-unrolling, which results in a potentially exponential increase in code size; using other techniques, this can be reduced to a quadratic increase [3], or even lower complexity by the introduction of a gotopause statement [37].

This treatment of local signal definitions is clearly not needed in all cases, and removal of local signals can be quite expensive in code size. Therefore an optimization can be applied to this rule, see Section 6.11.

Steps (3): State signals
The current execution state of an Esterel program is stored in registers/variables defined inside the synthesized code/circuit. Unfortunately there is no provision at the Esterel level to access the state of these registers/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 ST_i. (Note that many of the signals may be eliminated again by subsequent optimizations, see Chapter 6.)

An important property of the introduced state signal is that they are free of any dependencies. However, they are in turn the source of dependencies for the emission of other signals, and the transformation algorithm exploits this property to determine the emission of signals without introducing new signal dependencies.

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

Step (3a): suspend/abort

The introduction of state signals fails in the context of suspend/abort statements, because state signals emitted as part of a suspend/abort block are suppressed, too. This constitutes a dependency of the state signals on the suspension/abortion condition and invalidates the dependency-less emission of state signals. That influence introduces unwanted dependencies, potentially forming a new cyclic dependency.

The solution proposed here simulates the behavior of abort/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:

```
suspend
  p              ⇝        p'
when S
```

Here p denotes the suspended body and S the suspension condition. They are replaced by just 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; "await not S" then becomes:

```
                          trap T in
                            loop
                              pause;
await not S       ⇝           present S else
                                exit T
                              end
                            end
                          end
```

The following transformation step (Step 3d) will add state signals to the newly introduced pause statements. Those state signals are placed before the checks of the suspension condition and therefore not dependent on signals contained in the suspension condition.

An example program with suspend statements is discussed in Section 4.4.5 on page 78.

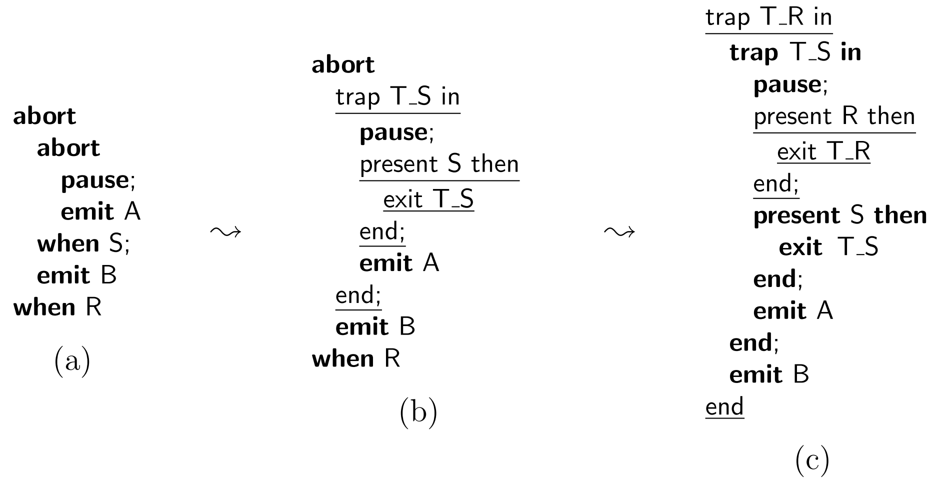


Figure 4.3: Preserving the priorities between cascaded abort statements: (a) original program, (b) the inner abort statements is transformed first, (c) the outer abort is transformed last.

The Esterel statement abort poses similar problems for state signals. The transformation algorithm is designed in its current form for kernel statements only, but the (non-kernel) abort statement is widely used and the conventional replacement expression in kernel statements is quite voluminous. This especially so because it contains suspend, which must be eliminated additionally. Therefore this proposal transforms the abort statement directly.

A construction consisting of a trap block with exit statements added to each pause statement is used inside the abortion block:

```
abort                trap T in
  p          ⇝         p'
when S               end
```

where p' is derived from p by performing the following replacement for each pause statement:

```
                     pause;
pause        ⇝       present S then
                       exit T
                     end
```

When transforming cascaded abort blocks, then the innermost abort statements must be tranformed first and the outermost last. This preserves the priorities of abort conditions, stating that the outer abort statements take priority over inner statements. An example of such a transformation of cascaded abort blocks is listed in Figure 4.3.

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

An alternative solution to handle state signals inside abort/suspend blocks is proposed

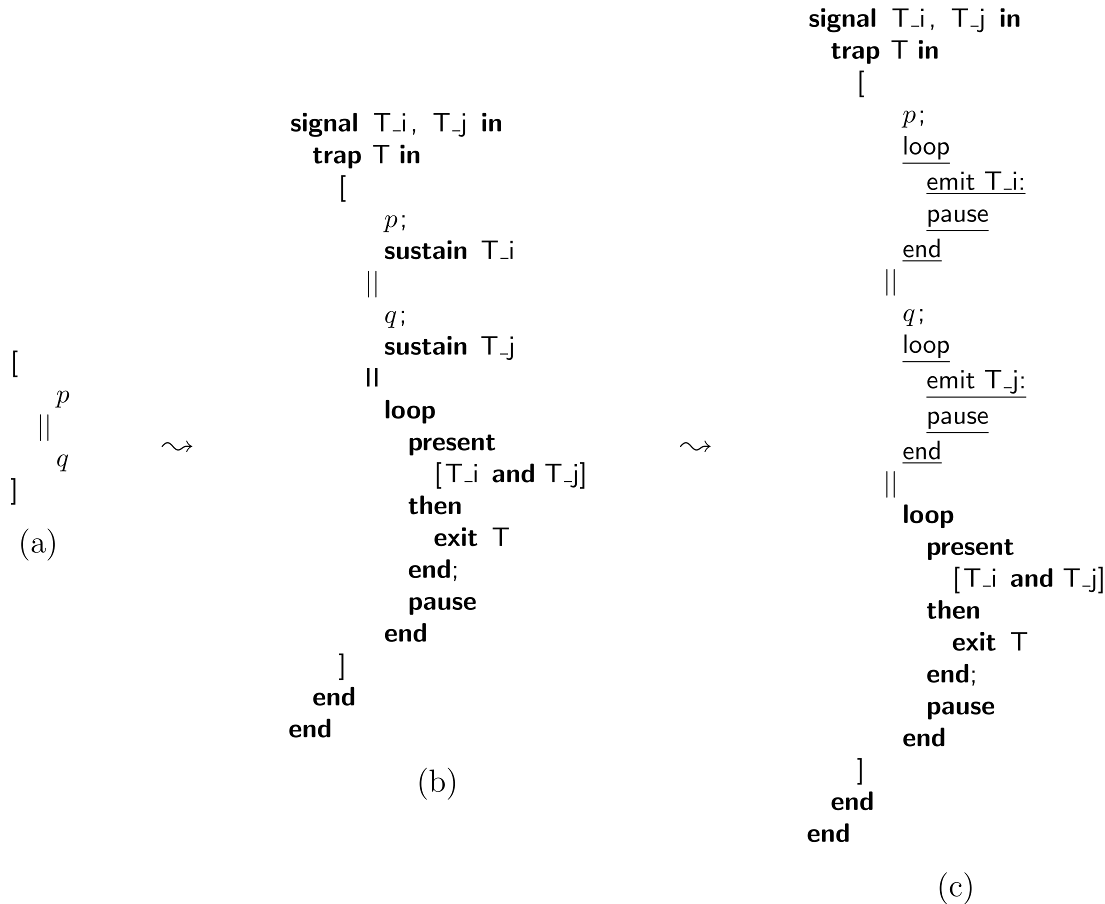


Figure 4.4: Making the termination state of parallel threads visible to signal expressions by continuous emission of state signals on terminated sub-threads: (a) original parallel block with threads p and q, (b) added termination handling by trap, (c) expansion of sustain into kernel statements.

in Section 4.3 on page 68.

Step (3b): Parallel

Another case of hidden program state is present in the termination control of parallel statements. Given a parallel block with two threads p and q:

```
  p
||
  q
```

This parallel block terminates, if both sub-blocks p and q terminate. The precise signal state of termination of the whole parallel statement is not directly accessible, because threads which 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 4.4 describes the addition of state signals to 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 state signals. If all termination 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 can be covered by simple state signals.

The complexity of these additions is proportional to the number of parallel threads in the program.

Section 4.4.3 contains an example with a cyclic dependency spanning the termination of a parallel thread.

Step (3c): Boot state

The first state signal ST_0 is emitted at program start. ST_0 corresponds to the boot register in the circuit representation of Esterel programs.

Step (3d): State signals

Each pause statement in the program is supplemented with the emission of a unique signal ST_i.

Step (4): Identification of cycles

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 cycle are of further interest. More details on the detection of cyclic dependencies are given in Section 3.4 on page 37.

If there is more than one cycle present in the program, then Steps (5) through (6) 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 (6c). Why this selection is justified is laid out in the remarks on Step (6c) on page 59.

Step (5): Cyclic input signals

This step splits each cyclic input signal σ_i into two signals σ_i and σ_i'. The signal with the original name σ_i is emitted outside the cycle or fed into the program as an input signal. All signal emissions which are part of the cycle use the new signal name σ_i'. The motivation of this step is to distinguish between emissions from inside and outside the cycle including the environment; the aim of the replacement expression (see Step (6a)) is to replace emissions inside the cycle.

In a way, this introduction of fresh signals, which are emitted exclusively in the cycle, is akin to Static Single Assignment (SSA) [13].

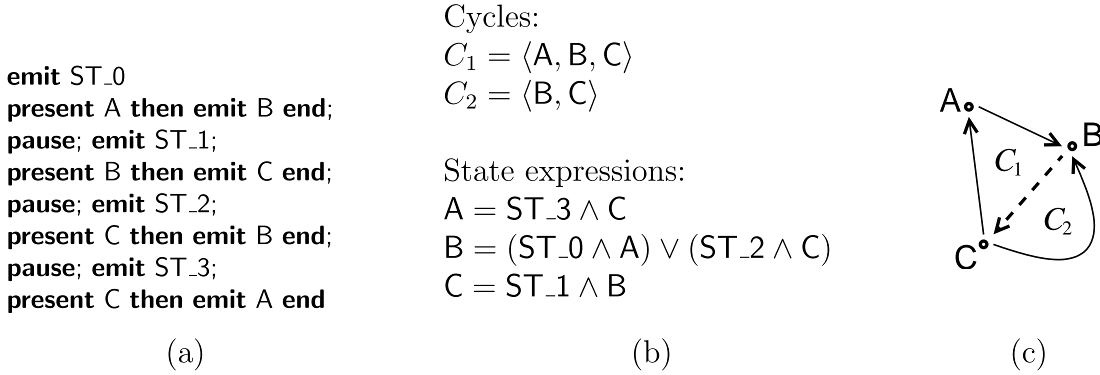


Figure 4.5: Program with potentially non-terminating iterative signal replacement: (a) Cyclic program containing two cycles with a common dependency from B to C, (b) Cycles and emission contexts of signals, (c) Graphical representation of the two cycles, the common dependency is indicated by a dashed line.

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

All tests for cyclic input signals in the original program are extended by tests for their replacement signals. Using the SSA analogy, this corresponds to a ϕ-node [13].

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

Step (6a): Replacement expression

The computation of replacement expressions E is described in detail in Section 4.2 on page 64 and following.

Step (6b): Point of cycle breaking

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

Step (6c): Iteration on the replacement expression

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 non-cycle signals are referenced in E_p^*.

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 growth in program size for one cycle is of cubic complexity.

If the original program contains multiple cycles with common signal dependencies then this iterative replacement scheme is potentially non-terminating. Formerly replaced cycle signals may be reintroduced by later expressions.

Figure 4.5 contains an Esterel program with two cycles $\vec{C}_1$ and $\vec{C}_2$. The key element here is the common dependency $\langle \mathsf{B}, \mathsf{C} \rangle$ of both cycles. If $\vec{C}_1$ is arbitrarily selected to be resolved first with A as the point to break the cycle then the following iteration is applied:

$$\mathsf{A} = \mathsf{ST_3} \wedge \mathsf{C} \tag{4.1}$$
$$\mathsf{A} = \mathsf{ST_3} \wedge \mathsf{ST_1} \wedge \mathsf{B} \tag{4.2}$$
$$\mathsf{A} = \mathsf{ST_3} \wedge \mathsf{ST_1} \wedge (\mathsf{ST_0} \wedge \mathsf{A}) \vee (\mathsf{ST_2} \wedge \mathsf{C}) \tag{4.3}$$

Cycle signal C is part of Equation (4.1). It is replaced by an an expression containing B in (4.2). The following iteration step reintroduces C into Equation (4.3). Further iteration steps will continue to oscillate between and C and B leading to an infinite size of the replacement expression for A.

To avoid this non terminating chain of replacements the algorithm requests in in Step (6c) to select a currently *smallest* cycle for transformation. In Figure 4.5 the smallest cycle is $\vec{C}_2$. Selecting B to break the cycle results in the following iteration:

$$\mathsf{B} = (\mathsf{ST_0} \wedge \mathsf{A}) \vee (\mathsf{ST_2} \wedge \mathsf{C})$$
$$\mathsf{B} = (\mathsf{ST_0} \wedge \mathsf{A}) \vee (\mathsf{ST_2} \wedge \mathsf{ST_1} \wedge \mathsf{B}) \tag{4.4}$$

Equation (4.4) is a valid result of the iteration process, because of its finite size and it contains no cycle signals of $\vec{C}_2$ besides B itself. Signal A in (4.4) must not be replaced, because it is not part of cycle $\vec{C}_2$. The following Step (6d) will yield an expression with the last occurrence of B in its replacement expression removed.

The following argues the validity of generally solving the termination problem of the iteration by selecting a smallest cycle.

Preconditions: Given constructive Esterel program P, including n cycles $\vec{C}_1, ..., \vec{C}_n$ involving signals $\sigma_i \in \Sigma$. One shortest Cycle $\vec{C}_k$ is selected with $(\forall i \in \{1, ..., n\} : |\vec{C}_k| \leq |\vec{C}_i|)$. The emission contexts of all cycle signals $\sigma_i \in \vec{C}_k$ are represented by expressions E_i $(i \in \{1, ..., |\vec{C}_k|\})$. 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 performing the iteration.

Claim: The iterative replacement of cycle signals $\sigma_i \in \vec{C}_k$ $(i \neq p)$ in expression E_p by expressions E_i terminates with finite 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 can not be part of cycle $\vec{C}_k$ (besides the first and last signal 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. **q.e.d.**

Step (6d): Making the replacement expression acyclic
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). Now E_p^* contains only non-cyclic signals.

This replacement of a cycle signal by a constant is described in Malik's work [28] on resolving cycles in cyclic circuits. The following argues the validity of this replacement.

Preconditions: Given constructive Esterel program P, including cycle involving signal σ_p, and other signals $\vec{S} = \langle s_0, ..., s_n \rangle$. A replacement function $E_p^*(\sigma_p, \vec{S})$ for signal σ_p is derived as of Step (6c) according to the circuit semantics of Esterel.

Claim: 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. Therefore the presence of σ_p can be derived without previous knowledge of the presence of σ_p for all reachable states of $\vec{S}$; in other words, σ_p is not allowed to depend on itself. E_p^* computes the presence of signal σ_p from all signals in P. Assuming E_p^* yielding different results for true and false in place of σ_p makes E_p^* depended on σ_p. This contradicts the constructiveness of P. Therefore E_p^* cannot depend on the presence value of σ_p. **q.e.d.**

Remark: The use of constructiveness here implies *strong* constructiveness as defined by Shiple *et al.* [36], *i. e.*, even local non-constructiveness with no influence on output signals are not allowed.

The remaining true or false values must be used to minimize the expression when yielding

E_p^{**} from E_p^*, because Esterel compilers do not support boolean constants in place of signal tests. Boolean constants may only be used for valued signals of boolean type, and not for the pure signals that are considered here. Hence these constants are used here only as intermediate place holders.

Signal σ_p has no influence on E_p^* for all reachable signal states and control states if the program is constructive. This does not necessarily hold for *all* states of signals in E_p^*, but only for those reached at runtime at the evaluation of E_p^*. This follows from the iterative process of signal replacement in Step (6c) which is equivalent to a symbolic version of a three-valued fix point iteration proposed by Malik [28] and Shiple *et al.* [36]. Note that if the result of the derived expression E_p^* is independent of σ_p for all *reachable* signal combinations at the control point where E_p^* is evaluated, then the program is constructive with regard to signal σ_p.

As mentioned above, this observation could be used to aid constructiveness analysis, as this would eliminate the need to perform a fixpoint iteration; nevertheless, to determine the reachable control flow and signal space remains a nontrivial problem. See Section 4.5 page 80 for details.

Step (6e): Inserting the replacement expression

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 is not part of the cycle. Therefore we have *broken* the current cycle $\vec{C}$.

Step (7): Multiple cycles

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 kind 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.

4.1.1 Cost of the Transformation Algorithm

In the previous section the discussion of each step of the transformation algorithm contained an estimate on the amount of program growth for each step. The following listing summarizes each part of the growth in program size:

- **Step (1):** The expansion of modules can reach exponential growth of code size.
- **Step (2b):** Expansion of derived statements is of constant cost with regard to the number of statements in the program.
- **Step (2c):** Elimination of reincarnation yields a quadratic growth in program size.
- **Step (3a):** Replacement of abort/suspend blocks is proportional in size to the number of pause statements inside abort and suspend statements.
- **Step (3b):** The complexity of additions for parallel statements is proportional to the number of parallel threads.
- **Step (3d):** Introduction of state signals is proportional to the number of pause statements in the program.
- **Steps (5a/5b):** The renaming of cyclic input signals is proportional to the number of input signals.
- **Step (5c):** The replacement of cyclic input signals in testing expressions is a constant factor on the number of signal test expressions in the program.
- **Step (6c):** The growth in program size by replacing a cycle signal with an expression is of cubic complexity.
- **Step (7):** The number of cycles to resolve is proportional to the number of statements in the program.

The expansion of modules with exponential cost in Step (1) must be done by Esterel compilers anyway and is therefore left out of the cost estimation for the cycle transformation alone. The same can be said on the resolving on reincarnation in Step (2c), but this depends on the kind of synthesis employed by the actual compiler. Steps (2b), (3a) to (5c), and (7) all introduce a cost proportional to different parts of the Esterel program. Therefore the overall cost of these steps can be summarized to be proportional to the size of the entire Esterel input program. In Step (6c) the actual cycle cutting takes place with a cost of cubic complexity.

Overall, a very conservative estimate results in a code size of $\mathcal{O}(n^4)$, where n is the source program size after module expansion and elimination of signal reincarnations. However, we expect the typical code size increase to be much lower. In fact, we often

```
(a)
  present I1 then
    present I2 else
      emit A
    end
  end
||
  present A then
    emit B
  end
```

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

Figure 4.6: Replacing the signal test for A by its emission context.

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 7) also demonstrate that typically the transformation results in a code size reduction.

4.2 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 (6a) 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 Circuit Translation of Esterel [3] 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 the set of signals, and Ψ the set of signal expressions):

- $\mathcal{E} : \Pi \times \Psi \rightarrow 2^{\Sigma \times \Psi} \qquad (P \times S) \mapsto \mathcal{E}(P, S)$
  This function searches for signal emissions and returns a mapping of signal names to their signal contexts at the point of their emission.

- $\mathcal{S} : \Pi \times \Psi \rightarrow \Psi \qquad (P \times S) \mapsto \mathcal{S}(P, S)$
  $\mathcal{S}$ takes the signal state context delivered by previous statements, computes the

Program $P =$	Equations	
emit A	$\mathcal{E}(P,S) = \{\langle A, S\rangle\}$ $\mathcal{S}(P,S) = S$	(4.5)
present A **then** p **else** q **end**	$\mathcal{E}(P,S) = \mathcal{E}(p, S \wedge A) \cup \mathcal{E}(q, S \wedge \overline{A})$ $\mathcal{S}(P,S) = \mathcal{S}(p, S \wedge A) \vee \mathcal{S}(q, S \wedge \overline{A})$	(4.6)
nothing	$\mathcal{E}(P,S) = \emptyset$ $\mathcal{S}(P,S) = S$	(4.7)
pause; **emit** ST_i	$\mathcal{E}(P,S) = \emptyset$ $\mathcal{S}(P,S) = \begin{cases} \mathsf{false} : S = \mathsf{false} \\ \mathsf{ST_}i : \text{otherwise} \end{cases}$	(4.8)
exit T	$\mathcal{E}(P,S) = \{\langle \mathsf{exit}\ T, S\rangle\}$ $\mathcal{S}(P,S) = \mathsf{false}$	(4.9)
trap T **in** p **end**	$\mathcal{E}(P,S) = \{\langle \sigma_i, S_j\rangle \in \mathcal{E}(p,S) \mid \sigma_i \neq \mathsf{exit}\ T\}$ $\mathcal{S}(P,S) = \mathcal{S}(p,S) \vee (\overline{\bigvee_{\langle \mathsf{exit}\ \sigma_i, S_j\rangle \in \mathcal{E}(p,S) \mid \sigma_i \neq T} S_j} \wedge \bigvee_{\langle \mathsf{exit}\ \sigma_i, S_j\rangle \in \mathcal{E}(p,S) \mid \sigma_i = T} S_j)$	(4.10)
p ; q	$\mathcal{E}(P,S) = \mathcal{E}(p,S) \cup \mathcal{E}(q, \mathcal{S}(p,S))$ $\mathcal{S}(P,S) = \mathcal{S}(q, \mathcal{S}(p,S))$	(4.11)
loop p **end**	$\mathcal{E}(P,S) = \mathcal{E}(p, S \vee \mathcal{S}(p,S))$ $\mathcal{S}(P,S) = \mathsf{false}$	(4.12)
signal A **in** p **end**	$\mathcal{E}(P,S) = \mathcal{E}(p,S)$ $\mathcal{S}(P,S) = \mathcal{S}(p,S)$	(4.13)
p || q	$\mathcal{E}(P,S) = \mathcal{E}(p,S) \cup \mathcal{E}(q,S)$ $\mathcal{S}(P,S) = \mathsf{false}$	(4.14)

Figure 4.7: 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 the state expression in the current program context.

signal state context from sub-statements, 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 Figure 4.7. 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 of $\mathcal{E}$ 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:

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

Rule (4.8) handles the pause statement with associated emission of its state signal: Function $\mathcal{E}$ does not return a context expression for the state signal, because state signals are considered free of dependencies. 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.

Trap signals are treated differently than regular signals: Function $\mathcal{E}$ in Rule (4.9) (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 (4.10) (trap) removes all references to its own trap signal to not interfere with upper trap definitions.

Function $\mathcal{S}$ in Rule (4.10) 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 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 (4.10). That problem is deferred to the Esterel parser.

Rule (4.14) (parallel) returns false for the termination context of all parallel statements, because the termination of parallel statements is assumed to be replaced by the scheme proposed in Figure 4.4. It replaces the implicit termination of parallel statements by explicit trap exception handling.

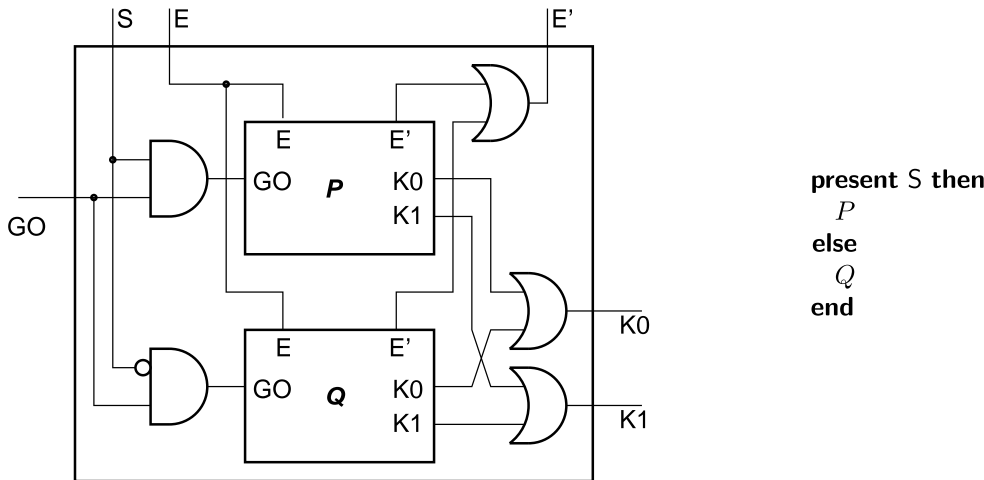


Figure 4.8: Circuit translation of the present statement as defined in the draft book on the Esterel semantics [3].

The Token Ring Arbiter, for example, is not susceptible to this problem because its parallel statements do not terminate. Section 4.4.3 contains a more problematic example program and its treatment.

Rules for suspend/abort are not given in Figure 4.7, because they are assumed to be substituted by means of other kernel statements in Step (3a) of the transformation algorithm. In Section 4.3 alternative treatments of suspend/abort are proposed.

4.2.1 Relation to the Circuit Transformation

The equations to derive replacement expressions for signals are modeled after the circuit transformation of Esterel. They follow the GO/K0 paths in the Esterel program recursively but without an explicit synthesis of a circuit representation. This is possible because the simple mapping of Esterel statements to sub-circuits preserves the control flow structure of the original program.

As an example to illustrate how the definitions of $\mathcal{E}$ and $\mathcal{S}$ correspond to circuits, consider the translation of the present statement. Its circuit is repeated in Figure 4.8 from the Esterel introduction (page 27). On entering a present block the $\mathcal{S}$ expression represents the GO signal. That signal is combined with the present condition and new GO signals are derived for the connection of the sub blocks P and Q. In Rule (4.6) the same extension to the $\mathcal{S}$ expression is performed to evaluate p and q respectively.

The termination signal K0 of the present block is generated by the OR combination of the termination signals from P and Q. This is equivalent to the handling of $\mathcal{S}$ in Rule (4.6). The higher order termination signal K1 (originating at pause statements) is mapped to $\mathcal{S}$ by combination with individual state signals ST_i for each pause statement in Rule (4.8).

All higher order termination signals K2 and up are connected to exit statements. These

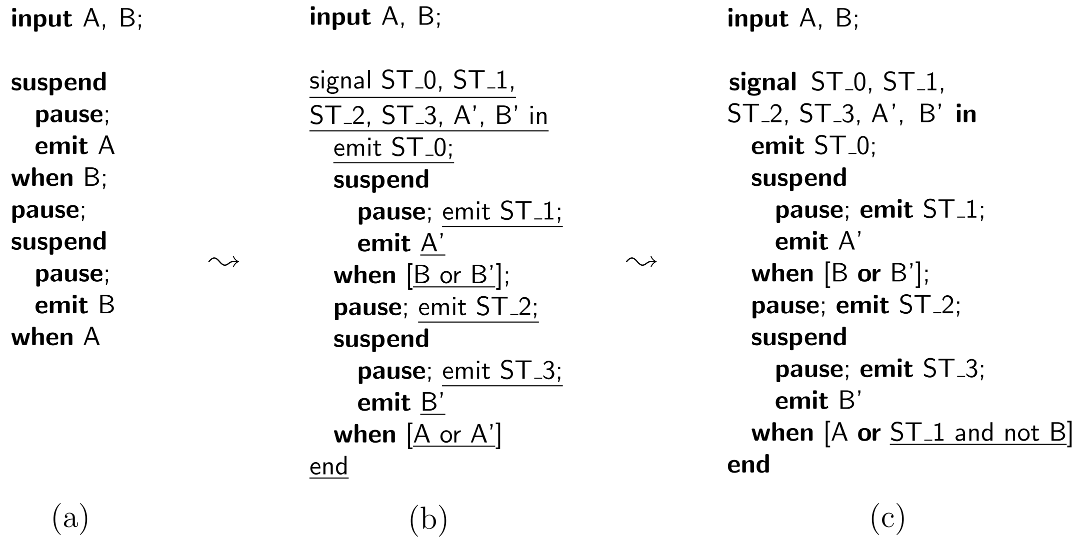


Figure 4.9: Failed resolving of a cyclic dependency involving two suspend statements: (a) original program, (b) preprocessing by introduction of state signals and signal renaming, (c) replacement of tests for A' by expression involving ST_1, but ST_1 is guarded by cycle signal B'.

are mapped as additional signal emissions to $\mathcal{E}$. Details are given at the explanations for Rules (4.9) and (4.10).

The tight relationship to the circuit transformation of Esterel ensures the preservation of the semantics of the original program when cycle signals are replaced by expressions.

4.3 Extending Esterel for an Alternative Solution to Handle suspend/abort

The transformation algorithm states in Step (2b) (page 53) as a prerequisite the replacement of all non-kernel statements by their definitions in kernel statements. Additionally the kernel statement suspend is replaced by means of other kernel statements (see page 55 for details). This replacement is needed, because the presented algorithm does not fully support splitting cycles across suspend statements. The reason for the failure lies in the use of state signals which are introduced in Step 3 of the transformation algorithm. One underlying assumption on replacing cycle signals with expressions is the fact that state signals are always independent of any signal dependencies and therefore do not introduce new cyclic dependencies. That assumption is invalid in the context of a suspend or abort statement. State signals which are emitted inside an abort or suspend block are subject to a dependency to the guard of the abort or suspend block.

Figure 4.9(a) contains a program fragment with a (constructive) cyclic dependency, which is not resolvable by replacement of cycle signals alone. Application of the trans-

formation algorithm will produce Figure 4.9(b) as an intermediate step: State signals are introduced and input signals part of the cycle are renamed. The new cycle signals A' and B' are emitted in the following state contexts:

$$\mathsf{A'} := \mathsf{ST_1} \wedge \overline{\mathsf{B} \vee \mathsf{B'}}$$

$$\mathsf{B'} := \mathsf{ST_3} \wedge \overline{\mathsf{A} \vee \mathsf{A'}}$$

Substituting B' in the expression for A' yields:

$$\mathsf{A'} := \mathsf{ST_1} \wedge \overline{\mathsf{B} \vee \mathsf{ST_3} \wedge \overline{\mathsf{A} \vee \mathsf{A'}}}$$

Replacing the remaining A' by true and some simplifications results in:

$$\mathsf{A'} := \mathsf{ST_1} \wedge \overline{\mathsf{B}}$$

The resulting program is listed in Figure 4.9(c). That program is *not* acyclic because the state signal ST_1 is introduced into the suspend guard. That signal is emitted inside the body of the other suspend statement and therefore dependent on the cycle signals in its guard.

An efficient solution for this problem is not obvious. Selecting another signal of the cycle for cycle breaking does not work, both cycle signals are symmetrically dependent between two suspend blocks.

The solution proposed in Section 4.1 on page 55 (Step (3a)) of substituting suspend into other kernel statements is viable, but possibly expensive in code size.

Another possibility is the extension of the Esterel pause command. Instead of adding an emit statement to all pause statements, the state signal is added to the pause statement itself:

pause ST_i

The modified pause statement will emit the added state signal in each instant it is activated. The purpose of this extension to the pause statement is to emit state signals with no regard for suspension and abortion. Therefore no additional cyclic dependencies can be introduced by state signals inside abort/suspend statements.

Figure 4.10 contains the extended equations to determine replacement expressions for signals under influence of abort/suspend statements. Functions $\mathcal{E}$ and $\mathcal{S}$ get an additional third parameter W (*watcher*) containing the abort/suspend expression in the current program context. That parameter is initialized with false at start of the program analysis:

$$E = \mathcal{E}(p, \mathsf{ST_0}, \mathsf{false})$$

When entering the body of a abort/suspend statement (Rules 4.17 and 4.18), the new abort/suspend condition is disjuncted with the current watcher expression.

The watcher expression is added to the signal context expression at pause statements in Rule (4.16). It is negated because if the watcher expression yields true, then the signal emissions are suppressed in that instant.

Rule (4.15) is not changed compared to Figure 4.7 besides the additional parameter W

for functions $\mathcal{E}$ and $\mathcal{S}$. One might consider to use the watcher expression W directly for signal emissions, *e. g.*, like this:

$$\mathcal{E}(!A, S, W) = \{\langle A, S \wedge \overline{W} \rangle\}$$

However the flaw in this attempt lies in the delayed nature of the abort/suspend statements, the abort/suspend condition is per definition not evaluated in the instant of entry into the abort/suspend block. When at least the first pause statement in the abort/suspend block had been encountered, then the watcher condition is evaluated. Therefore the right thing to do is to add the watcher condition at the pause statements to the state signal (Rule (4.16)). This ensures the correct handling of the exceptional "first instant". The additional case distinction in function $\mathcal{S}$ of Rule (4.16) is related to skipping of dead code.

Function $\mathcal{E}$ of Rule (4.16) returns the state signal as a tuple connected to an empty expression. The purpose of this addition is to collect all state signals contained in an abort block as possible points of abortions. That empty expression is just a placeholder which keeps the data format returned by $\mathcal{E}$ consistent.

Function $\mathcal{S}$ in Rule (4.18) computes the state context at termination of the abort block: The expression $\mathcal{S}(p, S, W \vee A)$ returns the regular termination of the abort body p when no abortion takes place. The collected state signals are used in the additional expression

$$\bigvee_{\langle \mathsf{ST}_i, \emptyset \rangle \in \mathcal{E}(p,S,W \vee A)} \mathsf{ST}_i \wedge A \wedge \overline{W}$$

It covers all possible control flows from pause statements inside the abort block to the end of the abort statement when the abortion condition A holds. This is done by extracting all state signals off the set returned by $\mathcal{E}$ applied to the abort body p. Each state signal is connected to the abortion condition A and the negation of the watcher expression from upper abort levels. This correctly handles abort hierarchies.

The handling of suspend in Rule (4.17) is much simpler, functions $\mathcal{E}$ and $\mathcal{S}$ just return the results of their sub-blocks.

4.4 Example Transformations

In this section the transformation algorithm from Section 4.1 is illustrated by applying it to some examples.

4.4.1 Transforming PAUSE_CYC

The algorithm is applied to the example PAUSE_CYC in Figure 3.2(a) on page 32, which is transformed into the acyclic program PAUSE_ACYC in Figure 3.2(c). The transformation of the program DRIVER_CYC in Figure 3.4(a), page 33, into DRIVER_ACYC in Figure 3.4(b) is similar.

Step (1): PAUSE_CYC is cyclic but nevertheless constructive, because a pause statement

$P =$ **emit** A	$\mathcal{E}(P,S,W) = \{\langle A, S\rangle\}$ $\mathcal{S}(P,S,W) = S$	(4.15)
...		
pause $\mathsf{ST_}i$	$\mathcal{E}(P,S,W) = \{\langle \mathsf{ST_}i, \emptyset\rangle\}$ $\mathcal{S}(P,S,W) = \begin{cases} \mathsf{false} & : S = \mathsf{false} \\ \mathsf{ST_}i \wedge \overline{W} & : \text{otherwise} \end{cases}$	(4.16)
...		
suspend p **when** A	$\mathcal{E}(P,S,W) = \mathcal{E}(p,S,W \vee A)$ $\mathcal{S}(P,S,W) = \mathcal{S}(p,S,W \vee A)$	(4.17)
abort p **when** A	$\mathcal{E}(P,S,W) = \mathcal{E}(p,S,W \vee A)$ $\mathcal{S}(P,S,W) = \mathcal{S}(p,S,W \vee A) \vee \bigvee_{\langle \mathsf{ST_}i,\emptyset\rangle \in \mathcal{E}(p,S,W\vee A)} \mathsf{ST_}i \wedge A \wedge \overline{W}$	(4.18)

Figure 4.10: Equations to determine replacement expressions for signals in a suspend and abort context. It requires an extension of Esterel with a pause statement that emits an associated state signal without suppression by suspend/abort conditions.

separates the execution of both parts of the cycle.

Steps (2a) to (2c) do not apply to PAUSE_CYC.

Steps (3) and (5): To prepare the removal of the cycle, we first transform PAUSE_CYC into the equivalent program PAUSE_PREP, shown in Figure 3.2(b). It differs from PAUSE_CYC in the introduction of state signals ST_0 to ST_2 and in that the signals carrying the cycle (A and B) have been replaced by fresh signals A_ and B_, which are only emitted within the cycle. All tests for A and B in the original program are replaced by tests for [A or A_] and [B or B_], respectively.

Step (4): PAUSE_CYC contains one cycle: $\vec{C} = \langle \mathsf{A}, \mathsf{B} \rangle$.

Step (6a): The computation of replacement expressions for A_ and B_ according to Section 4.2 results in:

$$\mathsf{A_} = \mathsf{ST_1} \wedge (\mathsf{B} \vee \mathsf{B_}) \quad (4.19)$$

$$\mathsf{B_} = \mathsf{ST_0} \wedge (\mathsf{A} \vee \mathsf{A_}) \quad (4.20)$$

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 (6b): In $\mathsf{PAUSE_PREP}$, we arbitrarily select $\mathsf{A_}$ as the signal to break the cycle.

Step (6c): To replace $\mathsf{B_}$ in Equation (4.19), substituting (4.20) into (4.19) results in:

$$\mathsf{A_} = \mathsf{ST_1} \wedge (\mathsf{B} \vee (\mathsf{ST_0} \wedge (\mathsf{A} \vee \mathsf{A_}))). \tag{4.21}$$

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

Step (6d): Replacing the self-reference of signal $\mathsf{A_}$ on the right hand side of (4.21) by false (absent) yields:

$$\mathsf{A_} = \mathsf{ST_1} \wedge (\mathsf{B} \vee (\mathsf{ST_0} \wedge \mathsf{A})). \tag{4.22}$$

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

$$\mathsf{A_} = \mathsf{ST_1} \wedge (\mathsf{B} \vee \mathsf{ST_0}). \tag{4.23}$$

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

Step (6e): Finally we are ready to break the cycle in $\mathsf{PAUSE_PREP}$. For that, we have to replace the signal selected in Step (6b) — in the cycle — by one of the expressions computed in Step (6d), which do not use any of the cycle signals, without changing the meaning of the program.

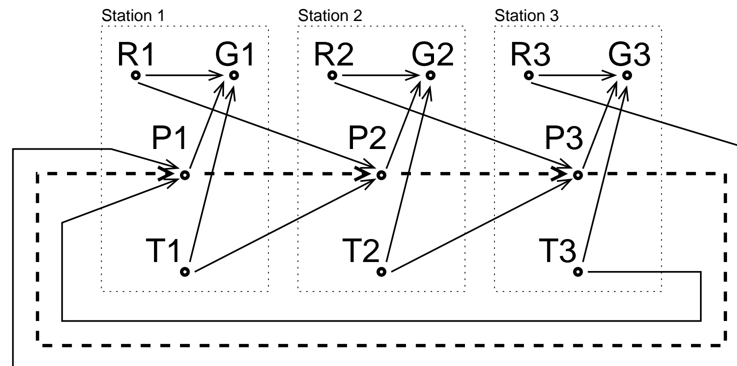
Substituting (4.23), the simpler of these expressions, for $\mathsf{A_}$ in $\mathsf{PAUSE_PREP}$ yields the now acyclic program $\mathsf{PAUSE_ACYC}$ shown in Figure 3.2(c).

4.4.2 Transforming the Token Ring Arbiter

Searching for signal dependencies in the program $\mathsf{TR3_CYC}$ from Figure 3.5 page 35 according to the algorithm presented in Chapter 3 yields the following set:

$$D = \left\{ \begin{array}{l} \langle \mathsf{R1},\mathsf{P2}\rangle, \langle \mathsf{R1},\mathsf{G1}\rangle, \langle \mathsf{R2},\mathsf{P3}\rangle, \langle \mathsf{R2},\mathsf{G2}\rangle, \langle \mathsf{R3},\mathsf{P1}\rangle, \langle \mathsf{R3},\mathsf{G3}\rangle, \\ \langle \mathsf{P1},\mathsf{P2}\rangle, \langle \mathsf{P1},\mathsf{G1}\rangle, \langle \mathsf{P2},\mathsf{P3}\rangle, \langle \mathsf{P2},\mathsf{G2}\rangle, \langle \mathsf{P3},\mathsf{P1}\rangle, \langle \mathsf{P3},\mathsf{G3}\rangle, \\ \langle \mathsf{T1},\mathsf{P2}\rangle, \langle \mathsf{T1},\mathsf{G1}\rangle, \langle \mathsf{T2},\mathsf{P3}\rangle, \langle \mathsf{T2},\mathsf{G2}\rangle, \langle \mathsf{T3},\mathsf{P1}\rangle, \langle \mathsf{T3},\mathsf{G3}\rangle \end{array} \right\}$$

These dependencies can be visualized by the following graph:



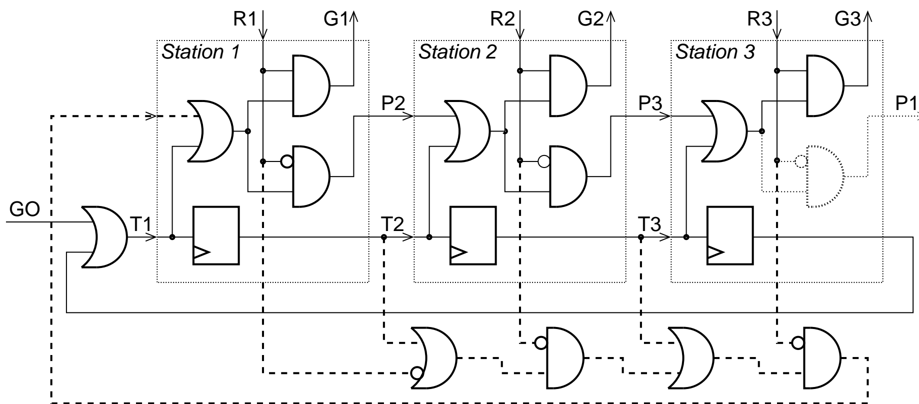
Searching in D for cyclic dependencies delivers the following cycle of length three:

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

(a)



(b)

Figure 4.11: Non-cyclic Token Ring Arbiter: (a) Transformation of the cyclic original of Figure 3.5(a) page 35, (b) Simplified circuit representation, dashed lines indicate the additions to make the original circuit acyclic. The emission of P1 is replaced by $(T3 \vee (T2 \vee \overline{R1}) \wedge \overline{R2}) \wedge \overline{R3}$.

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

The cycle is indicated in the graph by dashed lines.

Renaming of cycle signals is not needed here, because all three cycle signals are internally defined signals and not input signals.

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

$$\begin{aligned} \mathsf{P1} &= (\mathsf{ST_0} \vee \mathsf{ST_7}) \wedge (\mathsf{T3} \vee \mathsf{P3}) \wedge \overline{\mathsf{R3}} \\ \mathsf{P2} &= (\mathsf{ST_0} \vee \mathsf{ST_1}) \wedge (\mathsf{T1} \vee \mathsf{P1}) \wedge \overline{\mathsf{R1}} \\ \mathsf{P3} &= (\mathsf{ST_0} \vee \mathsf{ST_4}) \wedge (\mathsf{T2} \vee \mathsf{P2}) \wedge \overline{\mathsf{R2}} \end{aligned}$$

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

$$\begin{aligned} \mathsf{P1} &= (\mathsf{ST_0} \vee \mathsf{ST_7}) \wedge (\mathsf{T3} \vee \mathsf{P3}) \wedge \overline{\mathsf{R3}} \\ \mathsf{P1} &= (\mathsf{ST_0} \vee \mathsf{ST_7}) \wedge (\mathsf{T3} \vee ((\mathsf{ST_0} \vee \mathsf{ST_4}) \wedge (\mathsf{T2} \vee \mathsf{P2}) \wedge \overline{\mathsf{R2}})) \wedge \overline{\mathsf{R3}} \\ \mathsf{P1} &= (\mathsf{ST_0} \vee \mathsf{ST_7}) \wedge (\mathsf{T3} \vee ((\mathsf{ST_0} \vee \mathsf{ST_4}) \wedge (\mathsf{T2} \vee ((\mathsf{ST_0} \vee \mathsf{ST_1}) \wedge \\ &\qquad (\mathsf{T1} \vee \mathsf{P1}) \wedge \overline{\mathsf{R1}})) \wedge \overline{\mathsf{R2}})) \wedge \overline{\mathsf{R3}} \end{aligned} \tag{4.24}$$

Equation (4.24) 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 false yields:

$$\mathsf{P1} = (\mathsf{ST_0} \vee \mathsf{ST_7}) \wedge (\mathsf{T3} \vee (\mathsf{ST_0} \vee \mathsf{ST_4}) \wedge (\mathsf{T2} \vee (\mathsf{ST_0} \vee \mathsf{ST_1}) \wedge \mathsf{T1} \wedge \overline{\mathsf{R1}}) \wedge \overline{\mathsf{R2}}) \wedge \overline{\mathsf{R3}}. \tag{4.25}$$

Setting P1 to true yields:

$$\mathsf{P1} = (\mathsf{ST_0} \vee \mathsf{ST_7}) \wedge (\mathsf{T3} \vee (\mathsf{ST_0} \vee \mathsf{ST_4}) \wedge (\mathsf{T2} \vee (\mathsf{ST_0} \vee \mathsf{ST_1}) \wedge \overline{\mathsf{R1}}) \wedge \overline{\mathsf{R2}}) \wedge \overline{\mathsf{R3}}. \tag{4.26}$$

The shorter expression (4.26) is applied when transforming TR3_CYC into the acyclic program TR3_ACYC shown in Figure 4.11(a) on page 73. The other transformation steps are fairly straightforward.

The replacement expression is fairly complex, but close inspection yields an optimization. The expression (ST_0 ∨ ST_7) is contained in (4.26): The state signal ST_0 is emitted in the first instant and ST_7 is emitted in all instants but the first one. In a disjunction they will always return true. Therefore the expression can be replaced statically by true. The same holds for (ST_0 ∨ ST_4) and (ST_0 ∨ ST_1).

With this optimization (4.26) can be reduced to:

$$\mathsf{P1} = (\mathsf{T3} \vee (\mathsf{T2} \vee \overline{\mathsf{R1}}) \wedge \overline{\mathsf{R2}}) \wedge \overline{\mathsf{R3}}. \tag{4.27}$$

Further optimization opportunities are discussed in Chapter 6.

This much simpler equation is used to break the cyclic dependency in the circuit representation of the Token Ring Arbiter in Figure 4.11(b). Dashed lines indicate the new connections. The emission of P1 is not needed anymore, the associated parts of the circuit are marked with dotted lines.

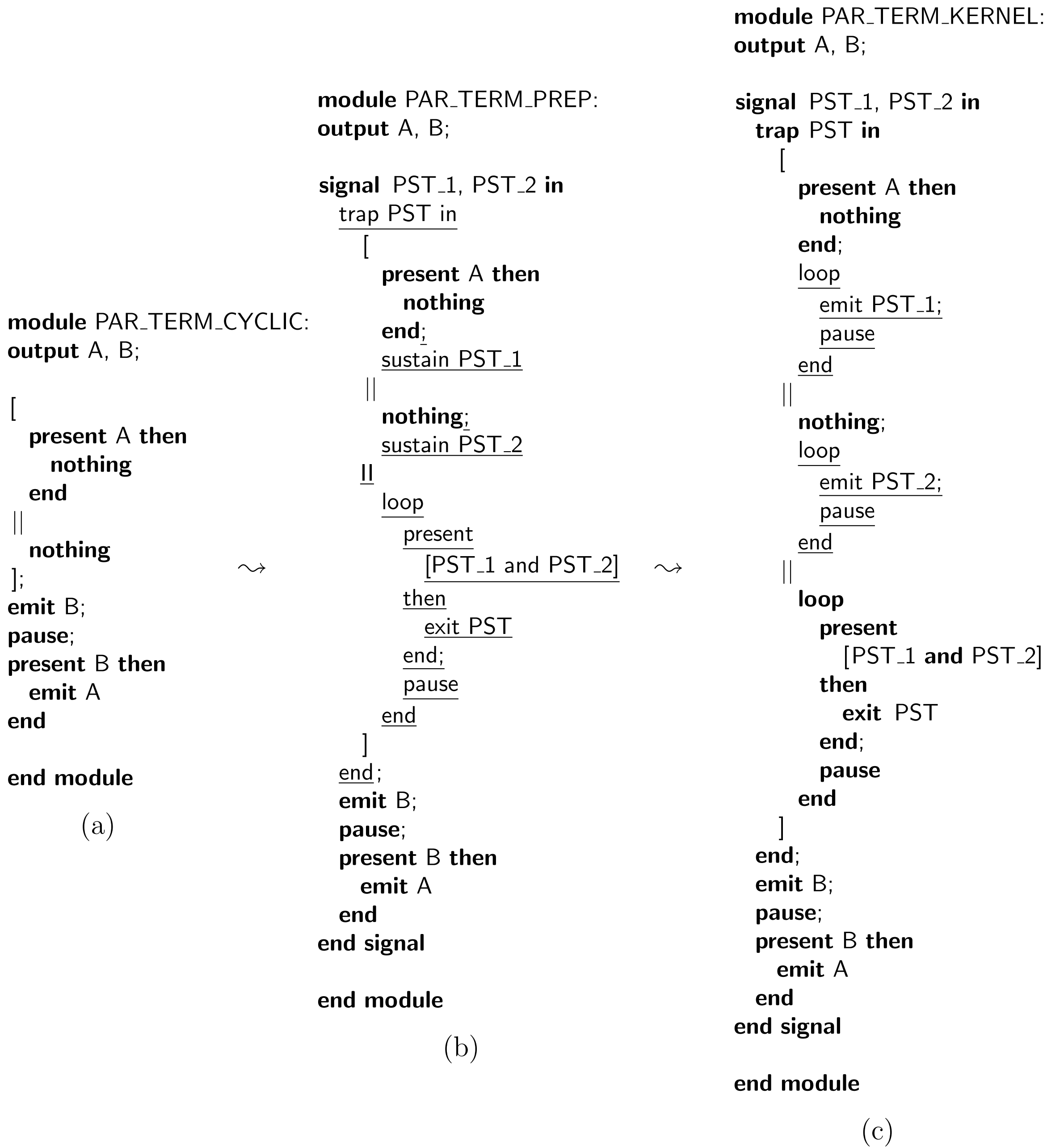


Figure 4.12: Treatment of cyclic dependencies crossing parallel termination: (a) Program with cyclic dependency across the termination of a parallel operator, (b) addition of trap/sustain/exit construct to correctly catch parallel termination, (c) expansion to kernel statements. See Figure 4.13 for resolving the cycle.

```
module PAR_TERM_ACYC:
output A, B;

signal PST_1, PST_2 in
signal ST_0, ST_1, ST_2,
       ST_3, ST_4 in
  emit ST_0;
  signal PST_1, PST_2 in
    trap PST in
      [
        present
          [ST_4 and (ST_0 or ST_3)
           and (ST_0 or ST_1) and PST_2]
        then
          nothing
        end present;
        loop
          emit PST_1;
          pause;
          emit ST_1
        end loop
      ||
        nothing;
        loop
          emit PST_2;
          pause;
          emit ST_2
        end loop
      ||
        loop
          present
            [PST_1 and PST_2]
          then
            exit PST
          end;
          pause;
          emit ST_3
        end loop
      ]
    end trap;
    emit B;
    pause;
    emit ST_4;
    present B then
      emit A
    end present
  end signal
end signal

end module
```

(a)

⇝

```
module PAR_TERM_OPT:
output A, B;

signal PST_1, PST_2 in
  trap PST in
    [
      present false then
        nothing
      end present;
      loop
        emit PST_1;
        pause
      end loop
    ||
      nothing;
      loop
        emit PST_2;
        pause
      end loop
    ||
      loop
        present
          [PST_1 and PST_2]
        then
          exit PST
        end;
        pause
      end loop
    ]
  end trap;
  emit B;
  pause;
  emit ST_4;
  present B then
    emit A
  end present
end signal

end module
```

(b)

Figure 4.13: Continuation of Figure 4.12: (a) Resolved cycle by application of the transformation algorithm, (b) optimized version.

4.4.3 Transforming Cycles Over Parallel Termination

Figure 4.12(a) contains an example program with a cycle over the termination of a parallel statement. The program consists of a parallel statement block with two threads. In the first thread signal A is tested and the thread terminates instantly. The second thread does nothing and terminates instantly, too. Sequentially following the parallel statement block is the emission of signal B. This constitutes a signal dependency from A to B over the termination of a parallel statement block. To close the cycle, a simple emission of A guarded by B is added. A pause statement makes the program constructive by separating both signal tests and emissions into separate instants.

The key in resolving this cyclic dependency lies in deriving a context expression for the emission of signal B. This involves finding a signal state expression describing the termination of the parallel statement. This may be doable intuitively in this fairly simple example by stating

$$\mathsf{B} = \mathsf{ST_0}$$

but the general case needs more effort.

In Figure 4.12(b) the additions performed by Step (3b) of the algorithm (see also Figure 4.4 on page 57) are applied. To each thread a sustain command is added at termination, and the combination of all termination signals leads to activation of the trap around the parallel statement.

Figure 4.12(c) contains the expansion of the sustain commands into kernel statements (Step (2b)) as indicated in Figure 4.3 on page 56. Now the program is ready to be applied to the transformation algorithm.

The signal dependencies in this program are:

$$D = \left\{ \begin{array}{l} \langle \mathsf{A}, \mathsf{PST_1} \rangle, \\ \langle \mathsf{B}, \mathsf{A} \rangle, \\ \langle \mathsf{PST_1}, \mathsf{B} \rangle, \langle \mathsf{PST_1}, \mathsf{PST} \rangle, \\ \langle \mathsf{PST_2}, \mathsf{B} \rangle, \langle \mathsf{PST_2}, \mathsf{PST} \rangle, \langle \mathsf{PST_2}, \mathsf{PST_1} \rangle \end{array} \right\}$$

This results in the following cyclic dependency:

$$C = \langle \mathsf{A}, \mathsf{PST_1}, \mathsf{B} \rangle$$

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

$$\begin{array}{lcl} \mathsf{A} & = & \mathsf{ST_4} \wedge \mathsf{B} \\ \mathsf{B} & = & \mathsf{false} \vee (\mathsf{ST_0} \vee \mathsf{ST_3}) \wedge \mathsf{PST_1} \wedge \mathsf{PST_2} \\ \mathsf{PST_1} & = & \mathsf{ST_0} \vee \mathsf{ST_1} \end{array}$$

The signal selected to cut the cycle is A. Cycle signals B and PST_1 are substituted in the equation for A:

$$\begin{array}{lcl} \mathsf{A} & = & \mathsf{ST_4} \wedge (\mathsf{false} \vee (\mathsf{ST_0} \vee \mathsf{ST_3}) \wedge \mathsf{PST_1} \wedge \mathsf{PST_2}) \\ & = & \mathsf{ST_4} \wedge (\mathsf{false} \vee (\mathsf{ST_0} \vee \mathsf{ST_3}) \wedge (\mathsf{ST_0} \vee \mathsf{ST_1}) \wedge \mathsf{PST_2}) \end{array}$$

The resulting expression does not contain the cycle signal A, therefore a simple removal of boolean constants suffices:

$$\mathsf{A} \;=\; \mathsf{ST_4} \wedge (\mathsf{ST_0} \vee \mathsf{ST_3}) \wedge (\mathsf{ST_0} \vee \mathsf{ST_1}) \wedge \mathsf{PST_2}$$

This expression is used to replace the test for signal A in the first thread of the parallel statement. The resulting acyclic program is listed in Figure 4.12(d).

Further inspection of the tranformed program in Figure 4.13(a) reveals that the expressions ($\mathsf{ST_0} \vee \mathsf{ST_3}$) and ($\mathsf{ST_0} \vee \mathsf{ST_1}$) yield both always true at their point of evaluation. But more impact has ST_4, which is absent at the same point inside the parallel block. This reduces the entire replacement expression to false. It is a correct reflection of the fact that the test and emission of A take place in different instants in the original program (Figure 4.12(a)).

The optimized program is listed in Figure 4.13(b). It must be noted that the "present false" expression is invalid. Instead, the entire present statement must be removed.

4.4.4 Multiple pause Statements

Figure 4.14 gives another program with a cycle involving signals A and B. However, the cycle is only present at certain instants; the guarded emit of B takes place every 3rd instant, whereas the guarded emit of A takes place every 5th instant, hence the cycle is active only every 15th instant. Here the transformation into an acyclic version requires the emission of the signal ST_2, which indicates the evaluation of the guard dependency with sink B. As an optimization, due to the invariant "T1 **or** T2 = **true**," which is ensured by the third parallel thread, it suffices to just replace the guard in the guard dependency involving A in the transformed program by "A **or** ST_2".

4.4.5 Suspend

So far we presented only cycles with a present test as a guard for an emit statement. Another way to influence the execution of emit is the suspend statement. A complication with suspend is that, unlike with present, one cannot easily generate a signal that is emitted unconditionally whenever the guard of a suspend is evaluated. The transformation algorithm therefore first transforms the suspend statements into equivalent present/trap statements, in Step (3a).

As an example, consider the program SUSP_CYC in Figure 4.15(a). The program contains a cyclic dependency on the signals A and B, the emission of each signal is inhibited by the presence of the other signal.

Applying Step (3a) results in the preprocessed programs SUSP_PREP and SUSP_KERNEL in Figure 4.15(b)/(c). Step (3d) adds state signals to SUSP_STATE shown in Figure 4.16(a). The result, after applying the whole transformation algorithm, is SUSP_ACYC in Figure 4.16(b).

```
module PAUSES_CYC:
input A, B;
output C, D;

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

(a)

↝

```
module PAUSES_PREP:
input A, B;
output C, D;

signal ST_1, ST_2 in
  signal T1, T2 in
    loop
      pause; pause;
      pause; emit ST_1;
      present [A or A_ or T1]
      then
        emit B_;
        emit C
      end
    end
  ||
    loop
      pause; pause;
      pause; pause;
      pause; emit ST_2;
      present [B or B_ or T2]
      then
        emit A_;
        emit D
      end
    end
  ||
    loop
      emit T1;
      pause;
      emit T2;
      pause
    end
end
end module
```

(b)

↝

```
module PAUSES_ACYC:
input A, B;
output C, D;

  signal ST_1, ST_2 in
  signal T1, T2 in
    loop
      pause; pause;
      pause; emit ST_1;
      present [A or
  ST_2 and (B or ST_1 or T2)
                    or T1] then
      then
        emit B_;
        emit C
      end
    end
  ||
    loop
      pause; pause;
      pause; pause;
      pause; emit ST_2;
      present [B or B_ or T2]
      then
        emit A_;
        emit D
      end
    end
  ||
    loop
      emit T1;
      pause;
      emit T2;
      pause
    end
end
end module
```

(c)

Figure 4.14: Example requiring state signals which cannot be eliminated: (a) Cyclic dependency between signals A and B, (b) introduction of state signals and signal separation (unused state signals are left out for brevity), (c) acyclic program obtained by replacing A_ with an expression.

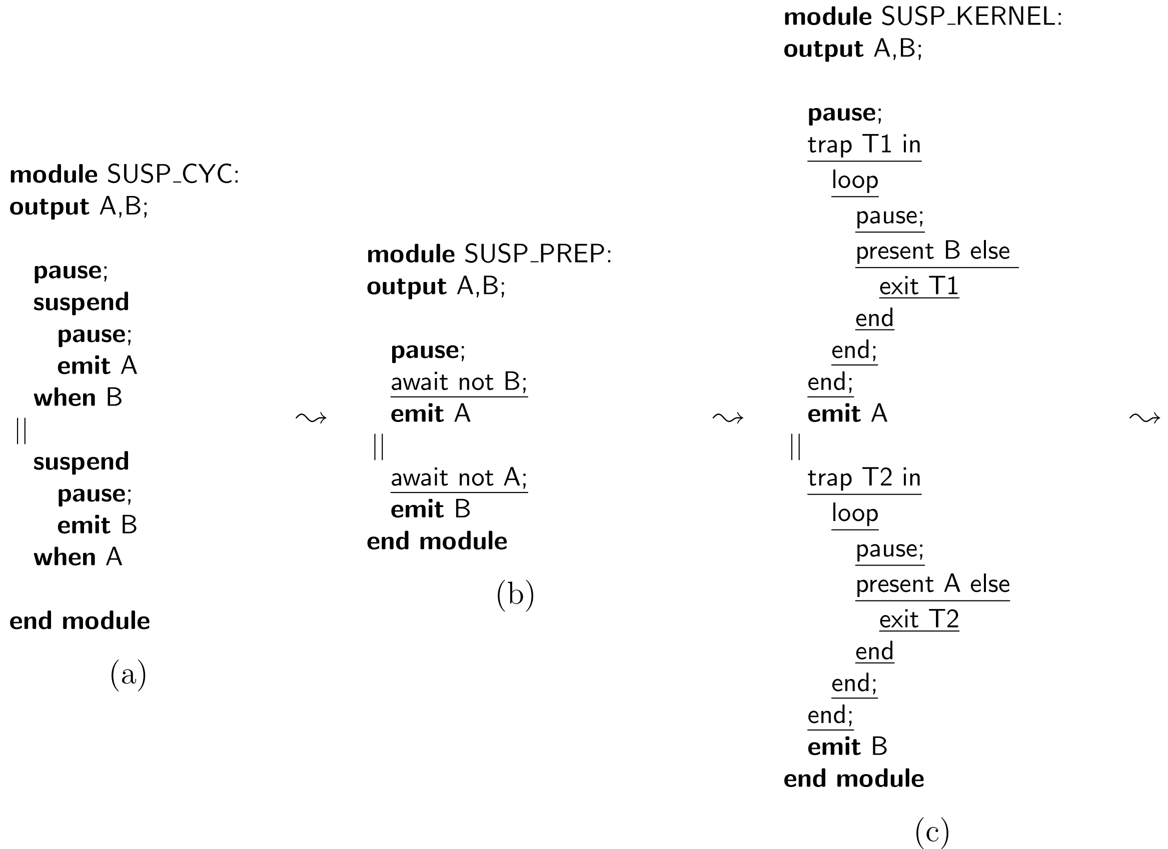


Figure 4.15: Simple cyclic program with suspend: (a) Original program with cycle between A and B, (b) replacement of pause inside suspend by await, (c) expansion of await into kernel statements. See Figure 4.16 for resolving the cycle.

SUSP_ACYC can be optimized further by noting that ST_3 is always present at its point of evaluation. This results in SUSP_OPT in Figure 4.16(c) after removing the now unneeded state signals.

4.5 Proposals for Constructiveness Analysis Using Replacement Expressions

The transformation algorithm as described until now needs the constructiveness of input programs as a precondition. The algorithm exploits the constructiveness while replacing self referencing signal names in replacement expressions (see the remarks on transformation Step (6d) on page 62).

Algorithms testing the constructiveness of cyclic dependencies in Esterel programs are usually not working on the original structure of the Esterel program but on some abstracted intermediate representation or a circuit synthesized from the Esterel program.

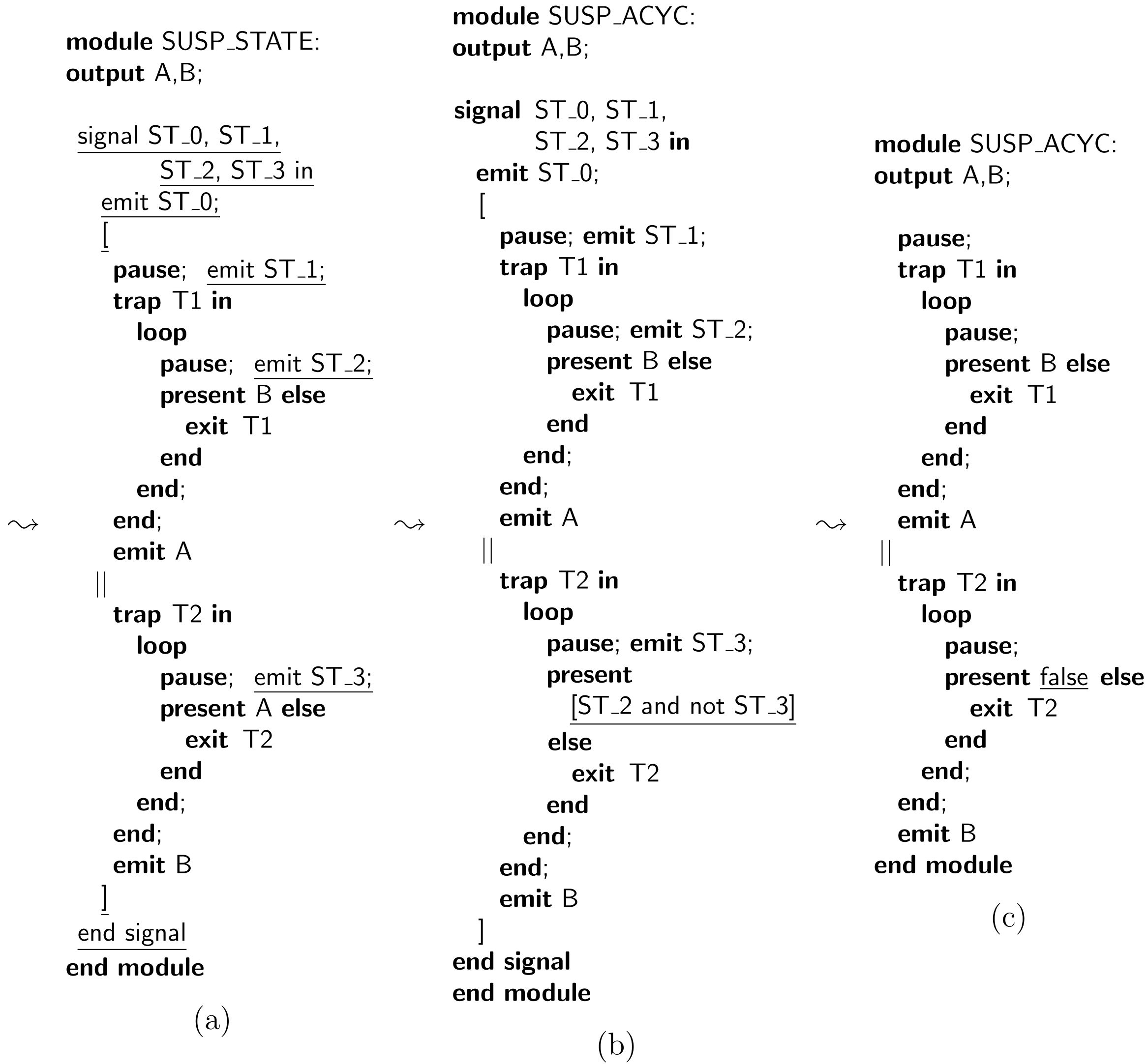


Figure 4.16: Continuation of Figure 4.15: (a) Introduction of state signals, (b) resolved cycle by application of the transformation algorithm, (c) optimized version.

The following two sections outline methods to implement constructiveness analysis directly on the Esterel level by utilizing the replacement expressions originally used to resolve cyclic dependencies.

Section 4.5.1 proposes changes for the classic constructiveness analysis based on three-valued fixpoint iteration and program state exploration on circuits. A summary of that section is given in Figure 4.17 on page 84 .

Section 4.5.2 modifies the algorithm on temporal induction proposed by Claessen [11].

4.5.1 Substitution of Fixpoint Iteration

Common algorithms for tests of constructiveness of Esterel programs are based on a three-valued fix point iteration to simulate the execution of the Esterel program under test. If all signal states converge to either "absent" or "present" in all reachable program states, then the program is considered constructive. Typical representations of these algorithms working on the circuit level are proposed by Malik [28] and Shiple *et al.* [36]. An additional effort of these procedures lies in the book keeping of already tested program states, the derivation of other states from reached ones, and the comparison against already checked states.

The modification proposed here changes the handling of the three-valued fixpoint iteration. Instead of performing the fixpoint iteration dynamically on all reached states of the input, the fixpoint iteration is statically solved in a preprocessing step. In that step equations are derived for each signal that is part of a cyclic dependency. Those equations describe for a given state context, if the emission of a signal depends on the signal itself. This would indicate the non-constructiveness of the program. The main benefit of the modification presented here lies in the much simpler evaluation of a binary expression instead of a fixpoint iteration on three-valued signals.

The expressions replacing the fixpoint iteration can be deduced from the procedure described in Section 4.2 on the computation of replacement expressions: For a cycle signal σ_i the expression E_i describes the signal context in which the signal is emitted. E_i generally involves cycle signals including σ_i and other signals of the program.

Now E_i is transformed into E_i^* by iteratively replacing all other cycle signals σ_j (but skipping σ_i) by their respective expressions E_j. The resulting expression E_i^* is a function of the cycle signal σ_i and other signals $\tilde{\sigma}$, which are not part of the cycle.

The function $E_i^*(\sigma_i, \tilde{\sigma})$ can now be used to test the constructiveness of the emission of σ_i in the context of some signal state $\tilde{\sigma}$. For that purpose each occurrence of σ_i in E_i^* is replaced by true and false yielding two new functions:

$$E_i^+(\tilde{\sigma}) = E_i^*(\mathsf{true}, \tilde{\sigma})$$
$$E_i^-(\tilde{\sigma}) = E_i^*(\mathsf{false}, \tilde{\sigma}).$$

If the results of $E_i^+(\tilde{\sigma})$ and $E_i^-(\tilde{\sigma})$ differ, then the emission of σ_i depends of itself and the program is not constructive regarding σ_i. This needs to be tested for all reachable signal states $\tilde{\sigma}$.

To simplify the application of $E_i^+(\tilde{\sigma})$ and $E_i^-(\tilde{\sigma})$ they can be combined by exclusive or ($\oplus$) into one function:

$$E_i^\oplus(\tilde{\sigma}) = E_i^+(\tilde{\sigma}) \oplus E_i^-(\tilde{\sigma}).$$

If $E_i^\oplus$ becomes true for some $\tilde{\sigma}$, then P is not constructive regarding $\tilde{\sigma}$.

If we can establish that $E_i^\oplus$ does not evaluate to true for *any* $\tilde{\sigma}$, then we know that P is constructive and we are done. This is basically a NP-complete satisfiability problem (SAT). But proving that $E_i^\oplus$ evaluates to true for *some* $\tilde{\sigma}$ is not sufficient to negate

the constructiveness of P. In this case to assert the overall constructiveness of P, all *reachable* signal statuses $\tilde{\sigma}$ must be tested against each individual $E_i^{\oplus}$ for the cycle signals. If all applications return false, then P is constructive.

The remaining hard part, however, which is not further improved here, is the identification of all reachable program states $\tilde{\sigma}$. At least the computation of replacement expressions can help to compute successor states for a given program state. The program state is expressed by the set of pause statements which start the execution in an instant. A pause statement starts the execution of an instant iff it has been reached by the control flow in the previous instant. This fact can be exploited to derive an expression for the execution of pause statements. An additional detail to distinguish the pause statements is the enumeration of all pause statement with state signals performed as part of the transformation algorithm (Step 3d). To derive the emissions of those state signals, the Equation (4.8) (Figure 4.7 on page 65) can be extended:

$$\begin{array}{lll} \textbf{pause;} & \mathcal{E}(1;!\mathsf{ST_}i, S) & = \{\langle \mathsf{next(ST_i)}, S\rangle\} \\ \textbf{emit}\ \mathsf{ST_}i & \mathcal{S}(1;!\mathsf{ST_}i, S) & = \begin{cases} \mathsf{false} : S = \mathsf{false} \\ \mathsf{ST_}i : \text{otherwise} \end{cases} \end{array} \tag{4.28}$$

The new part is the production of $\langle \mathsf{next(ST_i)}, S\rangle$ describing that "emit ST_i" and its associated pause statement will be executed in the next instant, if the current context expression S evaluates to true. If S contains input signals from the environment, then all combinations of input signals must be evaluated to get all possible successor states of the current state.

4.5.2 Temporal Induction

The constructiveness analysis described in the previous section depends on explicitly computing the entire reachable state space of the program. Claessen [11] proposes a method of temporal induction to prove the constructiveness of cyclic circuits without computation of the reachable state space. This is done by formulating *safety properties* for cycle signals. They identify stable derivations of signal states for constructive states by using a dual rail encoding for signal definitions. Furthermore *delayed definitions* express the transitions of program states between instants utilizing dual rail encodings, too.

A theorem prover is used to formally check that for all constructive states in an instant the following instants are constructive, too. This method is conservative, but very efficient, it is able to handle programs which cannot be practically handled by computation of state space.

Claessen's method to prove constructiveness of cycles is defined on circuits, but can be adopted to Esterel programs by utilizing the replacement expressions used in this work, like in the previous section.

Input: Program P, potentially containing non-constructive cycles

1. Search for signals with cyclic dependencies.
   If no cycle $\vec{C}$ can be found $\rightarrow P$ is constructive.

2. Add state signals to pause statements as in Steps (3c)/(3d) of the algorithm shown in Figure 4.2 on page 53.

3. Compute the emission context E_i for each cycle signal $\sigma_i \in \vec{C}$.
   The expressions E_i are a function of all signal states $\sigma \subset \Sigma$ in P and describe when the cycle signals are emitted:
   $(E_i(\sigma) = \mathsf{true}) \Leftrightarrow \sigma_i$ is emitted.

4. For all $\sigma_i \in \vec{C}$: Iteratively transform E_i to E_i^* by replacement of all signals $\sigma_j \in (\vec{C} \setminus \sigma_i)$ by their expressions E_j.

5. The expressions E_i^* are a function of the cycle signal $\sigma_i \in \vec{C}$ and other signals $\tilde{\sigma} = (\Sigma \setminus \vec{C})$: $E_i^*(\sigma_i, \tilde{\sigma}) \rightarrow \{\mathsf{true}, \mathsf{false}\}$

6. To be constructive, the emission of a signal must not depend on itself. Therefore the following must hold for all reachable program states $\tilde{\sigma} \subset (\Sigma \setminus \vec{C})$:
   $E_i^*(\mathsf{true}, \tilde{\sigma}) = E_i^*(\mathsf{false}, \tilde{\sigma})$

7. To test an actual program state for constructiveness,
   functions $E_i^{\oplus}(\tilde{\sigma}) \rightarrow \{\mathsf{true}, \mathsf{false}\}$ are created: $E_i^{\oplus}(\tilde{\sigma}) = E_i^*(\mathsf{true}, \tilde{\sigma}) \oplus E_i^*(\mathsf{false}, \tilde{\sigma})$
   This function returns true if P is not constructive with regard to cycle signal σ_i and program state $\tilde{\sigma}$.

8. To enumerate the reachable program state the emission of state signals can be derived as part of the computation of E_i by adding this rule to Figure 4.7:
   $\mathcal{E}(\mathsf{1;!ST_}i, S) = \{\langle \mathsf{next(ST_i)}, S \rangle\}$
   The result adds to each state signal $\mathsf{ST_}i$ the context S. When S evaluates to true in the *current* instant then $\mathsf{ST_}i$ is present in the *next* instant.

Figure 4.17: Outline of an algorithm to decide on the constructiveness of an Esterel program P. It is an alternative application of the replacement expressions introduced in Section 4.2. This is a summary of Section 4.5.1.

The dual rail encoding is applied to the replacement expressions $\sigma_i = E_i^*(\sigma_i, \tilde{\sigma})$ by splitting each binary signal σ_i into two signals σ_i^0 and σ_i^1 representing a ternary value:

$$\sigma_i^0 = E_i^{*0}(\perp, \tilde{\sigma})$$
$$\sigma_i^1 = E_i^{*1}(\perp, \tilde{\sigma}).$$

E_i^{*0} and E_i^{*1} are derived from E_i^* according to the rules of dual rail encodings given in [11]. The same pattern is applied to the delayed emissions obtained by (4.28) of state signals regarding the `pause` statements.

With these adoptions it becomes possible to directly prove the constructiveness of Esterel programs with Claessen's method without the need to synthesize a circuit beforehand.

Chapter 5

Cycles on Valued Signals

The previously presented Esterel programs featured just *pure* signals; *i. e.*, the only informational content they provide is their binary presence/absence state. This chapter outlines a possible extension of the transformation algorithm to valued signals, by way of some representative examples. Cycles involving variables are not addressed here.

5.1 Introduction to Valued Signals in Esterel

Valued signals are an extension to *pure* signals with an additional data field. The type of data must be specified at the point of signal definition. Esterel provides five primitive types: boolean, integer, float, double, and string. Additional types may be defined as a reference to the host language.

```
input A : float;
output B : integer;
signal S : integer in ... end
```

The content of a valued signal is set by a parameter on the signal name in an emit statement:

```
emit S(3)
```

This sets the value of signal S to 3. The current value of other signals can be accessed by the prefix ?:

```
emit B(?S + 1)
```

Setting a valued signal with emit will make its status present. In the next execution instant that presence status will be lost, but the value of the signal is preserved. It can be accessed by the ? operator until it is overwritten by other emissions.

The content of valued signals can be used to influence the control flow of the Esterel program similar to the presence state of signals is tested in present statements. Unfortu-

nately, signal values and presence states cannot mix in the same expression: The Esterel syntax demands a strict separation of presence and value expressions. The present, abort, suspend statements all evaluate only expressions on signal presence. The only facility to test signal values in Esterel is given by the if statement. It evaluates data expressions with a boolean result, which cannot contain any references to signal presence, just signal values are permitted.

This separation seems to be somewhat arbitrary since there is no strong technical reason to exclude access to signal presence statuses in if conditionals. This strict separation between access to signal statuses and signal values into if and present statements is a source of problems when dealing with replacement expressions for cycle signals.

If a pure signal is emitted one or more times in an instant, the results are identical: The signals state is considered "present" for the entire run time of the instant. This behavior fulfills the synchrony hypothesis, because only the first emission determined the state of the signal and further emissions are redundant. If multiple emissions on a valued signal happen in an instant (possibly with different data values), then the question arises which emitted value should be considered the valid one for the instant:

```
output A : integer;
  emit A(2);
||
  emit A(3)
```

Such an ambiguity is not acceptable in Esterel, and therefore multiple emissions on a regular valued signal in a single instant are not permitted. But nevertheless there is a way in Esterel to put a well-defined meaning into multiple emissions by binding a *combination function* to the respective signal:

```
output A : combine integer with +;
  emit A(2);
||
  emit A(3)
```

Here the operator + adds up all values of emissions on the signal A in an instant to produce a single value for the signal. In this case A will be assigned the value 5.

Not all functions are suitable for a combine operator, only commutative and associative functions produce identical results for multiple applications on more that two emissions in different orders.

5.2 Signal Dependencies on Valued Signals

Valued signals must obey the constructiveness principles just like pure signals. Not only the presence of valued signals, but also their values must be established in a constructive way, which adds additional restrictions for valid Esterel programs. The execution of an emit statement like

$P =$ **emit** $S(expr(?A))$	$\mathcal{G}(P,G,X) = G \cup \{A\}$ $\mathcal{D}(P,G,X) = \{\langle a, S\rangle \mid a \in (G \cup \{A\})\}$

if $expr(?S)$ **then**
 p
 else
 q
 end
	$\mathcal{G}_p \equiv \mathcal{G}(p, G \cup \{S\}, X)$
	 $\mathcal{G}_q \equiv \mathcal{G}(q, G \cup \{S\}, X)$
	 $\mathcal{G}(P,G,X) = \begin{cases} \bot & : (\mathcal{G}_p = \bot) \wedge (\mathcal{G}_q = \bot) \\ \mathcal{G}_p & : (\mathcal{G}_p \neq \bot) \wedge (\mathcal{G}_q = \bot) \\ \mathcal{G}_q & : (\mathcal{G}_p = \bot) \wedge (\mathcal{G}_q \neq \bot) \\ \mathcal{G}_p \cup \mathcal{G}_q & : \text{otherwise} \end{cases}$
	 $\mathcal{D}(P,G,X) = \mathcal{D}(p, G \cup \{S\}, X) \cup \mathcal{D}(q, G \cup \{S\}, X)$

Figure 5.1: Deriving signal dependencies of emissions and tests on valued signals involving expressions on valued signals. These rules extend the rule set in Figure 3.7 on page 38.

$P =$ **emit** $A(expr)$	$\mathcal{E}(P,S) = \{\langle A, S, expr\rangle\}$ $\mathcal{S}(P,S) = S$

if $expr$ **then**
 p
 else
 q
 end
	$\mathcal{E}(P,S) = \mathcal{E}(p, S \wedge expr) \cup \mathcal{E}(q, S \wedge \overline{expr})$
	 $\mathcal{S}(P,S) = \mathcal{S}(p, S \wedge expr) \vee \mathcal{S}(q, S \wedge \overline{expr})$

Figure 5.2: Equations to determine replacement expressions for emissions of valued signals. It is an extension to Figure 4.7 page 65.

```
emit S
```

on a pure signal does not depend on other signals besides control flow restrictions. But to execute an emission on valued signals like

```
emit A(?B)
```

adds a dependency to the signal B, *i. e.*, the value of B for that instant must be established before the emission statement can be executed. Such dependencies are covered in the equation in Figure 5.1. It is an extension to the equations in Figure 3.7 on page 38. It enables the cycle searching algorithm to handle cycles on valued signals. The equation for the value testing statement if/then/else is treated just like the regular present/then/else statement.

```
module VALUE_EXPR:
input A : integer;
output B : integer;

emit B(123);
pause;
present A then
   if (?A = 0) then
     emit B(?A + 1)
    else
     emit B(?A − 1)
   end if
end present

end module
```

(a)

$\rightsquigarrow$

```
module VALUE_EXPR_PREP:
input A : integer;
output B : integer;

signal ST_0, ST_1 in
  emit ST_0;
  emit B(123);
  pause; emit ST_1;
  present A then
     if (?A = 0) then
       emit B(?A + 1)
      else
       emit B(?A − 1)
     end if
  end present
end signal

end module
```

(b)

Figure 5.3: Example for emissions on valued signals: (a) Multiple emissions on signal B, (b) added state signals.

5.3 Replacement Expressions for Valued Signals

The actual resolving of cyclic dependencies involves the replacement of one or more cycle signals by a signal expression. The derivation of signal expressions for pure signals is described in Section 4.2 and uses the equations in Figure 4.7 on page 65. The extensions to the equations needed here to handle valued signals are listed in Figure 5.2.

The first equation collects emissions on valued signals into triples of signal names, current state expressions, and emitted signal values. The difference to pure signals lies in the additional value expression stored with the signal name representing the value emitted on the signal. It may consist of other signals or arithmetic operators and constants.

The second equation handles tests on signal values by if/then/else statements. They are treated like present/then/else statements by adding the predicate to the current program state. This method will mix signal state expressions with signal value expressions, although this is not valid in Esterel. Therefore when actually inserting such an expression back into the Esterel program, some more preprocessing is needed.

Figure 5.3(a) lists the short Esterel program VALUE_EXPR. It contains no cyclic dependencies, but helps to visualize the problems of replacement expressions in the context of valued signals. The output signal B is emitted three times with different arguments as values. The first one is just a numeric constant, the other two values for B are functions of the value of input signal A.

In Figure 5.3(b) state signals are added to the Esterel program to enable the derivation

of context expressions for all signal emission. The resulting context expressions for signal B are:

$$\mathcal{E}(P, \mathsf{ST_0}) = \left\{ \begin{array}{ll} \langle \mathsf{B},\ \mathsf{ST_0}, & \mathsf{123}\rangle, \\ \langle \mathsf{B},\ \mathsf{ST_1} \wedge \mathsf{A} \wedge (\mathsf{?A} = 0), & \mathsf{?A{+}1}\rangle, \\ \langle \mathsf{B},\ \mathsf{ST_1} \wedge \mathsf{A} \wedge \overline{(\mathsf{?A} = 0)}, & \mathsf{?A\text{-}1}\rangle \end{array} \right\}$$

These expressions include the actual value expressions according to the extension of derivation rules in Figure 5.2.

The next step for the replacement expressions according to the procedure as described for pure signals in Section 4.2 on page 66 would be to disjunct all state expressions for the signal emissions to yield a single expression representing the signal behavior:

$$\mathsf{B} := (\mathsf{ST_0}) \vee (\mathsf{ST_1} \wedge \mathsf{A} \wedge (\mathsf{?A} = 0)) \vee (\mathsf{ST_1} \wedge \mathsf{A} \wedge \overline{(\mathsf{?A} = 0)})$$

This attempt is incomplete, because the value expressions containing the actual emitted value is lost. Another difficulty — already mentioned before — lies in the mix of signal and value expressions which is invalid in Esterel. How this mix can be cleaned up to get valid Esterel code is illustrated in some examples in the following section.

5.4 Cycles on Pure Signals Broken by Valued Signals

Figure 5.4(a) contains an Esterel program with a cyclic dependency between two pure input signals Ain and Bin. Constructiveness is established by evaluation of a data expression on a valued signal S. Figure 5.4(b) lists the resulting program when applying the transformation algorithm presented sofar.

The remaining problem of mixed signal and value expression is addressed in Figure 5.4(c) by extraction of the expression (?S=0) and replacement by the auxiliary pure signal expr_1. The expression (?S=0) is tested in a separate if/then statement and the result propagated via expr_1 into the original expression.

This simple solution to separate mixed expressions applies only if the replaced signal itself is a pure signal. If the replaced cycle signal is a valued signal then the situation is considerably more complicated.

5.5 Cycles on Internal Valued Signals

Figures 5.5(a) and 5.8(a) contain variations of a simple cyclic dependency on valued signals A and B. The basic difference between both programs is the kind of signal definition for the cycle signals. In Figure 5.5(a) the cycle signals are local signals and in Figure 5.8(a) they are input signals of the program. The motivation for this duplication lies in the increased complexity to solve cycles on input signals. Therefore the key ideas

```
module VALUE1_CYC:

input S : integer;
input Ain, Bin;
output Aout, Bout;

if (?S = 0) then
  present Ain then
    emit Bin
  end
else
  present Bin then
    emit Ain
  end
end;

present Ain then emit Aout end;
present Bin then emit Bout end

end module
```

(a)

```
module VALUE1_MIXED_ACYC:

input S : integer;
input Ain, Bin;
output Aout, Bout;

signal ST_0, Ain_1, Bin_2 in
  emit ST_0;
  if (?S = 0) then
    present [Ain or ST_0 and not (?S = 0) and
             (Bin or ST_0 and (?S = 0))]
    then
      emit Bin_2
    end present
  else
    present [Bin or Bin_2] then
      emit Ain_1
    end present
  end if;
  present [Ain or Ain_1] then emit Aout end;
  present [Bin or Bin_2] then emit Bout end
end signal

end module
```

(b)

```
module VALUE1_ACYC:

input S : integer;
input Ain, Bin;
output Aout, Bout;

signal ST_0, Ain_1, Bin_2 in
  emit ST_0;
  if (?S = 0) then
    signal expr_1 in
      if (?S = 0) then emit expr_1 end;
      present [Ain or ST_0 and not expr_1 and
               (Bin or ST_0 and expr_1)] then
        emit Bin_2
      end present
    end signal
  else
    present [Bin or Bin_2] then
      emit Ain_1
    end present
  end if;
  present [Ain or Ain_1] then emit Aout end;
  present [Bin or Bin_2] then emit Bout end
end signal

end module
```

(c)

Figure 5.4: (a) Esterel program with a cycle on pure signals Ain, Bin which is broken by evaluation of a valued signal S, (b) Replacing Ain_1 by replacement expression with mixed signal and value expressions, (c) Value expressions are removed from replacement expression to get a pure signal expression.

```
module VALUE_PAUSE_CYC:

input  Ain : integer,
       Bin : integer;
output Aout : integer,
       Bout : integer;
signal
  A : combine integer with +,
  B : combine integer with +
in
  emit A(1);
  emit B(2);
  loop
    emit B(?A+3);
    pause;
    emit A(?B+4);
    pause;
  end
||
  loop
    present Ain then emit A(?Ain) end;
    present Bin then emit B(?Bin) end;
    present A   then emit Aout(?A) end;
    present B   then emit Bout(?B) end;
    pause
  end
end

end module
```

(a)

```
module VALUE_PAUSE_PREP:

input  Ain : integer,
       Bin : integer;
output Aout : integer,
       Bout : integer;
signal
  ST_0, ST_1, ST_2, ST_3,
  A : combine integer with +,
  B : combine integer with +
in
  emit ST_0;
  [
    emit A(1);
    emit B(2);
    loop
      emit B(?A+3);
      pause; emit ST_1;
      emit A(?B+4);
      pause; emit ST_2
    end
  ||
    loop
      present Ain then emit A(?Ain) end;
      present Bin then emit B(?Bin) end;
      present A   then emit Aout(?A) end;
      present B   then emit Bout(?B) end;
      pause; emit ST_3
    end
  ]
end

end module
```

(b)

$$\mathcal{E}(P, \mathsf{ST_0}) = \left\{ \begin{array}{ll} \langle \mathsf{A},\ \mathsf{ST_0}, & 1\rangle, \\ \langle \mathsf{A},\ \mathsf{ST_1}, & \mathsf{?B{+}4}\rangle, \\ \langle \mathsf{A},\ (\mathsf{ST_0} \vee \mathsf{ST_3}) \wedge \mathsf{Ain}, & \mathsf{?Ain}\rangle, \\ \langle \mathsf{B},\ \mathsf{ST_0}, & 2\rangle, \\ \langle \mathsf{B},\ (\mathsf{ST_0} \vee \mathsf{ST_2}), & \mathsf{?A{+}3}\rangle, \\ \langle \mathsf{B},\ (\mathsf{ST_0} \vee \mathsf{ST_3}) \wedge \mathsf{Bin}, & \mathsf{?Bin}\rangle, \end{array} \right\}$$

(c)

$$\begin{array}{lll} \textbf{emit}\ \mathsf{A(1)} & \Leftarrow & \mathsf{ST_0} \\ \textbf{emit}\ \mathsf{A(?B{+}4)} & \Leftarrow & \mathsf{ST_1} \\ \textbf{emit}\ \mathsf{A(?Ain)} & \Leftarrow & (\mathsf{ST_0} \vee \mathsf{ST_3}) \wedge \mathsf{Ain} \\ \textbf{emit}\ \mathsf{B(2)} & \Leftarrow & \mathsf{ST_0} \\ \textbf{emit}\ \mathsf{B(?A{+}3)} & \Leftarrow & (\mathsf{ST_0} \vee \mathsf{ST_2}) \\ \textbf{emit}\ \mathsf{B(?Bin)} & \Leftarrow & (\mathsf{ST_0} \vee \mathsf{ST_3}) \wedge \mathsf{Bin} \end{array}$$

(d)

Figure 5.5: Esterel program with a cycle on internal valued signals broken by pause: (a) Original program, (b) Introduction of state signals, (c) State contexts of emissions of the cycle signals A and B, (d) Alternative view of the emission contexts.

```
module VALUE_PAUSE_ACYC:

input  Ain : integer,  Bin : integer;
output Aout : integer, Bout : integer;
signal
  ST_0, ST_1, ST_2, ST_3, ST_4, ST_5,
  A : combine integer with +,
  B : combine integer with +,
  A_ : combine integer with +
in
  emit ST_0;
  [
    emit A(1); emit B(2);
    loop
      emit B(?A_+3);
      pause; emit ST_1;
      emit A(?B+4);
      pause; emit ST_2
    end
  ||
    loop
      present Ain then emit A(?Ain) end;
      present Bin then emit B(?Bin) end;
      present A   then emit Aout(?A) end;
      present B   then emit Bout(?B) end;
      pause; emit ST_3
    end
  ]
||
  loop
    present ST_0                                    then emit A_(1) end;
    present [ST_1 and ST_0]                         then emit A_(2+4) end;
    present [ST_1 and (ST_0 or ST_3) and Bin] then emit A_(?Bin+4) end;
    present [ST_1 and (ST_0 or ST_3) and Ain] then emit A_(?Ain) end;
    present [ST_1 and not (ST_0 or ((ST_0 or ST_3) and Bin))]
                                                    then emit A_(pre(?B)+4) end;
    pause; emit ST_4
  end loop
end
end module
```

Figure 5.6: Acyclic transformation of the cyclic Esterel program in Figure 5.5.

```
module VALUE_PAUSE_OPT_ACYC:

input  Ain  : integer,
       Bin  : integer;
output Aout : integer,
       Bout : integer;

signal
  ST_0, ST_1,
  A : combine integer with +,
  B : combine integer with +,
  A_ : combine integer with +
in
  emit ST_0;
  [
    emit A(1);
    emit B(2);
    loop
      emit B(?A_+3);
      pause; emit ST_1;
      emit A(?B+4);
      pause
    end
   ||
    loop
      present Ain then emit A(?Ain) end;
      present Bin then emit B(?Bin) end;
      present A   then emit Aout(?A) end;
      present B   then emit Bout(?B) end;
      pause
    end
  ]
||
  loop
    present ST_0                  then emit A_(1) end;
    present [ST_1 and Bin]        then emit A_(?Bin+4) end;
    present [ST_1 and Ain]        then emit A_(?Ain) end;
    present [ST_1 and not Bin]    then emit A_(pre(?B)+4) end;
    pause
  end
end

end module
```

Figure 5.7: Optimized version of the transformation in Figure 5.6.

are presented on internal cycle signals first, followed by the extensions needed for input signals.

The program in Figure 5.5(a) contains two parallel loops: The first loop contains the cyclic dependency between A and B: The value of A is propagated to B and vice versa. Constructiveness is assured by separating both emit statements into different instants by a pause statement. The second one propagates the values of input signals Ain and Bin to the internal signals A and B, and from A and B back to the output signals Aout and Bout. This construction makes the behavior of the cycle signals A and B visible to the environment.

Both signals account for several emissions with different sources for values in each instant. This makes the use of the combine operator necessary for these signals. In this program the simple + is used for this task.

To resolve the cyclic dependencies, replacement expressions must be derived for both cycle signals A and B. As a preparation for this task, state signals ST_0 to ST_3 are introduced in Figure 5.5(b). Applying the derivation rules of Figure 4.7 with extensions from Figure 5.2 to this program yields the expressions in Figure 5.5(c).

Figure 5.5(d) contains an alternative view on the state context of emissions derived in Figure 5.5(c). It lists each emit statement involving cycle signals on the left hand and its associated state context on the right hand. Basically this view will be used in the following transformations, only the emit keyword will sometimes be left out to save space.

The next step is the selection of a signal to cut the cycle: Here A is selected arbitrarily. The cutting of this cycle is based on deriving a replacement expression for this signal without using any cycle signals. This replacement will break the cycle. The algorithm to derive such an replacement expression for pure signals is described in Chapter 4. It will derive an expression to describe the presence of the signal in the program's state context. The additional complication for valued signals lies in the derivation of the value while avoiding references to values of other cycle signals.

To obtain a replacement expression for A, all references to other cycle signals must be replaced by their respective expressions. In the case of valued signals, these references may not only occur in the context state of the emission, but also in the expression of the emitted value. These occurrences must be replaced, too. To illustrate this, the state contexts of emissions on cycle signals A and B are repeated here:

$$\textbf{emit } A(1) \Leftarrow ST_0 \tag{5.1}$$
$$\textbf{emit } A(?B+4) \Leftarrow ST_1 \tag{5.2}$$
$$\textbf{emit } A(?Ain) \Leftarrow (ST_0 \vee ST_3) \wedge Ain \tag{5.3}$$
$$\textbf{emit } B(2) \Leftarrow ST_0 \tag{5.4}$$
$$\textbf{emit } B(?A+3) \Leftarrow (ST_0 \vee ST_2) \tag{5.5}$$
$$\textbf{emit } B(?Bin) \Leftarrow (ST_0 \vee ST_3) \wedge Bin \tag{5.6}$$

To build a replacement for A, a new signal A_ is introduced and the emissions on A applied to A_:

$$\textbf{emit}\ \mathsf{A_(1)} \Leftarrow \mathsf{ST_0} \tag{5.7}$$
$$\textbf{emit}\ \mathsf{A_(?B+4)} \Leftarrow \mathsf{ST_1} \tag{5.8}$$
$$\textbf{emit}\ \mathsf{A_(?Ain)} \Leftarrow (\mathsf{ST_0} \vee \mathsf{ST_3}) \wedge \mathsf{Ain} \tag{5.9}$$

Equations (5.7) and (5.9) can be used directly, they contain no references to other cycle signals. Equation (5.8) uses cycle signal B, which must be replaced. This replacement produces three emissions by applying (5.4), (5.5), and (5.6).:

$$\textbf{emit}\ \mathsf{A_(2+4)} \Leftarrow \mathsf{ST_1} \wedge \mathsf{ST_0} \tag{5.10}$$
$$\textbf{emit}\ \mathsf{A_(?A+3+4)} \Leftarrow \mathsf{ST_1} \wedge (\mathsf{ST_0} \vee \mathsf{ST_2}) \tag{5.11}$$
$$\textbf{emit}\ \mathsf{A_(?Bin+4)} \Leftarrow \mathsf{ST_1} \wedge (\mathsf{ST_0} \vee \mathsf{ST_3}) \wedge \mathsf{Bin} \tag{5.12}$$

Equation (5.11) contains a reference to the cycle signal A. This is the cycle signal which is intended to be replaced and therefore constitutes a self reference. It is equivalent to the self-reference of replacement expressions described in Step (6d) on page 62 of the transformation algorithm. With a similar reasoning founded in the constructiveness of the original program, the non-constructive part of this replacement emission is assumed as not effective at runtime of the program and is therefore just skipped. And in fact by inspection of the state context expression on the right hand side of (5.11) its non-fulfillment is obvious; the referenced state signals are mutually exclusive.

One aspect of valued signals had been neglected at the derivation of (5.10) and (5.12) out of (5.8): If valued signals are not emitted in an instant, then they keep the value from the previous instant. In this case B may be not emitted in an instance therefore keeping the value from the previous instance, *i. e.*, prev(?B). This behavior is not covered yet. This context state is identified by conjuncting the state expression from (5.2) with the negation of the disjunction of the state expressions from (5.4) and (5.6):

$$\textbf{emit}\ \mathsf{A_(prev(?B)+4)} \Leftarrow \mathsf{ST_1} \wedge \overline{\mathsf{ST_0} \vee ((\mathsf{ST_0} \vee \mathsf{ST_3}) \wedge \mathsf{Bin})} \tag{5.13}$$

When this state expression is fulfilled, then B is not emitted in the current instant. Therefore the value of B from the previous instant must be used to build the current A_. The current value of B could be used, too (since it is not overwritten in the current instant). But this would introduce an additional dependency from B to A, which leads to an additional cyclic dependency, thus spoiling the effort to resolve the cycle in the first place.

If the used Esterel compiler does not provide the functionality of the pre() operator, then the following substitute might help:

```
var v : integer in
  loop
    present B then v := ?B end;
    pause;
    emit pre_B(v)
  end loop
end var
```

When this small code is added as a parallel thread to the program, then use of pre(?B) can replaced by ?pre_B. It must be noted, however, that pre_B does not fully implement

the pre(B) operator. The presence state of B in the previous instant is not covered; pre_B is always present. But in the context of cycle transformation, the full implementation of pre() is not needed.

The final set of emissions comprising equations (5.7), (5.9), (5.10), (5.12), and (5.13) are used to establish the presence and value of A_:

```
loop
  present ST_0                                              then emit A_(1) end;
  present [ST_1 and ST_0]                                   then emit A_(2+4) end;
  present [ST_1 and (ST_0 or ST_3) and Bin]                 then emit A_(?Bin+4) end;
  present [ST_1 and (ST_0 or ST_3) and Ain]                 then emit A_(?Ain) end;
  present [ST_1 and not (ST_0 or ((ST_0 or ST_3) and Bin))] then emit A_(prev(?B)+4) end;
  pause
end
```

This multiple emission on a single signal relies on the combine operator + on A_ which is inherited from A.

This code block is added as a parallel thread into program VALUE_PAUSE_ACYC in Figure 5.6. The (single) testing occurrence of A in the cycle is replaced by A_, since A_ simulates the presence and value of A. The program is now acyclic because A_ does not depend on any cycle signals.

There exists another testing occurrence of A in the second loop block regarding the emission of Aout. This occurrence does not need to be replaced since it is not part of the cycle. Replacing it would effectively remove all testing occurrences of A and therefore even all emissions and the declaration of A could be removed.

The program VALUE_PAUSE_ACYC can be optimized further by detecting that some expressions on the state signals are either always true or false at the point of their evaluation. This leads to just Bin remaining in the present condition. Some state signals are now not referenced and can be removed. This final optimized version of the program is listed in Figure 5.7.

5.6 Preprocessing of Cyclic Valued input Signals

The program VALUE_INPUT_CYC in Figure 5.8(a) is a variation of VALUE_PAUSE_CYC in Figure 5.5(a). It contains a cyclic dependency on two valued signals A and B, too. However, A and B are input signals in this case. Cycles on input signals cannot be resolved directly for reasons laid out in the notes on Step (5) (page 58) of the transformation algorithm. Therefore, to be able to derive replacement expressions for those cycle signals, the cycle needs to be moved to internal signals.

In this section only the renaming of the cycle signals into internal signals is addressed, the remaining part of the transformation can be done by application of the procedure in the previous section on cyclic internal signals.

```
module VALUE_INPUT_CYC:

input  A : combine integer with +,
       B : combine integer with +;
output Aout : integer,
       Bout : integer;

  loop
    emit B(?A);
    pause;
    emit A(?B);
    pause
  end
||
  loop
    present A then emit Aout(?A) end;
    present B then emit Bout(?B) end;
    pause
  end

end module
```

(a)

```
module VALUE_INPUT_PREP:

input  A : combine integer with +,
       B : combine integer with +;
output Aout : integer,
       Bout : integer;

signal
  ST_0, ST_1, ST_2, ST_3, ST_4,
  A_  : combine integer with +,
  B_  : combine integer with +,
  A__ : combine integer with +,
  B__ : combine integer with +
in
  emit ST_0;
  [
    loop
      emit B_(?A__);
      pause; emit ST_1;
      emit A_(?B__);
      pause; emit ST_2
    end
  ||
    loop
      present (A or A_) then emit Aout(?A__) end;
      present (B or B_) then emit Bout(?B__) end;
      pause; emit ST_3
    end
  ]
||
  loop
    present A then
      present A_ then emit A__(?A+?A_)
      else emit A__(?A) end
    else
      present A_ then emit A__(?A_) end
    end;
    present B then
      present B_ then emit B__(?B+?B_)
      else emit B__(?B) end
    else
      present B_ then emit B__(?B_) end
    end;
    pause; emit ST_4
  end
end

end module
```

(b)

Figure 5.8: Esterel program with a cycle on valued input signals broken by pause: (a) Original program, (b) Introduction of state signals and renaming of cyclic input signals.

The basic idea of such a renaming of cyclic input signals is to introduce for each cyclic input signal (*e. g.*, A) a new internal signal (*e. g.*, A_). All emissions on A inside the program are relocated to A_ and all tests on A are replaced for a test of the combination of A and A_. The input signal A is now set only by the environment and not by the program itself anymore. This moves the functionality of the cycle from A to A_.

The key problem for cycles on valued input signals lies in the need for a "test of the *combination* of A and A_". For pure signals the combination is a simple "or" operation. For valued signals the current value of a signal may be combined from several emissions in that instant, needing a combine function. Alternatively the value is inherited from the previous instant if the signal is not emitted currently. These distinct cases are here implemented by introducing further helper signals (*e. g.*, A__) for each cycle signal. These helper signals will represent the current presence state and value of the former cyclic input signals.

To set up the state of a helper signal A__ the following cases must be considered.

- A and A_ are present: The values of A and A_ are combined with the use of the defined combine operator and emitted into A__.
- Only A is present: The value of A is emitted into A__.
- Only A_ is present: The value of A_ is emitted into A__.
- Neither A nor A_ are present: No new value is emitted into A__, making it absent. The value of A__ in the previous instant is preserved.

This distinction of different cases is defined in a parallel thread to the original program. Figure 5.8(b) contains the program VALUE_INPUT_PREP as a preprocessed version of VALUE_INPUT_CYC with added state signals and renamed cyclic input signals.

The additional signals A__ and B__ are not independent of the cycle, they enlarge the cycle. The cycle covers now:

$$\vec{C} = \langle \mathsf{A__}, \mathsf{A}, \mathsf{B__}, \mathsf{B} \rangle$$

Computing the context states of emissions on these cycle signals yields:

$$\textbf{emit } \mathsf{A_(?B__)} \Leftarrow \mathsf{ST_1} \quad (5.14)$$

$$\textbf{emit } \mathsf{B_(?A__)} \Leftarrow (\mathsf{ST_0} \vee \mathsf{ST_2}) \quad (5.15)$$

$$\textbf{emit } \mathsf{A__(?A+?A_)} \Leftarrow (\mathsf{ST_0} \vee \mathsf{ST_4}) \wedge \mathsf{A} \wedge \mathsf{A_} \quad (5.16)$$

$$\textbf{emit } \mathsf{A__(?A)} \Leftarrow (\mathsf{ST_0} \vee \mathsf{ST_4}) \wedge \mathsf{A} \wedge \overline{\mathsf{A_}} \quad (5.17)$$

$$\textbf{emit } \mathsf{A__(?A_)} \Leftarrow (\mathsf{ST_0} \vee \mathsf{ST_4}) \wedge \overline{\mathsf{A}} \wedge \mathsf{A_} \quad (5.18)$$

$$\textbf{emit } \mathsf{B__(?B+?B_)} \Leftarrow (\mathsf{ST_0} \vee \mathsf{ST_4}) \wedge \mathsf{B} \wedge \mathsf{B_} \quad (5.19)$$

$$\textbf{emit } \mathsf{B__(?B)} \Leftarrow (\mathsf{ST_0} \vee \mathsf{ST_4}) \wedge \mathsf{B} \wedge \overline{\mathsf{B_}} \quad (5.20)$$

$$\textbf{emit } \mathsf{B__(?B_)} \Leftarrow (\mathsf{ST_0} \vee \mathsf{ST_4}) \wedge \overline{\mathsf{B}} \wedge \mathsf{B_} \quad (5.21)$$

Equations (5.14) and (5.15) result from those two emit statements comprising the cyclic dependency in the original program. The synthesis of the helper signals A__ and B__ is

described by Equations (5.16) to (5.21). They depend on input signals A and B and the local cycle signals A_ and B_. Equations (5.16) and (5.19) make use of the combination function "+" as defined for the input signals A and B.

The remaining part of the transformation can be handled by the procedure introduced in the previous section.

5.7 Transforming mejia.strl from Estbench

The Estbench Esterel Benchmark Suite [10] collected by Edwards contains some Esterel programs intended for comparisons of Esterel compilers. One included program is mejia.strl. Most Esterel compilers are not able to produce code for this program because of cyclic dependencies. The cycle includes valued signals which poses problems even for the causality analysis implemented in the v5 compiler. The v5 compiler is only able to produce interpretation code (option -I) for mejia.strl that defers the signal dependency and scheduling analysis to runtime.

The original mejia.strl is separated in several modules and relies heavily on non kernel statements. As a preprocessing step, the modules are expanded and non-kernel statements replaced by blocks of kernel statements. The extension of the transformation algorithm to support valued signals is not yet implemented. Therefore the transformation of mejia.strl must be done manually. This is much too error prone, because the preprocessed program is several hundred lines of code long.

The way out taken here is to ruthlessly cut the preprocessed program down into the version listed in Figure 5.9(a). That version includes a cyclic dependency and is not handled by the causality analysis of the v5 compiler, too.

Searching for signal dependencies in the program STATION_CYC from Figure 5.9(a) according to the algorithm presented in Chapter 3 yields the following set:

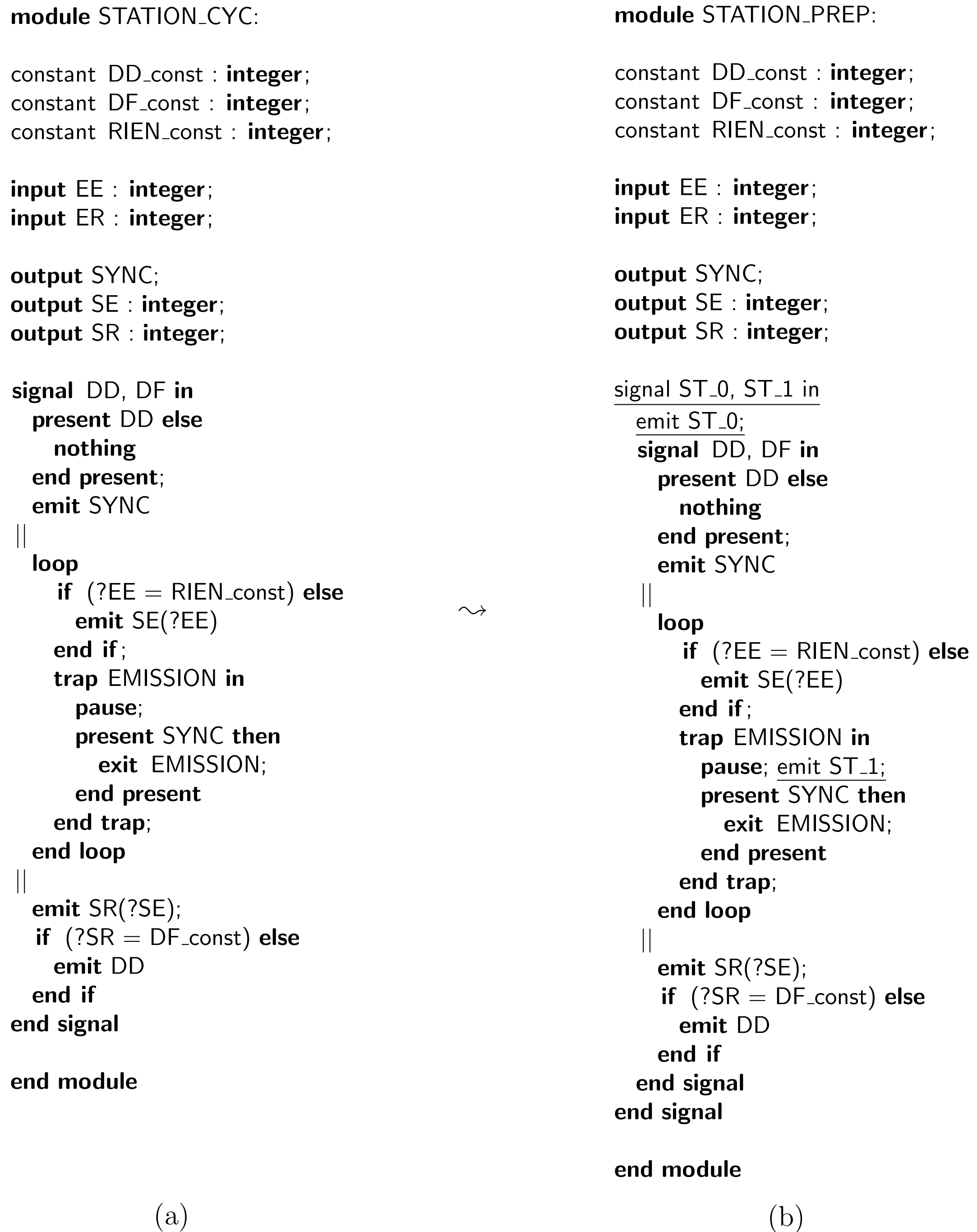
$$D = \left\{ \begin{array}{l} \langle \mathsf{EE}, \mathsf{SE} \rangle, \\ \langle \mathsf{SYNC}, \mathsf{SE} \rangle, \langle \mathsf{SYNC}, \mathsf{EMISSION} \rangle, \\ \langle \mathsf{SE}, \mathsf{DD} \rangle, \langle \mathsf{SE}, \mathsf{SR} \rangle, \\ \langle \mathsf{SR}, \mathsf{DD} \rangle, \\ \langle \mathsf{DD}, \mathsf{SYNC} \rangle \end{array} \right\}$$

Searching in D for cyclic dependencies delivers the following cycle of length three:

$$C = \langle \mathsf{SYNC}, \mathsf{SE}, \mathsf{DD} \rangle$$

Renaming of cycle signals is not needed here, because all three cycle signals are either internally defined or output signals and none of them is an input signal.

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



```
module STATION_CYC:

constant DD_const : integer;
constant DF_const : integer;
constant RIEN_const : integer;

input EE : integer;
input ER : integer;

output SYNC;
output SE : integer;
output SR : integer;

signal DD, DF in
  present DD else
    nothing
  end present;
  emit SYNC
||
  loop
    if (?EE = RIEN_const) else
      emit SE(?EE)
    end if;
    trap EMISSION in
      pause;
      present SYNC then
        exit EMISSION;
      end present
    end trap;
  end loop
||
  emit SR(?SE);
  if (?SR = DF_const) else
    emit DD
  end if
end signal

end module
```

(a)

~>

```
module STATION_PREP:

constant DD_const : integer;
constant DF_const : integer;
constant RIEN_const : integer;

input EE : integer;
input ER : integer;

output SYNC;
output SE : integer;
output SR : integer;

signal ST_0, ST_1 in
  emit ST_0;
  signal DD, DF in
    present DD else
      nothing
    end present;
    emit SYNC
  ||
    loop
      if (?EE = RIEN_const) else
        emit SE(?EE)
      end if;
      trap EMISSION in
        pause; emit ST_1;
        present SYNC then
          exit EMISSION;
        end present
      end trap;
    end loop
  ||
    emit SR(?SE);
    if (?SR = DF_const) else
      emit DD
    end if
  end signal
end signal

end module
```

(b)

Figure 5.9: Application of the transformation on a heavily cutdown version of mejia.strl: (a) cyclic cutdown version of the original, (b) added state signals.

```
module STATION_ACYC:

constant DD_const : integer;
constant DF_const : integer;
constant RIEN_const : integer;

input EE : integer;
input ER : integer;

output SYNC;
output SE : integer;
output SR : integer;

signal ST_0, ST_1 in
  emit ST_0;
  signal DD, DF in
    present DD else
      nothing
    end present;
    emit SYNC
  ||
    loop
      if (?EE = RIEN_const) else
        emit SE(?EE)
      end if;
      trap EMISSION in
        pause; emit ST_1;
        present ST_0 then
          exit EMISSION;
        end present
      end trap;
    end loop
  ||
    emit SR(?SE);
    if (?SR = DF_const) else
      emit DD
    end if
  end signal
end signal

end module
```

Figure 5.10: Acyclic transformation of the program in Figure 5.9.

$$\begin{aligned}
\textbf{emit}\ \mathsf{SYNC} &\Leftarrow \mathsf{ST_0}\\
\textbf{emit}\ \mathsf{SE(?EE)} &\Leftarrow (\mathsf{ST_0} \vee \mathsf{ST_1} \wedge (\mathsf{SYNC} \vee \overline{\mathsf{SYNC}})) \wedge \overline{(\mathsf{?EE} = \mathsf{RIEN_const})}\\
\textbf{emit}\ \mathsf{DD} &\Leftarrow (\mathsf{ST_0} \vee \mathsf{ST_3}) \wedge \mathsf{Ain}
\end{aligned}$$

The replacement expression for signal SYNC is derived with the optimization in Section 6.1 in mind. That optimization applies if a present statement does not change the program state and a following program block is not influenced besides the sequential execution. Omitting this optimization, the expression for SYNC would be derived as:

$$\textbf{emit}\ \mathsf{SYNC} \Leftarrow \mathsf{ST_0} \wedge (\mathsf{DD} \vee \overline{\mathsf{DD}})$$

If the optimization is applied then the cutting of the cycle is fairly simple: signal SYNC depends on no other cycle signal and can be easily replaced by the state signal ST_0. The resulting acyclic program STATION_ACYC is listed in Figure 5.10. It can now be compiled by the CEC and the v5 compiler without causality analysis.

The successful transformation of the cutdown version of mejia.strl is an indication that the application on the original mejia.strl might be successful, too. Assurance could only be achieved by a full implementation of the transformation algorithm including the treatment of valued signals.

Chapter 6

Optimizations

Figures 4.2 (page 53) and 4.7 (page 65) contain the algorithm and equations for replacement expressions in its basic form without any additional optimizations. In the following some possible improvements on the algorithm are presented.

6.1 Replacement Expression for present

The most important optimization refines the treatment of the present statement in Equation (4.6). Consider the following program fragment:

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

The application of the rules listed in Figure 4.7 on page 65 would result in a replacement expression for signal $\mathsf{C} = (\mathsf{ST_3} \wedge \mathsf{S}) \vee (\mathsf{ST_3} \wedge \overline{\mathsf{S}})$. It is obvious that this can be simplified to $\mathsf{C} = \mathsf{ST_3}$, since neither present branch has an influence on the emission of C. Or more generally:

Equation (4.6):

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

can be simplified to

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

if

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

holds.

```
present A then          emit ST_0;
  nothing               present A then
end;                      nothing
emit B;                 end;
pause;          ~>      emit B;            A := ST_1 ∧ B
present B then          pause;             B := ST_0
  emit A;               emit ST_1;
end                     present B then
                          emit A;
                        end
```

Figure 6.1: Cyclic dependency resolved without iterative replacement: The cycle between signals A and B is not reflected by a cycle in the respective emission contexts.

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

Another side effect of this optimization is an elimination of cyclic dependencies for some programs. Consider the program fragment in Figure 6.1 and the preprocessed version with added state signals (signal renamings are not necessary here):

This program contains a cyclic dependency between the signals A and B. Signal A is emitted in state ST_1 under the guard of signal B, therefore yielding a replacement expression for A:

$$A := ST_1 \wedge B$$

Signal B is not under the direct control of the present block on signal A, but the emission of B depends on the termination of that present block. The present block A is evaluated with a context expression $ST_0 \wedge A$ for the "then nothing end" branch and $ST_0 \wedge \overline{A}$ for the implicit "else nothing end" branch. Both branches do not change their respective context expression, and if the optimization suggested in this section is applied, then the context expression for the emission of B is:

$$B := ST_0$$

The useful thing visible here is the missing reference to other cycle signals (*i. e.*, A) in the replacement expression for B. This could be exploited in selecting signal B as the point of cutting the cyclic dependency. B should be preferred over A in this example, because B spares the need to perform an iterative replacement of other cycle signals by their respective context expressions. This would yield a smaller transformed acyclic program.

Despite forming cyclic signal dependencies, the replacement expressions for A and B are not cyclically dependent. The reason for this anomaly lies in the different handling of block terminations for signal dependencies and (optimized) context expressions for use in replacement expressions.

Another example of such a simplified transformation is described in Section 5.7.

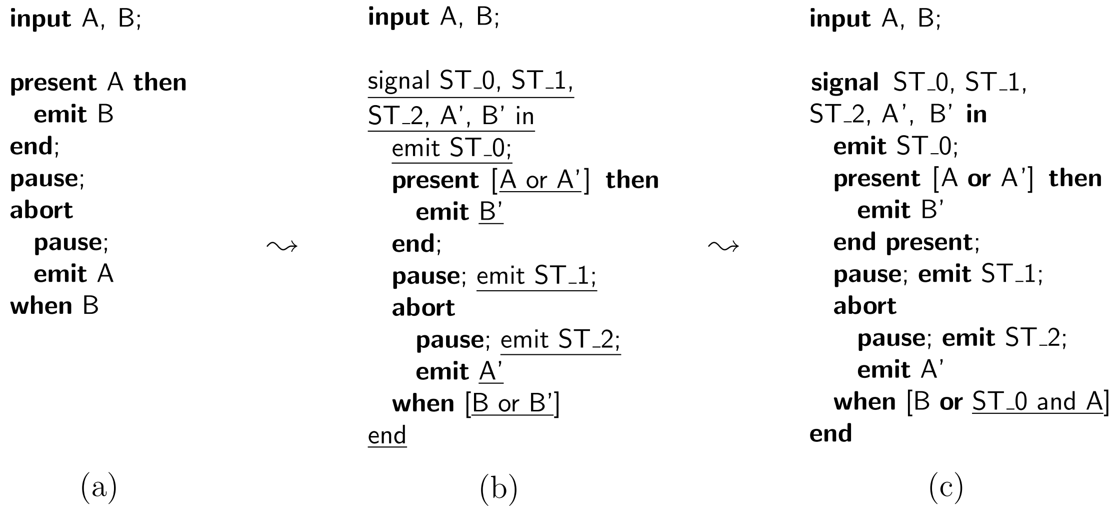


Figure 6.2: Successful resolving of a cyclic dependency involving present and abort statements: (a) original program, (b) preprocessing by introduction of state signals and signal renaming, (c) replacement of tests for B' in abort guard by an expression.

6.2 Termination of Parallel Statements

The general transformation algorithm for parallel statements calls for instrumentation to detect the termination of parallel statements at runtime. It involves additional signals, sustain statements, and a test for those signals (see Figure 4.4, page 57).

These additions are not needed, if the parallel statement cannot terminate at all at runtime, *e. g.*, one thread contains a loop statement on the top level. This is the case for the Token Ring Arbiter (Figure 3.5).

A more formal characterization of such a thread p with entry context S is:

$$\mathcal{S}(p, S) = \mathsf{false}$$

If one such thread can be found for a parallel statement, then the extensions in Figure 4.4 are not needed for that parallel statement. Using false as the termination context is sufficient here.

6.3 Interaction of Parallel Termination with Exceptions

The interaction of parallel threads with exception handling leaves significant room for optimization in the current implementation. Currently each pause statement contained in a trap block is considered as a point where the control is handed over to the end of the trap block. This is motivated by parallel threads which terminate each other by exit statements. Figure 3.10(a) contains such an example.

The current implementation of dependency analysis returns dependencies even for unreachable control flows. In Figure 3.10(b) such an example is listed. As an optimization this could be limited to actually (statically) reachable exceptions at pause statements.

Furthermore the hierarchy of traps limits the reachability of some kinds of signal dependencies. An example is listed in Figure 3.6(c) (page 36): The dependency of A to B is not reachable because of the priority of T1 over T2.

6.4 Substitution of suspend and abort

Step (3a) of the algorithm calls for a replacement of all suspend/abort statements, for the reasons outlined on page 55. However, the transformation algorithm does not necessarily fail on all programs containing suspend or abort statements. If the suspend/abort statements are not part of the cycle, then they trivially pose no problem. But even some participation in the cycle can be resolved if a way can be found to replace the suspend/abort guard predicates by non-cyclic expressions. Figure 6.2 contains such an example:

The cycle runs over two signals A and B. B is emitted under a present guard on A and A is emitted under an abort guard on B. A and B are renamed into A' and B' in Figure 6.2(b). A' and B' are emitted in the following contexts:

$$A' := ST_2 \wedge \overline{B \vee B'}$$

$$B' := ST_0 \wedge (A \vee A')$$

For this program, the key in successfully resolving the cyclic dependency lies in the selection of whether A' or B' is substituted by an expression. Replacing A' will not resolve the cycle, because the replacement expression for A' contains ST_2, which depends on B' in the abort statement. By choosing B' it is possible to break the cycle. Substituting A' in the expression for B' yields:

$$B' := ST_0 \wedge (A \vee (ST_2 \wedge \overline{B \vee B'}))$$

Replacing the remaining B' by true results in:

$$B' := ST_0 \wedge A$$

This expression replaces B' in the suspend guard in Figure 6.2(c) and results in an acyclic program.

The following optimization hierarchy can be proposed as a conclusion:

- Cutting of a cycle containing suspend/abort statements without explicit elimination of suspend/abort, if the derived replacement expressions do not depend on suspend/abort guards.
- Successive elimination of the suspend/abort statements, until the cycle cutting algorithm is applicable.

- Replacement of only those suspend/abort statements that are part of the cycle.

These optimizations have the potential to reduce the cost in code size to handle suspend/abort in cyclic Esterel programs.

6.5 Signal Renamings for Locally Defined Signals

Cycle signals that are part of the input interface of the program must be renamed, to separate emissions from the environment from those of the program itself. output signals need not and cannot be renamed without altering the interface behavior of the program.

Besides these fixed rules for interface signals, the internally defined signals in a signal block can optionally be renamed. It depends on the degree of usage as part of the cycle or outside the cycle if such a renaming results in better or worse efficiency.

The current implementation of the transformation algorithm does not rename internal cycle signals.

6.6 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 3.2(d) on page 32 by taking the reachable presence status of the signals ST_0 and ST_1 into account.

The replacement expression E_i^* (Step (6e) of the algorithm) may reference some state signal $ST_j \in ST$ that can be shown to be always present or absent:

1. If E_i^* replaces σ_i' in E_k, and we know that at this location in the program, ST_j must always be present, then we can replace ST_j by the constant true in E_i^*.

   In the program PAUSE_ACYC, this applies to the state signal ST_0 in the replacement expression (ST_1 and (B or ST_0)), which we therefore can simplify to (ST_1 and B).

   More generally, we can replace a state signal by the constant true if we know that it must be emitted in every instant. As it turns out, there cannot be any state signals that fulfill this condition by themselves; the boot state signal ST_0 is only present in the initial instant, and all other state signals are emitted only after a pause statement, and hence cannot be emitted in the initial instant. However, another optimization is possible:

2. If a state signal is emitted in every instant except for the initial instant, we can replace it with pre(tick). This replacement eliminates the need to introduce the state signal into the program.

In the Arbiter, for example, the state signals ST_1, ST_4, and ST_7 are the only state signals emitted in loops that run concurrently to the rest of the program. Hence, as loops must not be instantaneous, they must be emitted at every iteration and are present at every instant except the initial one.

3. Tests for "ST_0 or pre(tick)" can be replaced by true.

   This applies for example to the Token Ring Arbiter, where we know that all guarded emits that constitute the cycle are evaluated in every instant of the program. Hence this rule, together with the previous rule, leads to the simplified replacement expression already stated in Equation (4.27).

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

   In the program PAUSE_ACYC, this applies to the state signal ST_1; due to the pause statement between the evaluation of the replacement expression and the emission of ST_1, we can set ST_1 to false in the replacement expression. In this case, this reduces the whole replacement expression to false; therefore, the "[A or (ST_1 and (B or ST_0))]" from PAUSE_ACYC gets reduced to just A in PAUSE_OPT.

6.7 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.

6.8 Absence of External Tests of Cycle Breaking Signal

If the signal σ_p that is selected in Step (6b) to break the cycle is not tested outside of the cycle, this means that after replacing the tests for σ_p within the cycle (Step 6e) by E_i^*, the signal σ_p is not tested anywhere in the program. One can therefore eliminate its emission.

Emissions of output signals must not be removed, because emissions of them must reach the outside interface.

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

6.9 Simplification of External Tests

Depending on how often one must replace a particular signal σ_i in Step (5c) by the expression "(σ_i or σ_i')," it may be beneficial to introduce another fresh signal σ_i''. This signal must be emitted whenever σ_i or σ_i' are present, for example using a new globally parallel statement of the form "every [σ_i or σ_i'] do emit σ_i'' end." Then it suffices to replace tests for σ_i by tests for σ_i''.

6.10 Compiler-Specific Signal Dependencies

The semantics of Esterel [7] defines the reaction of Esterel programs on input signals and/or internal signal emissions. Different compilers implement the same semantics and therefore the behavior of compiled programs is expected to be the same independent of the compiler.

The basic criterion for a valid Esterel program — constructiveness — is well defined, too. But most Esterel compilers demand a stricter property on Esterel programs: Absence of cyclic signal dependencies. As noted in Section 3.3, cyclic signal dependencies are computed on the set of signal dependencies found in the program. Signal dependencies are *not* formally defined and as a consequence there are some differences between Esterel compilers on the notion of signal dependencies. This leads to different sets of signals dependencies found by different compilers. Therefore the same input program may be accepted by one compiler but rejected as cyclic by another one.

Two short example programs are listed in Section 3.3 in Figure 3.6 (a) and (b) on page 36. Each program is accepted by the v5 compiler and rejected by the CEC compiler and vice versa.

To make all constructive programs compilable by all Esterel compilers, cyclic dependencies must be removed. To support all Esterel compilers, identification of a superset of signal dependency types would be needed. But implementing this superset in the detection of cycles would introduce unnecessary transformations for cycles that are not even noted by some compilers.

An optimized version could make use of "compiler profiles" to match different compilers implementations of signal dependencies. This would enable the production of efficient transformed code for each individual compiler.

6.11 Lifting of Locally Defined Signals

The transformation algorithm (Figure 4.2, page 53) 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 may transport references to local signals out of their

respective scope. Demanding the relocation of *all* local signals is in most cases too conservative and certainly not easily doable either.

It would be more efficient to detect possible conflicts with replacement expressions first and then to relocate the problematic signals.

Chapter 7

Experimental Results

The proposed transformation has been implemented as an extension of the Columbia Esterel Compiler (CEC). The implementation handles cycles involving pure signals, using the algorithm presented in Chapter 4, except that local signals are not moved up to a global level (Step (2c)). The extensions towards valued signals outlined in Chapter 5 are not implemented. In addition, the optimization explained in Section 6.1 is implemented, but not the other ones.

For an experimental evaluation, we have defined several variants of the Token Ring Arbiter.

TR3: This is the Token Ring Arbiter with three network stations. The implementation is as in Figure 3.5.

TR10: This is an extension of TR3 from three to ten network stations. The aim is to test the scaling of the algorithm for code size and runtime of the resulting binary.

TR50, ... TR1000: These versions contain fifty to one thousand network stations. They are used to estimate the factor of growth for big program sizes and to measure the performance of the transformation itself.

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

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

7.1 Synthesizing Software

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

v5-L: The publicly available Esterel compiler v5.92 [8, 18]. It is used in this case with option -L to produce code based on the circuit representation of Esterel. The code is organized as a list of equations ordered by dependencies. This results in a fairly compact code, but with a comparatively slow execution speed. This compiler is able to handle constructive Esterel programs with cyclic dependencies.

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

v7: The Esterel v7 compiler (available from Esterel Technologies [17]) is used here in version v7_10i8 to compile acyclic code based on sorted equations, as the v5 compiler.

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

CEC: The Columbia Esterel Compiler (release 0.3) [9] produces event-driven C code, which is generally quite fast and compact. However, this compiler cannot handle cyclic dependencies. Thus it can only be applied to the transformed cyclic programs.

CEC-g: The CEC with the option -g produces code using computed goto targets (an extension to ANSI-C offered by GCC-3.3 [19]) to reduce the runtime even further.

A simple C back-end is provided for each Esterel program to produce input signals and accept output signals to and from the Esterel part. The back-end 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, 256KB Cache, 1GB Main Memory).

Table 7.1(a) compares the execution speed and binary sizes of the example programs for the v5, v7, and CEC compilers with their respective options. The v5 compiler is applied both to the original cyclic programs and the transformed acyclic programs. The CEC and v7 compiler can handle only acyclic code.

When comparing the runtime results of the v5 compiler (with sorted equations) for the cyclic and acyclic versions of the token ring arbiter, there is a noticeable reduction in runtime for the transformed acyclic programs. This came as a bit of a surprise. It seems that the v5 compiler is a little bit less efficient in resolving cyclic dependencies in sorted equations. For the automaton code there are only minor differences in runtime.

Table 7.1 includes the sizes of the compiled binaries, too. All compilers produce code of similar sizes, but with one exception: the v5 compiler produces a very big automaton code for the third token ring example. That program contains several parallel threads which are only loosely related. If someone tries to map such a program on a flat automaton, it is well known that such a structure results in a "state explosion". Actually, we had to limit the number of parallel tasks in this example to get the program to compile in reasonable time. While the v5 compiler seems to be competitive with respect to program run times, the binary sizes can reach several times the size of the binaries produced by the other compilers.

Variant	Compiler	TR3	TR10	TR10p
Cyclic	v5-L	1.55/ 14273	5.39/ 21530	17.19/ 32244
(original)	v5-A	0.90/**13041**	2.58/**16091**	**5.26**/304095
	v5-L	1.40/ 14067	5.07/ 20188	12.16/ 29110
Acyclic	v5-A	0.89/**13043**	2.58/**16093**	**5.26**/304097
(trans-	v7	1.69/ 14526	6.07/ 20255	12.34/ 27353
formed)	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 7.1: Run times (in seconds) and binary sizes (in bytes) of cyclic and acyclic Esterel programs compiled with the v5, v7, and CEC compiler.

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.

Table 7.2 compares the fastest code for our cyclic programs to the fastest code for the transformed acyclic programs. For each test program the relative reduction in runtime is listed.

Table 7.3 contains the compilation times for the different Esterel compilers to compile the various test programs. The v5 compiler for sorted equations code needs only little time to compile the acyclic versions of the test programs. In fact, it is among the fastest compilers in all three acyclic test cases. When this compiler is applied to cyclic programs, the compilation times are several times slower but within reasonable limits. The transformation times for the acyclic test programs (Table 7.4) are not included in Table 7.3, but even if we add the transformation times to the compilation times of the acyclic programs the picture will not change much.

When compiling for automaton code with the v5 compiler, then the compilation time is mostly independent of cyclic and acyclic properties of the compiled program. The compilation times are low for small programs with few states, but drastically higher for programs with many independent, parallel states. The CEC compiler is comparatively slow for small acyclic programs, but the compilation time does not rise that much for more complex programs. The v7 compiler behaves similarly.

As an indication of the cost of the transformation algorithm in terms of processing time and source code increase, Table 7.3 lists transformation times and program sizes before and after the transformation of the token ring arbiter with 3, 10, 50, 100, 500, and 1000 nodes. The size of the transformed code is nearly proportional with respect to the arbiter network size. The current transformation times show a sub-quadratic effort for the transformation.

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

Table 7.2: Relative run time reduction from the fastest cyclic version to the fastest version for the acyclic transformation, with $reduction = 100\% * \left(1 - \frac{min(T_{acyclic})}{min(T_{cyclic})}\right)$.

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

Table 7.3: Compiler run times for Esterel v5, v7, and CEC (in seconds).

7.2 Synthesizing Hardware

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

Table 7.5(a) compares the number of nodes synthesized. Considering the v5 compiler,

	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 transf.	1.53	1.56	1.58	1.59	1.60	1.61	1.39
Transformation time (sec.)	0.05	0.07	0.27	0.57	5.18	17.5	0.11

Table 7.4: Transformation times (in seconds) and resulting program sizes (in bytes) for token ring arbiters with 3 to 1000 nodes.

Variant	Compiler	Node Count			Latch Count		
		TR3	TR10	TR10p	TR3	TR10	TR10p
Cyclic	v5	112	357	759	10	31	55
Acyclic	v5	108	346	748	10	31	55
	v7	**52**	**171**	**351**	10	31	55
	CEC	146	468	756	**4**	**11**	**47**

(a)

Variant	Compiler	Unoptimized			Optimized		
		TR3	TR10	TR10p	TR3	TR10	TR10p
Cyclic	v5	208	745	1551	**82**	**266**	539
Acyclic	v5	197	645	1377	89	299	**524**
	v7	**108**	**360**	**702**	91	315	591
	CEC	221	725	1301	89	313	679

(b)

Table 7.5: Comparison of: (a) node and latch count for BLIF output, (b) sum-of-product (lits(sop)) count for BLIF output with and without optimization by SIS.

there is a noticeable reduction in the number of nodes generated for the Arbiter. When considering the synthesis results of v7 and CEC for the acyclic version of the Arbiter, v7 produces the best overall results, with the node count less than half of v5's synthesis results for the cyclic variants.

Table 7.5(b) compares the number of latches needed by the synthesization results. Here the CEC is able to reduce the number of latches considerably.

Table 7.5(c) compares the number of literals generated. The overall results are similar to the ones for the node count; the transformation has lowered the literal count for the arbiter.

Table 7.5(c) compares the number of literals which remain after a SIS [35] optimization.

Chapter 8

Assessment

The transformation presented in this thesis should resolve most cycles involving pure signals. However, some limitations exist, which are described in Section 8.1 and Section 8.2.

As already noted, we assume that the program to be transformed is constructive; Section 8.3 discusses the effects of the program transformation on non-constructive programs.

Finally, Section 8.4 elaborates on the consequences of the non-accessibility of the internal state in the Esterel language.

8.1 Scope of the Cycle Identification

The identification of cycles in Esterel programs as part of the cycle transformation is based on the detection of signal dependencies. Loops in these dependencies relate to cycles in the Esterel program. The algorithm presented in Section 3.4 maps the signal dependencies as implemented by Esterel compilers fairly well. Even unreachable code is left out of dependency calculations. The remaining point is the termination of parallel threads. A synchronizer circuit is used by the circuit semantics to derive the termination status from the parallel block. The dependency calculation presented here is based on sets of signal guards and does not evaluate exit codes. Therefore the cycle detection algorithm may consider some more signal dependencies than actually are identified by Esterel compilers. This limitations of the cycle detection may lead to unnecessary resolving of cycles. The program CYCLE_TRAP listed in Figure 3.6(c) on page 36 is an example for such a behavior of the transformation.

8.2 Scope of the Transformation Algorithm

The transformation algorithm in Chapter 4 is limited to the Esterel language with pure signals. Not supported are variables and valued signals. A possible extension of the transformation to valued signals is outlined in Chapter 5.

As noted in Section 4.1 cycles on local signals pose problems when replacement expressions carry the signal name out of the scope of the signal. To rule out such cases, the algorithm demands a lifting of local signal declarations in Step (2c). However, reincarnation problems are not addressed by the transformation, they must be resolved in a preprocessing step.

8.3 Transformation of Non-Constructive Programs

The transformation algorithm demands a constructive program as its input. This property is exploited in the proof for Step (6d) of the algorithm on page 61. A valid question is, what happens if the input program is *not* constructive?

For a non-constructive program P three basic cases exist, on transforming it into a program P':

Case 1: P is logically correct, but not constructive

Consider the following program P:

```
present S then
  emit S
end;
present S then
  present T else
    emit T
  end
end
```

This program contains two cycles: from S on itself and from T on itself. It is logically correct for S absent.

The state contexts of the emissions of S and T are

$$\textbf{emit}\ \mathsf{S} \Leftarrow \mathsf{S}$$
$$\textbf{emit}\ \mathsf{T} \Leftarrow \mathsf{S} \wedge \overline{\mathsf{T}}$$

The cycle on S is selected as the first cycle to resolve. By replacing S by True the following program is obtained:

```
present True then
  emit S
end;                          emit S;
present True then      ⇝      present T else
  present T else                emit T
    emit T                    end
  end
end
```

This program is not reactive anymore and therefore not logically correct.

On the other hand, replacing S by False yields:

```
present False then
  emit S
end;
present False then     ⇝      nothing;
  present T else              nothing
    emit T
  end
end
```

This result is logically correct and even constructive. But the selection between True or False changes the behavior of the transformed program considerably.

Case 2: P is not deterministic

Consider the following non-deterministic program P:

```
present A then
  emit A
end
```

The state context of the emission of A is

emit A $\Leftarrow$ A

The next step in the transformation algorithm is the replacement of the self-reference of signal A by True or False. Selecting True results in

```
present True then
  emit A               ⇝      emit A
end
```

Selecting False results in

```
present False then
  emit A               ⇝      nothing
end
```

Both cases remove the non-determinism from P' by statically selecting different branches of the present statement making it constructive.

Case 3: P is not reactive

Consider the following non-reactive program P:

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

The state context of the emission of A is

emit A $\Leftarrow \overline{\mathsf{A}}$

Replacement of signal A by True results in:

```
present False else
  emit A                ⤳        emit A
end
```

Replacement of signal A by False results in:

```
present True else
  emit A                ⤳        nothing
end
```

Both replacements remove the non-reactiveness from the program P' making it constructive. This behavior of P' is not covered by P, because P has no valid reaction and the transformation "invents" a behavior for P'.

In all three case the behavior of P is not preserved in the transformed program P'. The different behavior even depends on the selection between True or False on replacing the self-reference in replacement expressions. Therefore the insistence of the transformation algorithm on constructive input programs is legitimate.

8.4 Accessing the Program State

A central problem of the cycle treatment on the Esterel level is the unavailable internal execution state of the program at runtime. This state is represented by the currently active pause statements where execution starts in each instant. It is needed to express the emission of cycle signals as a function of the program state. Therefore the access to the program state is needed. To resolve this problem, emissions of artificial state signals are introduced into the program after each pause statement. These state signals are emitted as the first action in each instant and enable an access to the program state.

These added emissions of state signals have two basic problems: First these emissions are subject to suspensions by the kernel statement suspend. This leads to a dependency from the suspend condition to the emissions of the state signals. If the suspend condition contains cycle signals, then the cutting of the cyclic dependency will fail. As a remedy, Step (3a) of the algorithm substitutes the suspend statement. The functionality of the

suspend statement is emulated at the point of each pause statement by means of other statements. The removal of suspend solves the suspension problem at the cost of a growth in code size proportional to the number of affected pause statements.

The second problem with state signals lies in potential extensions of the transformation algorithm to non-kernel statements. Many of these statements include implicit pause statements which are not directly accessible (*e. g.*, await, sustain). This prevents the direct addition of state signals and makes it necessary to expand these non-kernel statements into kernel statements until the pause statements are exposed. If the pause states are directly accessible then efficient replacement expressions for non-kernel statements are conceivable.

Chapter 9

Conclusions and Future Work

This thesis 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 codes as well. Furthermore, the experiments showed that the code transformation proposed here can even improve code quality produced by compilers that can already handle cyclic programs.

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 delete some operations at run time. In a certain way, the transformation performs a partial evaluation of the given program.

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

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. Nevertheless a successful manual application of the transformation on a Lustre implementation of the Token Ring Arbiter is listed in Section A.3.

Regarding future work, the transformation algorithm spells out only how to handle cycles carried by pure signals. Chapter 5 outlines how to remove cycles involving a valued signal, but this still has to be generalized to handle variables as well.

The transformation algorithm in its current form includes a preprocessing step that expands most of the derived statements into kernel statements. The main reason for this is the limited access to pause statements buried in derived statements. As outlined in Section 4.3, this could be avoided if state signals would be directly accessible in Esterel.

There are also numerous optimizations possible, some of which were presented in Chapter 6. Some of these might be helpful for Esterel programs in general, not just as a post-processing step to the transformation proposed here. For example, the analysis from the transformation may also detect cases where tested signals are never emitted, *e. g.*, in the example given in Figure 4.12. This information cannot only be used to optimize the program, but also to point out possible programming errors.

Finally, as observed earlier, the concept of constructiveness is a fundamental building block for the transformation presented here. Constructiveness allows us to ultimately break a cycle by replacing the occurrence of a self-dependent signal in a replacement expression for that signal by an arbitrary value (true or false). However, to determine in the first place whether a program is constructive or not, the transformation proposed here might be employed to accelerate this analysis. As explained in Section 4.5, one may replace signal occurrences by expressions as computed by the algorithm (including possible self-references). This would substitute a generally computationally expensive iterative procedure, which is a classical approach to analyze constructiveness, by a more efficient analysis.

Appendix A

Example Transformations

This chapter contains some more Benchmarks for examinations on the size of compiled binaries. The v5 compiler is applied to the original cyclic and the transformed acyclic programs with compilation options to produce automaton code (v5-A) and circuit code (v5-L), resulting in four different binaries each. The CEC is applied to the acyclic programs in its default configuration (CEC) and with an option to use computed goto (CEC-g). This results in two different binaries for the CEC for each program.

The abbreviations of the compiler names and options are the same as in Section 7.1.

A.1 present, pause

```
module PAUSE_CYC:
input A, B;
output A_out, B_out;

  present A then
    emit B
  end;
  pause;
  present B then
    emit A
  end
||
  loop
    present A then emit A_out end;
    present B then emit B_out end;
    pause
  end

end module
```

```
module PAUSE_ACYC:
input A, B;
output A_out, B_out;

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
  ||
    loop
      present [A or A_] then emit A_out end;
      present [B or B_] then emit B_out end;
      pause; emit ST_2
    end
  ]
end signal

end module
```

Variant	Compiler	PAUSE
Cyclic (original)	v5-L	21534
	v5-A	18441
Acyclic (trans-formed)	v5-L	17782
	v5-A	18443
	CEC	**17617**
	CEC-g	17619

The noticeable feature in this example is the reduction in code size between the cyclic and transformed program for the v5 compiler using the circuit synthesis (v5-L).

```
module DRIVER_CYC:
input D;
input  Ain,   Bin;
output Aout, Bout;

loop
  present D then
    present
      [Ain or Aout]
    then
      emit Bout
    end
  else
    present
      [Bin or Bout]
    then
      emit Aout
    end
  end;
  pause
end
end module
```

```
module DRIVER_AYC:
input D;
input  Ain,   Bin;
output Aout, Bout;

signal ST_0, ST_1 in
  emit ST_0;
  loop
    present D then
      present [Ain or (ST_0 or ST_1) and not D
        and (Bin or (ST_0 or ST_1) and D)] then
        emit Bout
      end present
    else
      present [Bin or Bout] then
        emit Aout
      end present
    end present;
    pause;
    emit ST_1
  end loop
end signal

end module
```

Variant	Compiler	DRIVER
Cyclic (original)	v5-L	17849
	v5-A	18591
Acyclic (trans-formed)	v5-L	17851
	v5-A	18593
	CEC	**17623**
	CEC-g	17625

This transformation lacks some optimizations, *e. g.*, the state signal expressions yield always true. This inefficiency leads to a slightly worse result for the v5 compiler in the acyclic case. Nevertheless the CEC achieves a slightly better result on the acyclic program than the v5 compiler.

A.2 Termination of parallel Threads

```
module PAR_TERM_CYC:
output A, B;

[
  present A then
    nothing
  end
||
  nothing
];
emit B;
pause;
present B then
  emit A
end

end module
```

```
module PAR_TERM_ACYC:
input A_in, B_in;
output A, B;

signal PST_1, PST_2 in
signal ST_0, ST_1, ST_2, ST_3, ST_4, ST_5 in
  emit ST_0;
  signal PST_1, PST_2 in
    [
      trap PST in
        [
          present
            [ST_4 and (ST_0 or ST_3)
             and ((ST_0 or ST_1) and PST_2
             or (ST_0 or ST_5) and B_in) or
             (ST_0 or ST_5) and A_in]
          then
            nothing
          end present;
          loop emit PST_1; pause; emit ST_1 end
        ||
          nothing;
          loop emit PST_2; pause; emit ST_2 end
        ||
          loop
            present [PST_1 and PST_2] then
              exit PST
            end;
            pause; emit ST_3
          end loop
        ]
      end trap;
      emit B;
      pause; emit ST_4;
      present B then
        emit A
      end present
    ||
      loop
        present A_in then emit A end;
        present B_in then emit B end;
        pause; emit ST_5
      end loop
    ]
  end signal
end signal

end module
```

Variant	Compiler	PAR_TERM
Cyclic (original)	v5-L	17780
	v5-A	18424
Acyclic (trans-formed)	v5-L	17910
	v5-A	18426
	CEC	**17698**
	CEC-g	17700

The result of this benchmark is a little bit surprising, because the transformation is only able to handle parallel termination with a considerable growth in Esterel code size. Even under these circumstances was the CEC able to produce efficient code from the transformed acyclic program.

A.3 Implementing the Token Ring Arbiter in Lustre

```
node three_stations_cyc ( 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 ;
```

```
node three_stations_acyc ( 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 ;
```

The Lustre implementation of the Token Ring Arbiter listed here is rejected by Lustre compilers because of cyclic dependencies on streams pass1, pass2, and pass3. The replacement expression (4.26) (without state signals) is used 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 to Lustre programs.

Bibliography

[1] Albert Benveniste, Paul Caspi, Stephen A. Edwards, Nicolas Halbwachs, Paul Le Guernic, and Robert de Simone. The Synchronous Languages Twelve Years Later. In *Proceedings of the IEEE, Special Issue on Embedded Systems*, volume 91, pages 64–83, January 2003.

[2] Gérard Berry. Esterel on Hardware. *Philosophical Transactions of the Royal Society of London*, 339:87–104, 1992.

[3] Gérard Berry. *The Constructive Semantics of Pure Esterel*. Draft Book, 1999. ftp://ftp-sop.inria.fr/esterel/pub/papers/constructiveness3.ps.

[4] Gérard 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.

[5] Gérard Berry. The Foundations of Esterel. *Proof, Language and Interaction: Essays in Honour of Robin Milner*, 2000. Editors: G. Plotkin, C. Stirling and M. Tofte.

[6] Gérard Berry and Laurent Cosserat. The ESTEREL Synchronous Programming Language and its Mathematical Semantics. In *Seminar on Concurrency, Carnegie-Mellon University*, volume 197 of *LNCS*, pages 389–448. Springer-Verlag, 1984.

[7] Gérard Berry and Georges Gonthier. The Esterel Synchronous Programming Language: Design, Semantics, Implementation. *Science of Computer Programming*, 19(2):87–152, 1992. http://citeseer.nj.nec.com/berry92esterel.html.

[8] Gérard Berry and the Esterel Team. *The Esterel v5_91 System Manual*. INRIA, June 2000. http://www-sop.inria.fr/esterel.org/.

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

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

[11] Koen Claessen. Safety property verification of cyclic synchronous circuits. In *Electronic Notes in Theoretical Computer Science*, volume 88. Elsevier, July 2003. http://www.inrialpes.fr/pop-art/people/girault/Slap03/Final/claessen.pdf.

[12] Etienne Closse, Michel Poize, Jacques Pulou, Patrick Venier, and Daniel Weil. SAXO-RT: Interpreting Esterel semantic on a sequential execution structure. In Florence Maraninchi, Alain Girault, and Éric Rutten, editors, *Electronic Notes in Theoretical Computer Science*, volume 65. Elsevier, July 2002. `http://www.elsevier.com/gej-ng/31/29/23/117/53/34/65.5.010.pdf`.

[13] Ron Cytron, Jeanne Ferrante, Barry K. Rosen, Mark N. Wegman, and F. Kenneth Zadeck. Efficiently computing static single assignment form and the control dependence graph. *ACM Transactions on Programming Languages and Systems*, 13(4):451–490, October 1991. `http://citeseer.nj.nec.com/cytron91efficiently.html`.

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

[15] Stephen A. Edwards. Making Cyclic Circuits Acyclic. In *Proceedings of the Design Automation Conference*, pages 159–162. ACM Press, New York, NY, USA, June 2003.

[16] Stephen A. Edwards and Edward A. Lee. The Semantics and Execution of a Synchronous Block-Diagram Language. In *Science of Computer Programming*, volume 48. Elsevier, July 2003.

[17] Esterel Technologies. Company homepage. `http://www.esterel-technologies.com`.

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

[19] Free Software Foundation. GCC – The GNU Compiler Collection. `http://gcc.gnu.org/`.

[20] Nicolas Halbwachs, Paul Caspi, Pascal Raymond, and Daniel Pilaud. The synchronous data-flow programming language LUSTRE. *Proceedings of the IEEE*, 79(9):1305–1320, September 1991. `http://citeseer.nj.nec.com/halbwachs91synchronous.html`.

[21] John E. Hopcroft and Jeffrey D. Ullman. *Introduction to Automata Theory, Languages, and Computation*. Addison-Wesley, Reading, MA, 1979.

[22] Cornelis Huizing and Rob Gerth. Semantics of reactive systems in abstract time. In *Proceedings of the Real-Time: Theory in Practice, REX Workshop*, pages 291–314. Springer-Verlag, 1992.

[23] Xin Li, Marian Boldt, and Reinhard von Hanxleden. Compiling Esterel for a multi-threaded reactive processor. Technical Report 0603, Christian-Albrechts-Universität zu Kiel, Department of Computer Science, May 2006. Revised September 2006, `http://www.informatik.uni-kiel.de/reports/2006/0603.html`.

[24] Xin Li, Jan Lukoschus, Marian Boldt, Michael Harder, and Reinhard von Hanxleden. An Esterel Processor with Full Preemption Support and its Worst Case Reaction Time Analysis. In *Proceedings of the International Conference on Compilers, Architecture, and Synthesis for Embedded Systems (CASES)*, pages 225–236, New York, NY, USA, September 2005. ACM Press.

[25] Xin Li and Reinhard von Hanxleden. A concurrent reactive Esterel processor based on multi-threading. In *Proceedings of the 21st ACM Symposium on Applied Computing (SAC'06), Special Track Embedded Systems: Applications, Solutions, and Techniques*, Dijon, France, April 23–27 2006.

[26] Jan Lukoschus and Reinhard von Hanxleden. Removing cycles in Esterel programs. In Florence Maraninchi, Marc Pouzet, and Valérie Roy, editors, *International Workshop on Synchronous Languages, Applications and Programming (SLAP'05)*, Edinburgh, April 2005.

[27] Jan Lukoschus and Reinhard von Hanxleden. Removing cycles in Esterel programs. Technical Report 0502, Christian-Albrechts-Universität zu Kiel, Department of Computer Science, March 2005. `http://www.informatik.uni-kiel.de/en/ifi/research/technical-reports/`.

[28] Sharad Malik. Analysis of cyclic combinational circuits. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 13(7):950–956, July 1994.

[29] Paritosh Pandya. The saga of synchronous bus arbiter: On model checking quantitative timing properties of synchronous programs. In Florence Maraninchi, Alain Girault, and Éric Rutten, editors, *Electronic Notes in Theoretical Computer Science*, volume 65. Elsevier, 2002.

[30] Gordon D. Plotkin. A Structural Approach to Operational Semantics. Technical Report DAIMI FN-19, University of Aarhus, Denmark, 1981. `http://homepages.inf.ed.ac.uk/gdp/publications/SOS.ps`.

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

[32] Marc D. Riedel. *Cyclic Combinational Circuits*. PhD thesis, California Institute of Technology, Pasadena, California, USA, May 2004.

[33] Marc D. Riedel and Jehoshua Bruck. The Synthesis of Cyclic Combinational Circuits. In *Proceedings of the conference on Design automation (DAC)*, Anaheim, California, USA, June 2003.

[34] Robert Sedgewick. *Algorithms in C++, Part 5, Graph Algorithms.* Addison-Wesley, third edition, 2002.

[35] E. M. Sentovich, K. J. Singh, L. Lavagno, C. Moon, R. Murgai, A. Saldanha, H. Savoj, P. R. Stephan, R. K. Brayton, and A. Sangiovanni-Vincentelli. SIS: A System for Sequential Circuit Synthesis. Technical Report UCB/ERL M92/41, University of California at Berkeley, May 1992.

[36] Thomas R. Shiple, Gérard Berry, and Hervé Touati. Constructive Analysis of Cyclic Circuits. In *Proc. International Design and Test Conference ITDC 98, Paris, France*, March 1996.

[37] Olivier Tardieu. Goto and Concurrency—Introducing Safe Jumps in Esterel. In *Proceedings of Synchronous Languages, Applications, and Programming (SLAP)*, Barcelona, Spain, March 2004.

[38] Olivier Tardieu and Stephen A. Edwards. Approximate Reachability for Dead Code Elimination in Esterel*. In *In Proceedings of the Third International Symposium on Automated Technology for Verification and Analysis (ATVA)*, Taipei, Taiwan, October 2005.