Christian-Albrechts-Universität zu Kiel

Master Thesis

From Esterel to SCL

Karsten Rathlev

March 31, 2015

Department of Computer Science
Real-Time and Embedded Systems Group

Prof. Dr. Reinhard von Hanxleden

Advised by:
Dipl.-Inf. Steven Smyth
Dipl.-Inf. Christian Motika

Eidesstattliche Erklärung

Hiermit erkläre ich an Eides statt, dass ich die vorliegende Arbeit selbstständig verfasst und keine anderen als die angegebenen Hilfsmittel verwendet habe.

Kiel,

Abstract

Esterel is a synchronous language targeting deterministic concurrency by making heavy restrictions on the program structure. Even if the sequential ordering of statements reveals how the developer wants them to be executed, programs may be refused by the compiler. Languages such as the Sequentially Constructive Language (SCL) ease these restrictions by considering the sequential order of statements and allowing concurrent modifications of variables as long as a deterministic behavior can be guaranteed. The Sequentially Constructive Model of Computation (SC MoC) used by SCL is a conservative extension to the synchronous MoC, thus it is possible to translate Esterel to SCL. However, as SCL is designed as a minimal language, i.e., only a small amount of language features is provided, the rich compendium of Esterel statements have to be expressed by using the small set of SCL statements. As the translation rules should be kept simple and thus comprehensible, the resulting SCL code may contain redundant statements and therefore may be optimized. In this thesis, the transformation rules and the subsequent optimizations are presented and evaluated. Additionally, it is described how this transformation can be used to compile Esterel within the KIELER framework by using the existent compilation chain from SCL to target code and how the SC MoC can be applied to Esterel.

Keywords Esterel, synchronous languages, constructiveness, sequential constructiveness, SCL, SCG, KIELER

Contents

1 Introduction **1**

1.1 The Synchronous Model of Computation . . . 1

1.1.1 Esterel . . . 2

1.1.2 Constructiveness . . . 5

1.2 The Sequentially Constructive Model of Computation . . . 6

1.2.1 Sequentially Constructive Language . . . 7

1.2.2 Sequentially Constructive Graph . . . 8

1.2.3 Compilation of SCL/SCG . . . 8

1.3 KIELER . . . 12

1.4 Problem Statement . . . 14

1.5 Outline . . . 14

2 Related Work **17**

2.1 From Esterel to SyncCharts . . . 17

2.2 Compilation of SCCharts . . . 18

2.3 INRIA Esterel Compiler . . . 19

2.3.1 Circuit Approach . . . 19

2.3.2 Automaton Approach . . . 20

2.4 Columbia Esterel Compiler . . . 20

2.4.1 Graph Code Intermediate Format . . . 20

2.4.2 Program Dependence Graph Code Generation . . . 22

2.4.3 Dynamic List Code Generation . . . 24

2.4.4 Virtual Machine Code Generation . . . 24

2.5 Freesterel . . . 24

3 Used Technologies **27**

3.1 Eclipse . . . 27

3.1.1 Plug-ins . . . 27

3.1.2 Eclipse Modelling Framework . . . 28

3.1.3 Xtext . . . 28

3.1.4 Xtend . . . 30

3.2 KIELER . . . 30

3.2.1 KIEM . . . 31

3.2.2 KART . . . 32

3.2.3 KIELER Compiler . . . 33

3.2.4 KIELER Model View . . . 33

4 Esterel to SCL Transformation 35
4.1 Definitions . . . 35
4.2 Transformation Rules . . . 36
4.2.1 Emit . . . 36
4.2.2 Signals . . . 36
4.2.3 Valued Signals . . . 41
4.2.4 Parallel . . . 43
4.2.5 Loop . . . 44
4.2.6 Trap . . . 45
4.2.7 Await . . . 49
4.2.8 Strong Suspend . . . 50
4.2.9 Strong Abort . . . 52
4.2.10 Weak Abort . . . 56
4.2.11 Local Variable Declarations . . . 58
4.2.12 Halt . . . 58
4.2.13 Module Instantiation . . . 58
4.2.14 Count Delays . . . 59
4.2.15 Additional Language Features . . . 59
4.3 Step-Wise Transformation of ABRO . . . 60
4.4 Optimization . . . 65
4.4.1 Optimized Signal Transformation . . . 65
4.4.2 SCL Optimization . . . 66
4.4.3 Cross Dependencies of Optimizations . . . 67
4.4.4 Optimization Example . . . 69

5 Implementation 71
5.1 SCL Implementation . . . 71
5.1.1 Expression Language . . . 71
5.1.2 Meta-Model . . . 72
5.1.3 Grammar . . . 74
5.1.4 Formatting . . . 76
5.1.5 Code Validation . . . 76
5.1.6 Scoping . . . 78
5.2 Esterel Implementation . . . 80
5.2.1 Expression Language . . . 80
5.2.2 Grammar . . . 80
5.3 Esterel to SCL Transformation . . . 81
5.3.1 Main Transformation Class . . . 81
5.3.2 Interface Transformation . . . 84
5.3.3 Expression Transformation . . . 84
5.3.4 Extensions . . . 85
5.4 SCL Optimizations . . . 86

5.5 KART Regression Testing . . . 87

6 Evaluation and Experimental Results 89
6.1 Test Set-Up . . . 89
6.2 Compile Performance . . . 89
6.2.1 Esterel to SCL . . . 89
6.2.2 Esterel to Target Code . . . 90
6.3 Scalability . . . 90
6.3.1 SCL Code Size . . . 91
6.3.2 Optimizations . . . 91
6.3.3 Target Code . . . 93
6.4 Target Code Execution Time . . . 93
6.5 Sequentially Constructive Esterel . . . 95
6.5.1 Sequential Ordering of Statements . . . 95
6.5.2 The Initialize-Update-Read Paradigm . . . 96
6.5.3 The ABO Example . . . 98
6.5.4 Results . . . 98

7 Conclusion 101
7.1 Summary . . . 101
7.2 Future Work . . . 101
7.2.1 Weak Suspend . . . 101
7.2.2 Transformation of Tasks . . . 104
7.2.3 Step-Wise Transformation . . . 104
7.2.4 SCL to Esterel Transformation . . . 105
7.3 Transformation of Further Synchronous Languages to SCL . . . 108

List of Figures

1.1 The ABRO Example . . . 4
1.2 Mapping between SCL and SCG [vHMA+13]. . . . 8
1.3 ABO example in SCL and the corresponding SCG. . . . 9
1.4 The compilation chain for SCGs. . . . 9
1.5 The dependencies in SCGs are indicated by differently colored arrows [Smy13]. 10
1.6 The ABO program with basic blocks marked in purple. . . . 11
1.7 The sequentialized SCG corresponding to ABO. . . . 12
1.8 Overview of the KIELER Components (KIELER Documentation) . . . 13
1.9 Sequentially Constructive Esterel . . . 14

2.1 ABRO transformation from Esterel to SyncChart [Rüe11]. . . . 17
2.2 An implementation of ABO as an SCChart. . . . 18
2.3 Esterel program and the corresponding circuit [Ber02]. . . . 20
2.4 An Esterel program and the corresponding GRC [EZ07]. . . . 21
2.5 Program Dependence Graph [EZ07] . . . 23

3.1 The movie database meta-model in the tree editor and in the diagram editor. 28
3.2 Code editor generated by Xtext. . . . 29
3.3 Schematic overview of KIEM. . . . 31
3.4 Simulation via KIEM in Eclipse. . . . 32
3.5 Kieler Compiler View for the Esterel compilation via SCL. . . . 33
3.6 Kieler Compiler View and Model View . . . 34

4.1 The SCG corresponding to the example for the output/local signal transformation in Listing 4.6. . . . 39
4.2 The SCG corresponding to the program in Listing 4.7 with overlaid dependencies. . . . 40
4.3 Even though the valued emit statement is split to two SCL statements, no unintended behavior is introduced. . . . 42
4.4 Exception handling blocks in Esterel are transformed to concurrent threads executing the do-block if the trap was activated. . . . 46
4.5 The SCG corresponding to the SCL program in Listing 4.19a. A potentially instantaneous loop is generated by a trap nested within a loop in the source Esterel program in Listing 4.18b. . . . 47
4.6 await Cases Transformation to SCG . . . 50
4.7 The strong delayed abort example in Listing 4.26 illustrated as an SCG (signal transformation omitted). . . . 53

4.8 Step-wise transformation of ABRO (a). 61
4.8 Step-wise transformation of ABRO (b). 62
4.8 Step-wise transformation of ABRO (c). 63
4.8 Step-wise transformation of ABRO (d). 64
4.9 Cross Dependencies of SCL Optimizations 68
4.10 SCL Optimizations without Cross Dependencies 68

5.1 The annotations Meta-Model 71
5.2 The KExpressions Meta-Model 73
5.3 The SCL Meta-Model 74
5.4 Diagram showing the communication between the different components of the Esterel to SCL transformation implementation. 82
5.5 The test cases are simulated via SCL and the CEC. The traces are validated against each other and create the regression testing suite together with the corresponding test cases. 88
5.6 For the regression testing, every Esterel program in the testing suite is simulated with the same inputs as in the validated trace. The resulting trace is compared to the validated one. 88

6.1 Results for the performance analysis of the optimized and the unoptimized transformation. 91
6.2 Results for the compiler performance comparison of the compilation via SCL and the CEC in KIELER. 92
6.3 Amount of statements after transforming Esterel programs to SCL with optimizations and without. 93
6.4 Lines of code generated by the compilation to C via SCL and the CEC. . . 94
6.5 Comparison of the performance for compiled Esterel programs via SCL and by the CEC. 94
6.6 Extract of the SCG for the IUR Program 96

Listings

1.1 Transformation of the await Statement to Kernel Statements 3
1.2 Transformation of the loop each Statement to Kernel Statements 3
1.3 ABRO with Kernel Statements 5
1.4 Esterel programs that are not B-constructive. 6
1.5 Esterel program that is B-constructive. 6
1.6 Sequentially constructive program that is invalid in Esterel. 7

3.1 The movie database grammar in Xtext. 29
3.2 Adding a movie to the database with Xtend. 30
3.3 Simple transformation method in Xtend (a) and compiled to Java (b). . . 31

4.1 emit Transformation 36
4.2 sustain Transformation 36
4.3 Input Signal Transformation 36
4.4 Output Signal Transformation 37
4.5 Local Signal Transformation 37
4.6 Output/Local Signal Transformation Example 38
4.7 Reincarnation Example in Esterel 38
4.8 Valued Output Signal Transformation 41
4.9 Valued emit Transformation 41
4.10 Definition of a resolution function in Esterel and in SCL. f_e is the neutral element of the function f. 43
4.11 emit Transformation with Resolution Function 43
4.12 Parallel Transformation 43
4.13 loop Transformation 44
4.14 loop each is transformed to a combination of loop, abort and halt, which is subsequently transformed to SCL. 44
4.15 The every do statement can be expressed in Esterel by a loop and a preceding await. 44
4.16 repeat n times Transformation 45
4.17 trap Transformation 45
4.18 Rejection of potentially instantaneous loops in the CEC. 47
4.19 By removing gotos originating by the trap transformation in the initial tick, a larger set of Esterel programs can be scheduled after the transformation to SCL. 48
4.20 Delayed await Transformation 49
4.21 Immediate await Transformation 49

4.22 Strong Delayed suspend Transformation 50
4.23 Strong Immediate suspend Transformation 51
4.24 Strong Delayed suspend Example (Signal Transformation Omitted) 51
4.25 Strong Delayed abort Transformation 52
4.26 Example for the strong delayed abort transformation (signal transformation omitted). 52
4.27 Without the additional flag, the strong delayed abort transformation may not behave as aspected (signal transformation omitted). 53
4.28 Strong Immediate abort Transformation 54
4.29 Example for the transformation of abort with cases (signal transformation omitted). 55
4.30 Weak Delayed abort Transformation 56
4.31 Weak Immediate abort Transformation 56
4.32 The transformation rule for weak abortion may produce unnecessary potentially instantaneous loops. With the refined transformation, this is avoided in many cases. 57
4.33 Local Variable Transformation 58
4.34 halt Transformation 58
4.35 run Transformation 58
4.36 Example for the Count Delay Transformation 59
4.38 Generation of redundant code when transforming Esterel to SCL. 65
4.39 Removing Redundant goto Optimization 66
4.40 Removing Double Jumps Optimization 67
4.41 Removing Dead Code 67
4.42 The label optimization may produce dead code. 68
4.43 Removing dead code may produce unused labels. 68
4.44 Step-wise appliance of the SCL optimizations. 69

5.1 An SCL program starts with the keyoword module followed by variable declarations. 75
5.2 Extract of the SCL grammar implementation in Xtext. 75
5.3 Automatic formatting for SCL should make sure that there is no white-space before and a line-break after each semicolon. 76
5.4 An SCL program before and after invoking auto-formatting. 77
5.5 This static validation method for SCL should make sure that the target label of a goto statement is in the same scope. 77
5.6 The global variable a should be shadowed in the statement scope since a is redefined. Thus, the output variable a is not set to false whilst b is. 78
5.7 An implementation of a scope provider for SCL. 79
5.8 Left-recursion can be avoided by using the list assignment mechanism of Xtext. 81

5.9 Two methods can be called by KiCo to invoke the transformation either with or without optimization. 82
5.10 Xtend Implementation of the loop Transformation 83
5.11 Xtend Implementation of gotoj 85
5.12 Adding a goto to a StatementSequence 86
5.13 Definition of the addGoto Extension Method in Xtend 86
5.14 Xtend implementation of the dead code elimination. 87

6.1 The transformation of a simple invalid Esterel program to a valid SCL program. 95
6.2 The Transformation of IUR from Esterel to SCL 97
6.3 The Transformation of ABO from Esterel to SCL 99

7.1 Tick boundaries may occur depending on input variables. 102
7.2 A transformation rule for weak suspend that does not work for programs containing concurrency. 102
7.3 Esterel programs containing concurrent statements can not be translated by the rule given in Listing 7.2 since ticks may also start in parallel running threads. 103
7.4 For weak suspend, also at entry points of threads additional gotos are necessary. 103
7.5 Definition and Execution of a Task in Esterel 104
7.6 Applied in the wrong order, the Esterel to SCL transformation may produce wrong code. 105
7.7 Overlapping Conditional Loops 107
7.8 Removing Unproper Nested Loops 107
7.9 Tranformation of a Backward Jump 108

List of Tables

6.1 Extract of the test cases for the evaluation with a short description. . . . 90
6.2 Amount of statements in Esterel programs and the resulting SCL programs. 92

Abbreviations

ANTLR Another Tool for Language Recognition

API Application Programming Interface

AST Abstract Syntax Tree

BAL Bytecode Assembly Language

BLIF Berkeley Logic Interchange Format

BB Basic Block

CEC Columbia Esterel Compiler

DSL Domain Specific Language

EBNF Extended Backus-Naur Form

EMF Eclipse Modeling Framework

ESO Esterel Simulator Output

GRC GRaph Code

IDE Integrated Development Environment

IEEE Institute of Electrical and Electronics Engineers

INRIA French Institute for Research in Computer Science and Automation

KART KIELER Automated Regression Testing

KIEL Kiel Integrated Environment for Layout

KIELER Kiel Integrated Environment for Layout Eclipse Rich Client

KIEM KIELER Execution Manager

KiCo Kieler Compiler

MoC Model of Computation

M2M Model-to-Model

OAW Open Architecture Ware

PDG Program Dependence Graph

RCP Rich Client Platform

RTSYS Real-Time and Embedded Systems

SC Synchronous C

SC MoC Sequentially Constructive Model of Computation

SCG Sequentially Constructive Graph

SCL Sequentially Constructive Language

SLIC Single-Pass Language-Driven Incremental Compilation

SC Synchronous C

UI User Interface

VHDL Very High Speed Integrated Circuit Hardware Description Language

XML Extensible Markup Language

Chapter 1

Introduction

Embedded, real-time systems can be found in many daily used objects. They are input-driven and reactive, i.e., they have to be capable to continuously react to the environment [Ber00]. Typical examples are cars, planes, medical equipment and printers.

Whilst it is tolerable if a printer is malfunctioning, the systems of a plane have to work properly in every situation. Such systems are called *safety-critical* [MP95].

A main concern of reactive systems, especially if they are safety-critical, is predictability and thus determinism. That is, every time a particular input sequence is entered, the same output sequence has to be produced [Ber00].

As stated in *The Problem with Threads* by E. A. Lee [Lee06], classical programming languages do not provide deterministic concurrency and therefore have to be used with care. Even systems running without faulty behavior for a long time may reveal programming mistakes. To overcome this issue and guarantee deterministic behavior, the family of *synchronous languages* makes heavy restrictions on the program structure. Among others, *sequentially constructive languages* try to ease these restrictions without compromising on predictability [vHMA+13]. This trade-off between deterministic behavior and expressiveness is an ongoing challenge.

1.1 The Synchronous Model of Computation

A Model of Computation (MoC) describes *the rules that govern the interaction, communication, and control flow of a set of components* [BLL+05]. It defines the allowed operations that may be used in computations.

In the *synchronous* MoC, time is divided into discrete *ticks*. The execution done within a tick is called a *macro step*, which can further be divided into finitely many *micro steps*. This *multiform notion of time* abstracts from physical time and only considers the ordering of events [BCE+03].

Further, the synchronous MoC considers that calculations take zero time, thus it is presumed that results are present at the same time as the inputs are read. As this is impossible, *zero time* can be read as *before the next execution*. This assumption is called the *Synchrony Hypothesis* [Ber91].

A frequently used construct in synchronous languages is that of *signals*. Signals have a well-defined status within each tick and must respect the *signal coherence law* [Tar05]. Signals are *absent* by default and *present* only if they are emitted.

1.1.1 Esterel

Esterel is a programming language developed by G. Berry *et al.* since the early 1980s and is based on the synchronous MoC [Ber00]. It is an imperative, textual language designed for the development of complex, reactive systems. Currently, the most relevant versions are Esterel v5 and Esterel v7, whereby Esterel v7 brings in some new statements and extensions [Ber05].

Esterel v5 Kernel Statements

The statements of Esterel v5 can be classified into *kernel statements* and *derived statements*. The kernel statements of Esterel v5 are the following [Ber93]:

nothing
: Do nothing; can be seen as a dummy statement.

pause
: Stop execution for the current tick, in other words `pause` is a tick boundary.

p**;** q
: First execute p and subsequently q.

$p \mid\mid q$
: Execute p and q in parallel. The statement returns when both sub-statements have returned.

emit s
: Make the signal s present in the current tick.

present s **then** p **else** q **end**
: If s is present in the current tick, then p is executed, otherwise q.

loop p **end**
: Repeat the execution of p infinitely often.

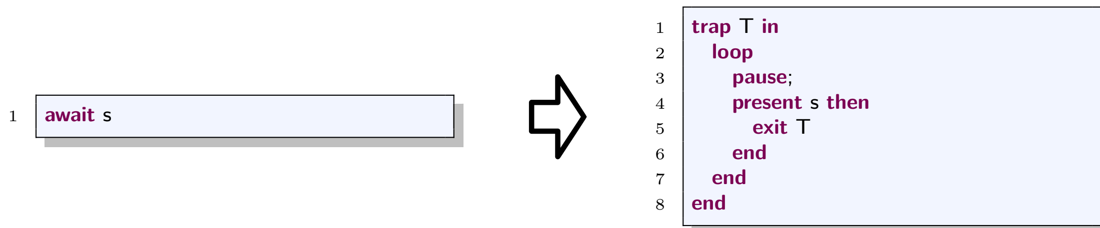
signal s **in** p **end**
: A local signal declaration. If s is already defined in higher hierarchical levels, it is shadowed.

suspend p **when** s
: Suspend p if s is present, i.e., p does not continue execution in a tick when s is present.

trap T **in** p **end**
: Execute p until an `exit` statement is reached that invokes T.

exit T
: Jump to the end of the surrounding trap statement for which T is defined. Concurrently executed threads may continue the execution to the next tick boundary and terminate afterwards. When nested, the outermost trap statement takes precedence.

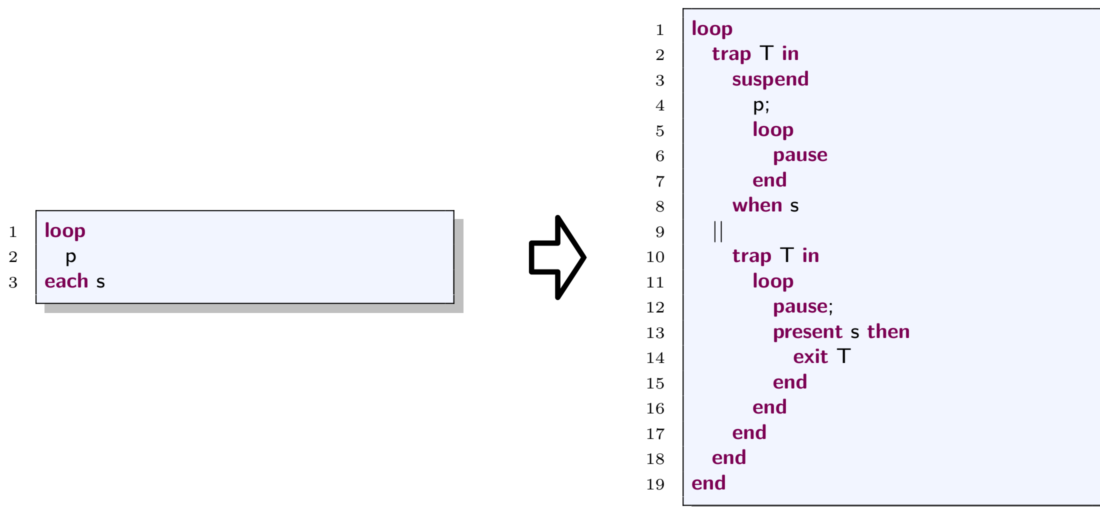
The derived statements can be seen as syntactic sugar. They can all be translated to kernel statements [BG92, Ber00]. As an example, Listing 1.1 shows the transformation of the await and Listing 1.2 of the loop each statement. Until a specific event occurs after the initial tick, await interrupts the execution. loop each starts the execution of its body every time a specific event occurs. Therefore, the execution of its body may be restarted in a preemptive manner.



```
1 await s
```

```
1 trap T in
2   loop
3     pause;
4     present s then
5       exit T
6     end
7   end
8 end
```

Listing 1.1. Transformation of the await Statement to Kernel Statements

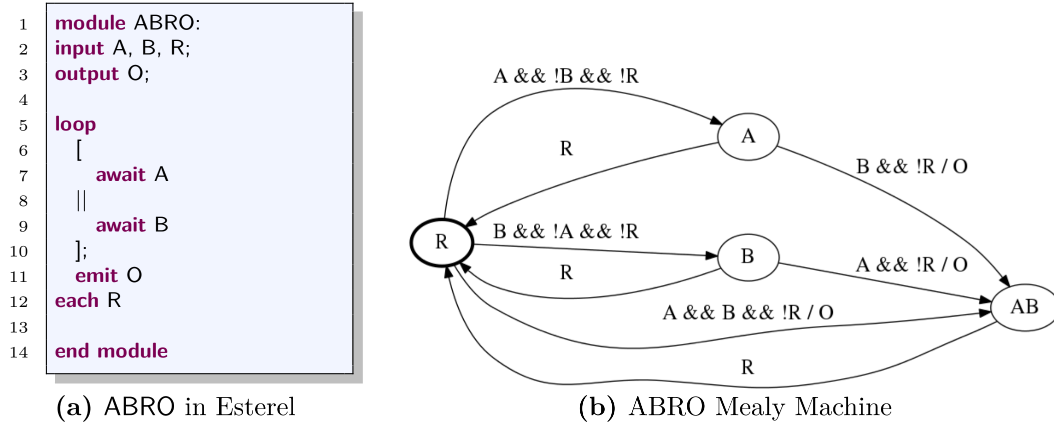


```
1 loop
2   p
3 each s
```

```
1  loop
2    trap T in
3      suspend
4        p;
5        loop
6          pause
7        end
8      when s
9    ||
10     trap T in
11       loop
12         pause;
13         present s then
14           exit T
15         end
16       end
17     end
18   end
19 end
```

Listing 1.2. Transformation of the loop each Statement to Kernel Statements

Example for an Esterel Program



(a) ABRO in Esterel (b) ABRO Mealy Machine

Figure 1.1. The ABRO Example

Figure 1.1 shows ABRO, the *Hello World* of the synchronous MoC, in Esterel and as a mealy machine. In the ABRO program, O is emitted when A and B have been present at least once. Every time R is present the behavior is reset, thus O is not emitted, even if A and B are both present. The advantage of derived statements can be seen in Listing 1.3. Using only kernel statements, the program is by far not as comprehensible as with derived ones.

Host Language Integration in Esterel

Esterel is compiled to a host language, which might be, among others, C, Java or the Very High Speed Integrated Circuit Hardware Description Language (VHDL). The main concern of an Esterel compiler is handling the concurrency. That means, all thread interaction is handled statically at compile time for guaranteeing deterministic behavior.

An advantage of using a host language is that its features can be used. For instance, Esterel itself only offers a small set of data types, whilst in other languages more complex and also user defined types are possible. Esterel allows to import these types and makes it even possible to use functions defined in the host language.

Esterel v7

Esterel v7 introduces some extensions and new features. Besides technical enhancements as arrays and multi-clocks, also a new statement, weak suspend, is added [Est05]. When weak suspend is triggered in a tick, the control flow remains where it is, but the current tick's micro steps are executed nevertheless. This can be seen as a *last wish* before suspension takes place. Another view would be that the current tick is executed normally, but afterwards, the control returns to the state it was at the start of this tick.

```
module ABRO:
input A, B, R;
output O;

loop
  abort
    [
      trap T in
        loop
          pause;
          present A then
            exit T
          end
        end
      end
    ||
      trap T in
        loop
          pause;
          present B then
            exit T
          end
        end
      end
    ];
    emit O;
    halt
  when R
end

end module
```

Listing 1.3. ABRO with Kernel Statements

Weak suspension results in increased expressiveness in comparison to Esterel v5, hence it is not possible to translate it to Esterel v5 kernel statements. This can be derived from its different nature. All other preemptive statements have a fixed point where to continue or, in the case of strong suspension, rather just do not execute any micro steps. For weak suspension, it has to be remembered at which points a tick started, which can not be determined statically, since tick boundaries may occur dependent on input signals.

1.1.2 Constructiveness

The *constructive semantics* of Esterel, used in the official compiler, were introduced by G. Berry in 1999. They are defined by the *constructive coherence laws* [Ber02]:

- A signal is declared present if and only if it must be emitted.
- A signal is declared absent if and only if it cannot be emitted.

When reasoning which signals must be present and which signals cannot be present suffice in establishing a presence state for all output signals, a program is considered

```
1 present s then
2   nothing
3 else
4   emit s
5 end
```

(a)

```
1 present s then
2   emit s
3 else
4   nothing
5 end
```

(b)

Listing 1.4. Esterel programs that are not B-constructive.

```
1 present s then
2   pause
3 else
4   pause;
5   emit s
6 end
```

Listing 1.5. Esterel program that is B-constructive.

constructive, or in reference to the creator *B-constructive*. The official Esterel compiler uses this notion to accept or reject Esterel programs.

For example, the Esterel program in Listing 1.4a is not B-constructive. Besides no assumption about the presence state of s would be logical, there is no chance to reason about the presence state of s by the constructive coherence laws.

The Esterel program in Listing 1.4b is not B-constructive, even though assuming s as present is the only logical signal configuration. Without information about s, none of the conditional branches has to be taken, but also there is no justification for assuming that one of them cannot be taken.

In contrast, the program in Listing 1.5 is B-constructive. Even though there is nothing the program must do if s is unknown, the program cannot emit s in the first tick. By propagating facts, s is determined to be absent in the first tick and therefore s is emitted in the second tick.

1.2 The Sequentially Constructive Model of Computation

The Sequentially Constructive Model of Computation (SC MoC) is a conservative extension of the synchronous MoC [vHMA+13]. *Conservative* means that all valid programs in the synchronous MoC are also valid in the SC MoC.

The main intention of the SC MoC is reducing the restrictions of classical synchronous languages without introducing non-determinate concurrency. In the SC MoC, also the order of actions matters, which allows variables to have different values within one tick. For instance, the program in Listing 1.4a is correct in terms of the SC MoC.

Also the restrictions for the concurrent modification of a variable are softened. In the

```
1 [
2   s = false;
3   if s then
4     t = true
5   end
6 ||
7   s = s | true
8 ]
```

Listing 1.6. Sequentially constructive program that is invalid in Esterel.

SC MoC, variable modifications are executed in an *initialize-update-read* order. Firstly, variable initialization, i.e., absolute writes, is done. Secondly, updates are applied, which are relative writes. At last, reading accesses are scheduled. For example, the program in Listing 1.6 sets `s` to `false` first by the absolute write in line 2. Afterwards, `s` is set to `true` by the relative write in the other thread in line 7 and finally, `s` is read in line 3. Thus, `t` is set to `true`. Further, confluent writes are also admissible. These are writes that are concurrent, but lead to the same variable value regardless of the execution order.

1.2.1 Sequentially Constructive Language

The Sequentially Constructive Language (SCL) is designed as a minimal, sequentially constructive language proposed by R. v. Hanxleden *et al.* [vHMA⁺13]. It only consists of the following eight instructions for statements s, variables x and expressions e:

$x = e$
: Assign the value of expression e to variable x.

s_1**;** s_2
: First execute s_1 and subsequently execute s_2.

if e **then** s_1 **else** s_2
: Execute s_1 if e holds and s_2 otherwise.

l**:** s
: Tag statement s with label l.

goto l
: Continue execution at label l which has to be in the same thread as the `goto` statement.

fork s_1 **par** s_2 **join**
: Execute s_1 and s_2 in parallel.

pause
: Stop execution for the current tick.

{d_1; d_2; . . . ; d_n; s}
Execute s with respect to the local variable declarations $d_1 \ldots d_n$. That is, outer variable declarations may be shadowed if they are redefined.

1.2.2 Sequentially Constructive Graph

The Sequentially Constructive Graph (SCG) is a directed labeled graph and the graphical representation of SCL [vHDM+14]. Nodes correspond to the program statements and edges to the sequential control flow.

Figure 1.2 shows the mapping between SCL and SCG components. Rectangles are statements, triangles and inverted triangles represent forks and joins. Tick boundaries are illustrated by a surface and a depth node connected by a dotted line. Every thread is entered with an enter node and left with an exit node.

Figure 1.3 shows the ABO example, which can be seen as the *Hello World* of sequential constructiveness. If A is present, B and O1 are set to true. As B is true, O1 is set to true again in the other thread, which is confluent with the former write. Afterwards, the join can be executed and O1 is set to false whilst O2 is set to true.

1.2.3 Compilation of SCL/SCG

As well as for other languages, various compiling approaches for SCL/SCGs exist. A compiler implementation using a *netlist* based approach was introduced by S. Smyth in 2013 [Smy13] and is in ongoing development. The compilation approach is related to a common compilation technique for synchronous languages. The objective is to sequentialize concurrency and therefore create a *tick-function*. That is, a sequential function that calculates the reaction for each tick. The code generation is done in several steps, as illustrated in Figure 1.4.

Dependency Analysis

Two statements are said to be *concurrent*, if they have a least common ancestor fork node. As stated earlier, concurrent execution is done in an initialize-update-read manner and therefore the dependencies of statements are calculated in this stage. These dependencies are used in the upcoming steps, as they affect the order in which the statements have to

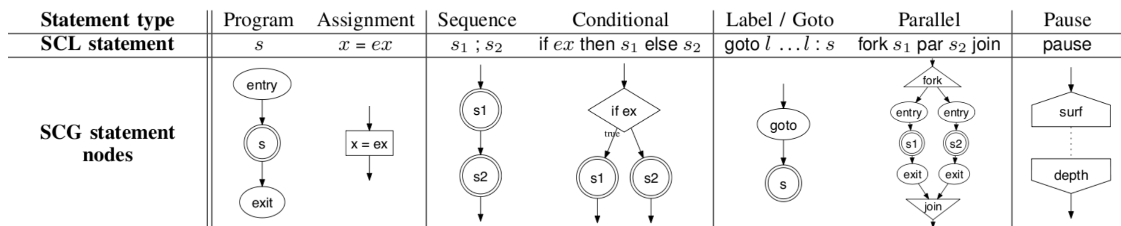
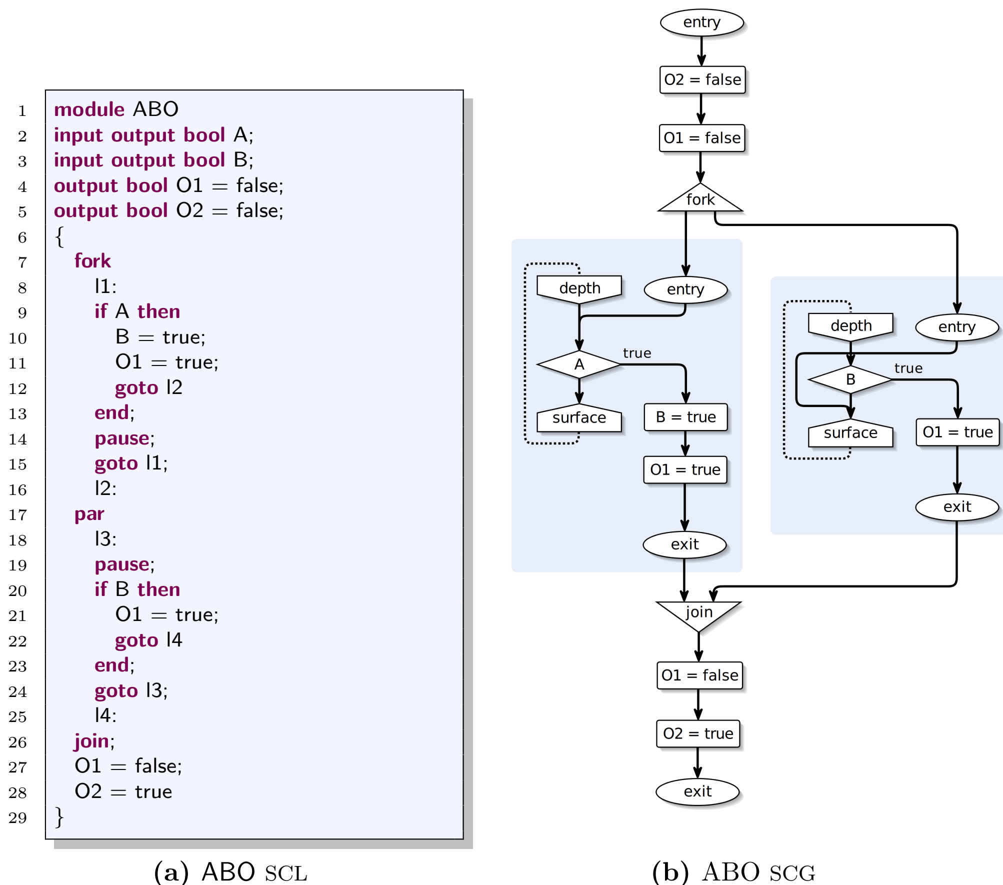


Figure 1.2. Mapping between SCL and SCG [vHMA+13].

be scheduled. Figure 1.5 illustrates the four different concurrent dependencies and their illustration.

The *write-write* dependency in Figure 1.5a causes a conflict and cannot be scheduled as long as the writes are not confluent. The *initialize-update* dependency in Figure 1.5b states that the absolute write has to be scheduled before the relative write. Figure 1.5c shows the *initialize-read* dependency, which states that the absolute write is scheduled prior to the reading of the variable. The *update-read* dependency in Figure 1.5d declares that the relative write is scheduled before the variable is read.



(a) ABO SCL (b) ABO SCG

Figure 1.3. ABO example in SCL and the corresponding SCG.

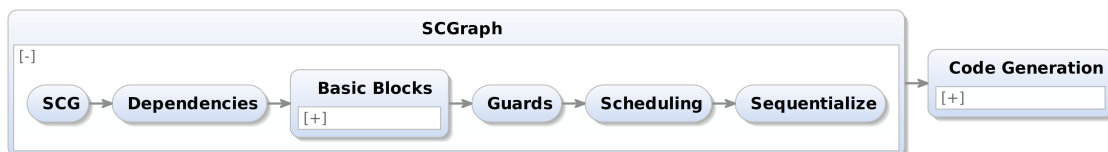
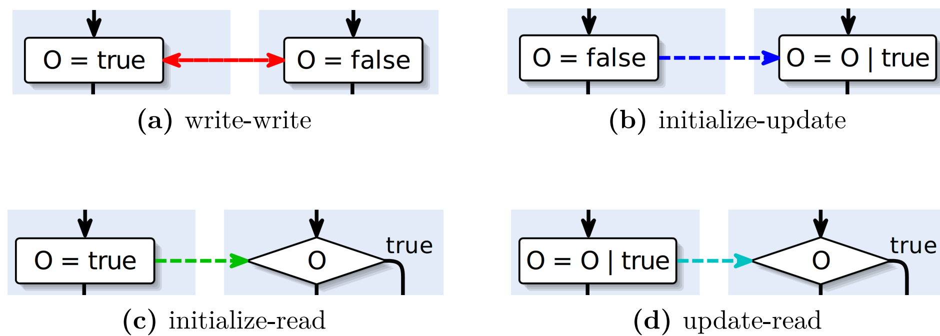


Figure 1.4. The compilation chain for SCGs.



(a) write-write (b) initialize-update

(c) initialize-read (d) update-read

Figure 1.5. The dependencies in SCGs are indicated by differently colored arrows [Smy13].

Basic Block Analysis

The partitioning of a program into Basic Blocks (BBs) is a common technique used in compilers [ASU86]. BBs are program parts that can be executed monolithically, i.e., if the first statement of a BB is executed also the following statements are executed in the given order. Each node is included in only one BB. In Figure 1.6, the calculated basic blocks for the ABO SCG are given.

Guards

Guards determine which BBs may be executed at which time. After a BB, other guards may be activated by guard expressions. The guard for the first BB, thus the starting point of a program, is initially set to true instantiating the execution.

Scheduling

In the scheduling phase, the previously calculated dependencies are used to further divide the BBs. When calculating a schedule, i.e., determining in which order the program parts should be executed, dependencies between concurrent statements have to be considered. For this purpose, BBs have to be structured further into scheduling blocks. This is realized by splitting BBs at incoming dependency edges since this indicates that another statement should be executed previously.

Sequentialization

In this phase, the concurrent SCG is transformed to a sequential one. That is, an SCG without pauses, i.e., surface and depth nodes, and without parallel threads. This sequential SCG can be seen as a tick function that is executed once in each tick. The calculated guards can be interpreted as a state in which the program is and depending on them, statements are executed or not. As an example, Figure 1.7 illustrates the result of transforming ABO into a sequential SCG.

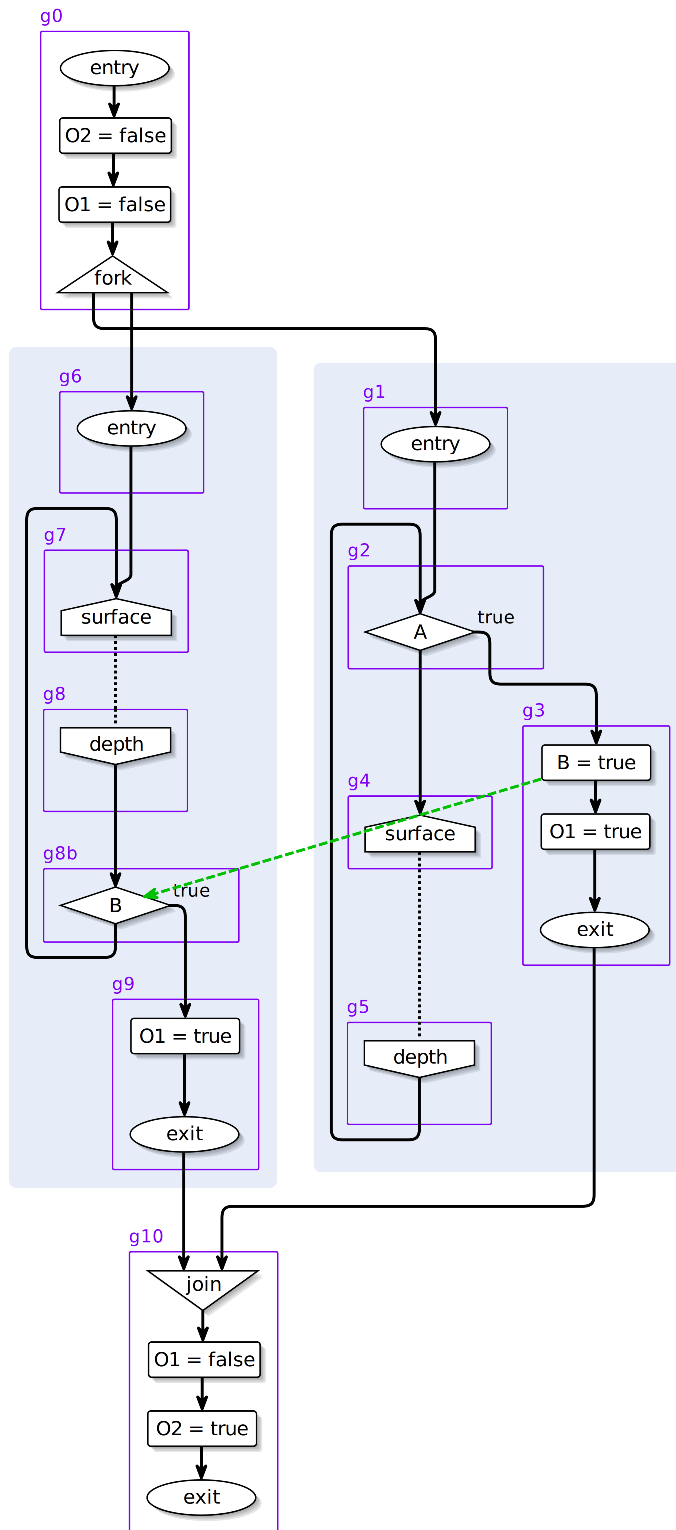


Figure 1.6. The ABO program with basic blocks marked in purple.

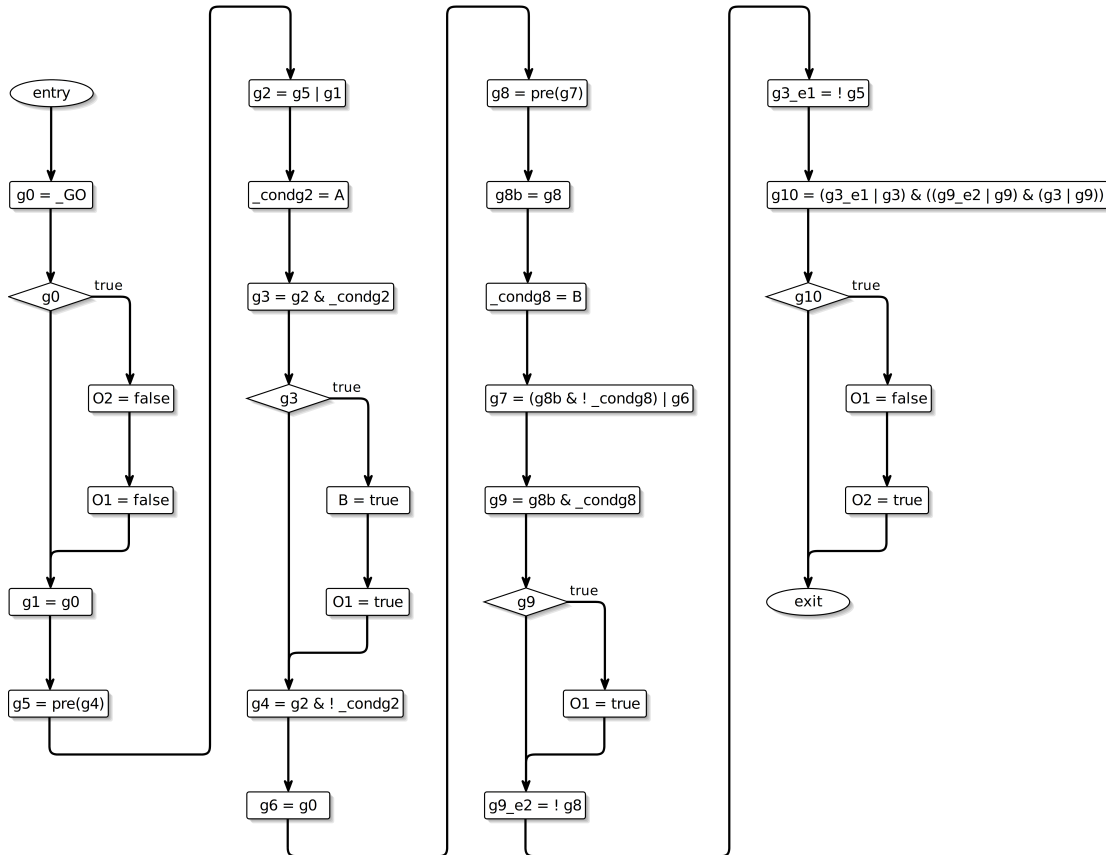


Figure 1.7. The sequentialized SCG corresponding to ABO.

Code Generation

Having a sequential SCG, the code can be generated straightforwardly. Depending on the target language, various intermediate steps may be necessary. For example, the compilation to Synchronous C (SC), a language that adopts the synchronous principles to C, may be used to generate executable code [vH09].

1.3 KIELER

The Kiel Integrated Environment for Layout Eclipse Rich Client (KIELER) is an ongoing research project by the Real-Time and Embedded Systems group at the Christian-Albrechts-Universität zu Kiel, Germany, which mainly targets on graphical, model-based design. It is structured as a collection of Eclipse plug-ins, allowing a modular and easily extendable integration into the Eclipse IDE. The components of KIELER illustrated in Figure 1.8 are the following:

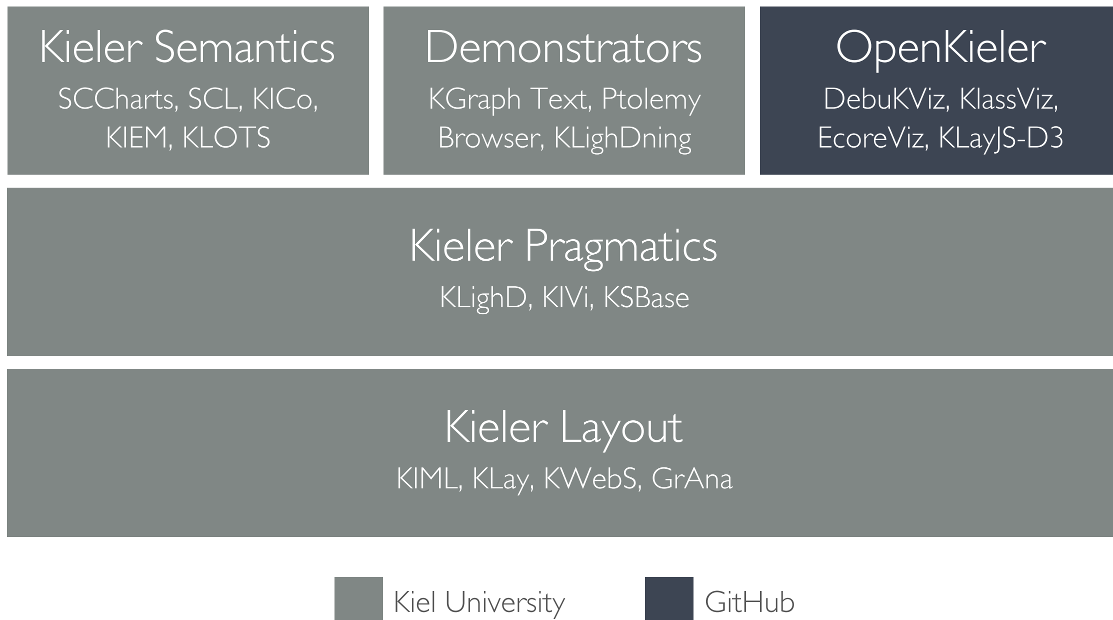


Figure 1.8. Overview of the KIELER Components (KIELER Documentation)

Layout
In this component, everything related to automatic layout is classified.

Pragmatics
Part of the *pragmatics* is everything that is related to the practical aspects of handling graphical models.

Semantics
Everything related to the execution or simulation of models is classified as *semantics*.

Demonstrators
These are tools used for testing and demonstrating the different components of KIELER.

OpenKieler
Some popular KIELER projects are hosted on GitHub and are opened to external contributions.

1.4 Problem Statement

Since SCL relies on the SC MoC, which is intended to be a conservative extension to the synchronous MoC [vHMA+13], it should be possible to translate an Esterel program to SCL. Such a compiler allows to compile Esterel programs within KIELER with the existing compilation chain for SCL/SCG. Further, it illustrates that SCL has at least the same expressiveness as Esterel.

Since SCL is a minimal language, such a transformation would also testify that there is a smaller set than the Esterel kernel statements that can express Esterel programs.

Additionally, it is explored how SCL and the representation as an SCG can be used as an intermediate format for the compilation of synchronous languages.

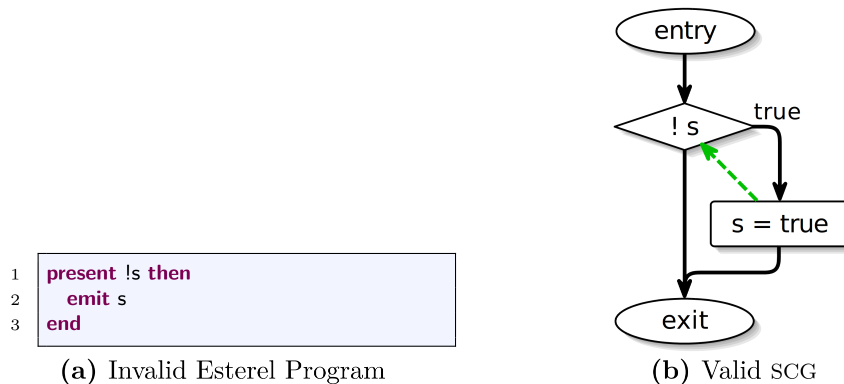
Moreover, it is evaluated how some incorrect Esterel programs are translated to valid SCL programs and the SC MoC can be adopted to Esterel. As an example, Figure 1.9 illustrates how an invalid Esterel program is transformed into a valid SCG.

1.5 Outline

In Chapter 2, related work is discussed. Besides discussing the transformation from Esterel to *SyncCharts*, a graphical, synchronous language, the transformation of *SCCharts* to SCL/SCG is examined. Finally, other compilation approaches for Esterel are presented.

Chapter 3 introduces the used technologies. As the implementation evolved from this thesis is integrated into KIELER, which is a collection of Eclipse plug-ins, the general plug-in mechanism of Eclipse is discussed. Further, utilized frameworks, as the Eclipse Modeling Framework (EMF), Xtext and Xtend, are presented. In addition, KIELER and the components used for this thesis are examined.

Chapter 4 is the leading part of this document. Here, the transformation rules are presented and justified. Optimizations of the transformation rules and the resulting SCL code are discussed, followed by the step-wise transformation of ABRO as presented in Section 1.1.




```
1 present !s then
2   emit s
3 end
```

(a) Invalid Esterel Program

(b) Valid SCG

Figure 1.9. Sequentially Constructive Esterel

The implementation of the Esterel to SCL transformation and its characteristics are presented in Chapter 5. Besides the transformation and optimizations, also contributions to the SCL editor and the grammar are described.

In Chapter 6, the transformation and its implementation are evaluated. It is described how the performance compares to a popular existing Esterel compiler. Additionally, observations on how the SC MoC can be adopted to Esterel are examined.

Finally, the results are summarized and some possible future work is proposed in Chapter 7. This covers initial thoughts on the transformation of the Esterel v7 statement `weak suspend` to SCL, as well as a proof of concept for the transformation of B-constructive SCL to Esterel, *inter alia*.

Chapter 2

Related Work

At first, the transformation of Esterel to *SyncCharts*, a graphical, synchronous language is presented and the similarities and differences of the requirements for the transformation in comparison to the transformation to SCL are discussed.

Afterwards, an approach for the compilation of *SCCharts* is examined, a graphical, sequentially constructive language developed by R. v. Hanxleden *et al.* [MSvH14]. SCCharts are also transformed to SCL/SCG and subsequently to target code.

Finally, other Esterel compilers are compared to the approach taken in this thesis and similarities, as well as differences, are distinguished.

2.1 From Esterel to SyncCharts

SyncCharts are Statecharts enriched by synchronous semantics and can be seen as a graphical representation of Esterel. They are fully compatible as they have been defined via a translation to Esterel [And03, PTvH06].

The transformation from Esterel to SyncCharts has been implemented into the Kiel Integrated Environment for Layout (KIEL) framework, a predecessor of KIELER, in 2006

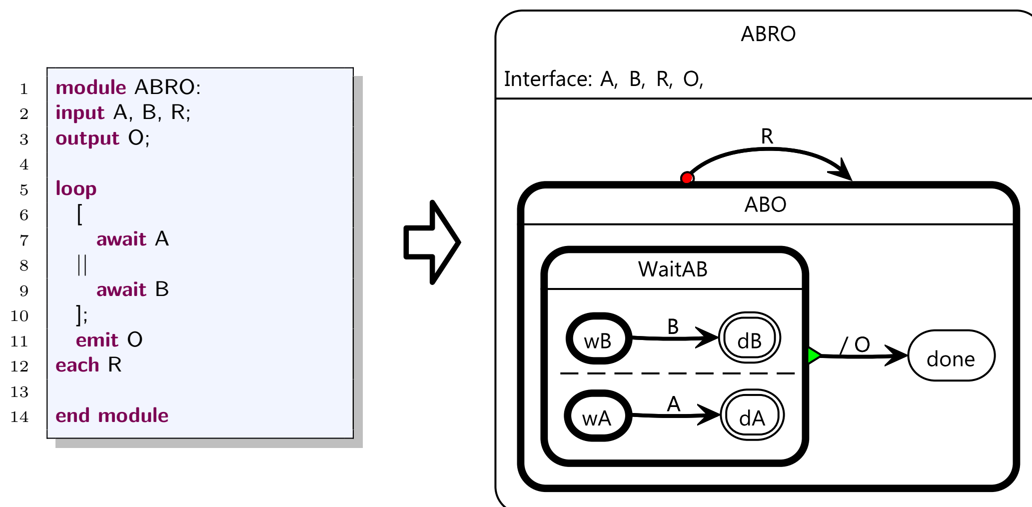


Figure 2.1. ABRO transformation from Esterel to SyncChart [Rüe11].

by L. Kühl [Küh06]. An implementation for KIELER was introduced by U. Rüegg in 2011 [Rüe11].

As an example, Figure 2.1 shows the transformation of ABRO, as presented in Section 1.1. In the SyncChart on the right, states surrounded by bold lines are *initial*, double encircled states are *final*. Continuous lines without any further marks are *delayed* transitions. In the tick the state from that they emerge is entered, they are not visible.

The green triangle at the start of the transition from WaitAB to done indicates a *normal termination*. That is, if the state WaitAB terminates, which happens if A and B are each present at least once after the initial tick, this transition is taken and O is emitted. As the state done has no outgoing transitions and is not final, the control remains there. The red dot at the self-transition of the ABO state represents *strong abortion*. If R is present, this transition preempts the execution of the ABO state and restarts it.

SyncCharts provide a broad compendium of language features that correspond to those of Esterel. Therefore, the transformation is reasonably straightforward. In contrast, SCL only provides a minimal set of statements. Accordingly, the transformation rules are more complex. For example, whilst SyncCharts have abort transitions with the same behavior as abortion in Esterel, in SCL, this has to be expressed by means of conditionals and gotos, *inter alia*.

2.2 Compilation of SCCharts

SCCharts can be seen as SyncCharts enriched by the SC MoC. As an example, Figure 2.2 shows how ABO, as presented in Section 1.2.2, can be realized as an SCChart [vHDM+14].

To compile SCCharts, at first *extended* language features, which can be seen as syntactic sugar, are transformed to *core* language constructs. Afterwards, the core SCChart is normalized, i.e., transformed to a structure that can be mapped to SCG. Finally, the resulting SCG is compiled to target code.

Similar to the compilation of SCCharts, Esterel can be compiled to target code by a prior transformation to SCL/SCG, which can be used as an intermediate format for

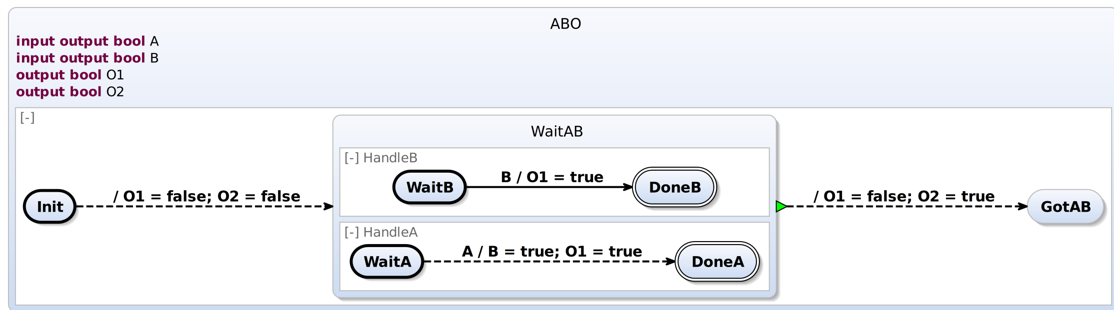


Figure 2.2. An implementation of ABO as an SCChart.

compiling synchronous and sequentially constructive languages. Since SCCharts are based upon SyncCharts, the graphical representation of Esterel, some challenges for the transformation are the same.

A difference between the transformation of SCCharts to SCL/SCG and Esterel to SCL/SCG is the different source MoC. However, as the SC MoC is a conservative extension to the synchronous MoC, this should not be a main concern.

In contrast to the compilation of SCCharts, the approach to transform Esterel to SCL presented in this thesis does not rely on removing all syntactic sugar and only provide transformation methods for kernel statements. Even though there are statements that are transformed to kernel statements and subsequently to SCL, for most language features a specific transformation rule is given. This aims on reducing overhead in both, compilation run-time and code size, as well as on giving a direct connection between Esterel and SCL.

2.3 INRIA Esterel Compiler

The INRIA[1] Esterel compiler is the official compiler for Esterel. The development started at the French Institute for Research in Computer Science and Automation (INRIA) under the supervision of G. Berry. The compiler can compile Esterel programs to either hardware or software. It is possible to do so either by using the *circuit-based* or the *automaton-based* approach, both having the same run-time interface [Bt00]. The INRIA Esterel compiler for Esterel v5 is stable since the late 1990s [Ber00], for the Esterel v7 compiler, a standardization proposal for the Institute of Electrical and Electronics Engineers (IEEE) was released in 2005 [Est05].

2.3.1 Circuit Approach

As it is relatively simple to simulate circuits, this compilation can also be used when targeting software. Since the generation of the netlist code, as well as its size, is approximately linear with respect to the source code size, this approach scales well and is suitable for large programs. However, when the resulting circuit is simulated, also irrelevant results are computed, leading to slow and inefficient code.

Figure 2.3 illustrates the derivation of a circuit from a given Esterel program. The equations derived from the Esterel program and represented by the circuit are

$$O1 = I \land O2$$
$$O2 = O1 \land \neg I.$$

[1] http://www.inria.fr/en

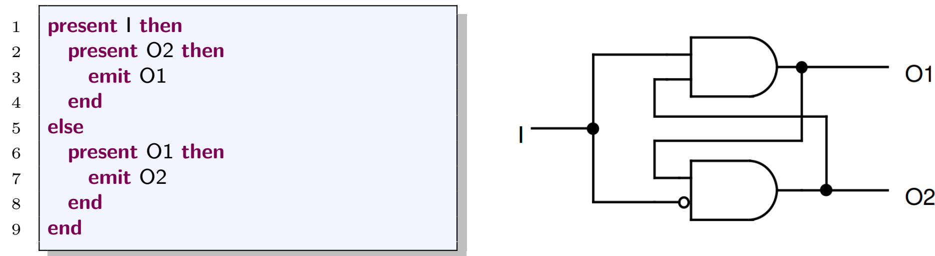


Figure 2.3. Esterel program and the corresponding circuit [Ber02].

As stated in Section 1.2, also SCGs can be compiled via netlists. In contrast to the INRIA Esterel compiler, the SCG code synthesis additionally has to consider the sequential ordering of statements.

2.3.2 Automaton Approach

Instead of creating a circuit, this approach compiles to an explicit final state machine. The resulting code is faster than the code generated by the circuit approach, but its drawback is the size, which may be exponentially bigger than the source code [Bt00]. Accordingly, this approach is only suitable for small programs.

2.4 Columbia Esterel Compiler

The Columbia Esterel Compiler (CEC)[2] is an open source Esterel compiler developed by S. A. Edwards *et al.* at the Columbia University since 2001 [PBEB07]. It is intended to be an academic project to experiment with different code generation techniques. Therefore, the CEC has a modular design to allow substitution of modules and a good extensibility. For this purpose, it consists of several executables which communicate via a format based on the Extensible Markup Language (XML).

The CEC can compile Esterel to C, Verilog and the Berkeley Logic Interchange Format (BLIF) and provides three different code generation methods which are presented briefly in this section. In-depth descriptions can be found in the article *Code Generation in the Columbia Esterel Compiler* by S. A. Edwards *et al.* [EZ07].

2.4.1 Graph Code Intermediate Format

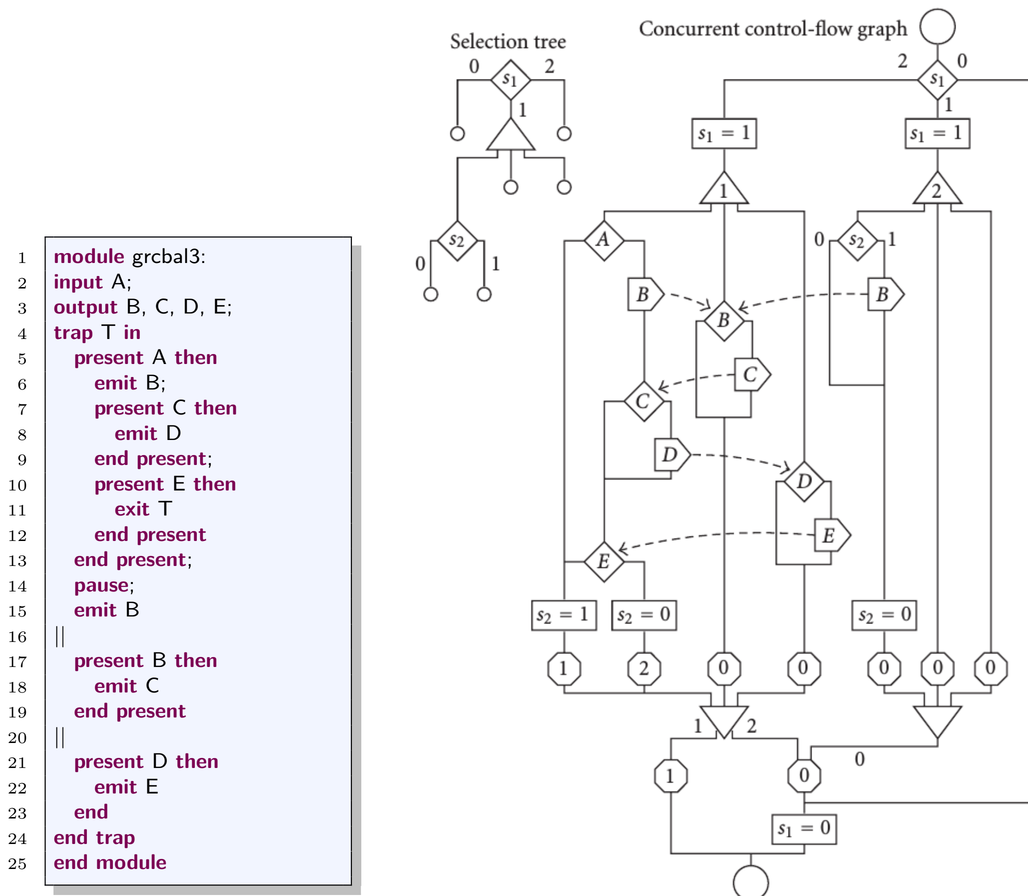
In order to generate code, a GRaph Code (GRC) representation, initially proposed by D. Potop-Butucaru [PB02] and slightly modified, is used as an intermediate format. Figure 2.4 shows an Esterel program and the associated GRC. The small tree in the upper-left part of Figure 2.4b is called *Selection Tree*. It is generated straightforwardly by

[2] http://www.cs.columbia.edu/~sedwards/cec

the structure of the Esterel program and represents the state in which it is at a specific time.

The control-flow graph, which is the bigger one in Figure 2.4b, is the essence of the code generation process. It is executed once in every tick and can be seen as the graphical representation of a tick function that determines the reaction in a specified state or tick with respect to inputs. Dashed lines indicate dependencies, e.g., between the emission of a signal and a conditional statement. Triangles and inverted triangles represent forks and joins, diamonds are predicates. Pointed rectangles indicate signal emission and octagons termination. Octagons include a number determining the termination code: 0 for normal termination, 1 for pausing and 2 when an exit statement was triggered.

The construction of the GRC can be implemented inductively on the parsed Abstract Syntax Tree (AST) and is similar to creating circuits from Esterel code. Simulating a GRC is reasonably simple. Depending on the state given by the Selection Tree, the control-flow



```
1  module grcbal3:
2  input A;
3  output B, C, D, E;
4  trap T in
5    present A then
6      emit B;
7      present C then
8        emit D
9      end present;
10     present E then
11       exit T
12     end present
13   end present;
14   pause;
15   emit B
16 ||
17   present B then
18     emit C
19   end present
20 ||
21   present D then
22     emit E
23   end
24 end trap
25 end module
```


(a) Esterel (b) GRC

Figure 2.4. An Esterel program and the corresponding GRC [EZ07].

graph can be executed, starting from the root node with respect to the signals and values. Along the way, the selection tree is updated appropriately.

At this stage, some optimizations are applied, e.g., dead and redundant code elimination. The substantial similarities in appearance to SCGs are not incidental. The approach taken by the CEC is related to the transformation of Esterel to SCL/SCG as an intermediate format and subsequent code generation.

In contrast to GRC, SCGs permit arbitrary control-flow, for example, loops, and have additional types of dependencies. Further, SCGs do not use a structure such as the Selection Tree to keep state. They utilize registers to save the state between tick boundaries [vHDM+14].

Example

In the example control-flow graph in Figure 2.4b, at first the left-most path would be taken, causing s_1 to be set to 1. As the presence state of B depends on A, the condition statement is executed. If A is present, after some thread communication, E is emitted. Subsequently, s_2 is set to 0 and the thread terminates with exit code 2, whilst the other two threads terminate with exit code 0. After the join node, the highest exit code alternative is taken, in this case the arc labeled with 2. This causes the termination code of the root thread to be 0, thus signaling termination. s_1 is set to 0 and the tick ends. In any subsequent tick, at the first choice node, the right-most path is taken. Thus, the program does nothing, as it has already terminated.

Absence of A in the initial tick would result in the termination code 1 for the first pass of the control-flow graph, hence indicating a pause. In the second pass, at the first choice node, the path in the middle would be taken. Since s_2 would have been set to 1 in the previous tick, E would be emitted and the program terminates afterwards.

2.4.2 Program Dependence Graph Code Generation

For this approach, a variant of the Program Dependence Graph (PDG) developed by J. Ferrante *et al.* [FOW87] is used. A PDG is a directed, acyclic graph where nodes correspond to program instructions and edges to dependencies between them. *Statement nodes* represent statements in the source program, for example assignments. They have no outgoing arcs, as they do not affect the control-flow. *Predicate nodes* represent conditionals. As the subsequent control-flow depends on the predicate, they have two outgoing edges. The last class of nodes are *fork nodes*, which represent concurrency and therefore pass control to their ancestor nodes. A characteristic of PDGs is that there are no unnecessary dependencies. Two additional rules have to be satisfied for a valid PDG:

Predicate Least Common Ancestor Rule If a node has more than one incoming arc, the least common ancestor node of every two distinct predecessor nodes have to be a predicate node. This requirement arises from the fact that if it would be a

fork node, there could be more than one active path to the node. This could lead to a multiple execution of the same statement.

No Post-Dominance Rule For two nodes n_1 and n_2 holds that if n_1 is an ancestor node of n_2, there exists a node n_3 which is an descendant of n_1, and n_2 is not on the path between them. If there was not such a node n_3, unnecessary dependencies exist.

The transformation is done by an algorithm proposed by R. Cytron *et al.* [CFR+91], which was slightly modified to handle concurrency. Figure 2.5 shows the PDG corresponding to the GRC shown in Figure 2.4b. Again, the dashed lines represent data dependencies, e.g., before a variable can be read, it has to be written.

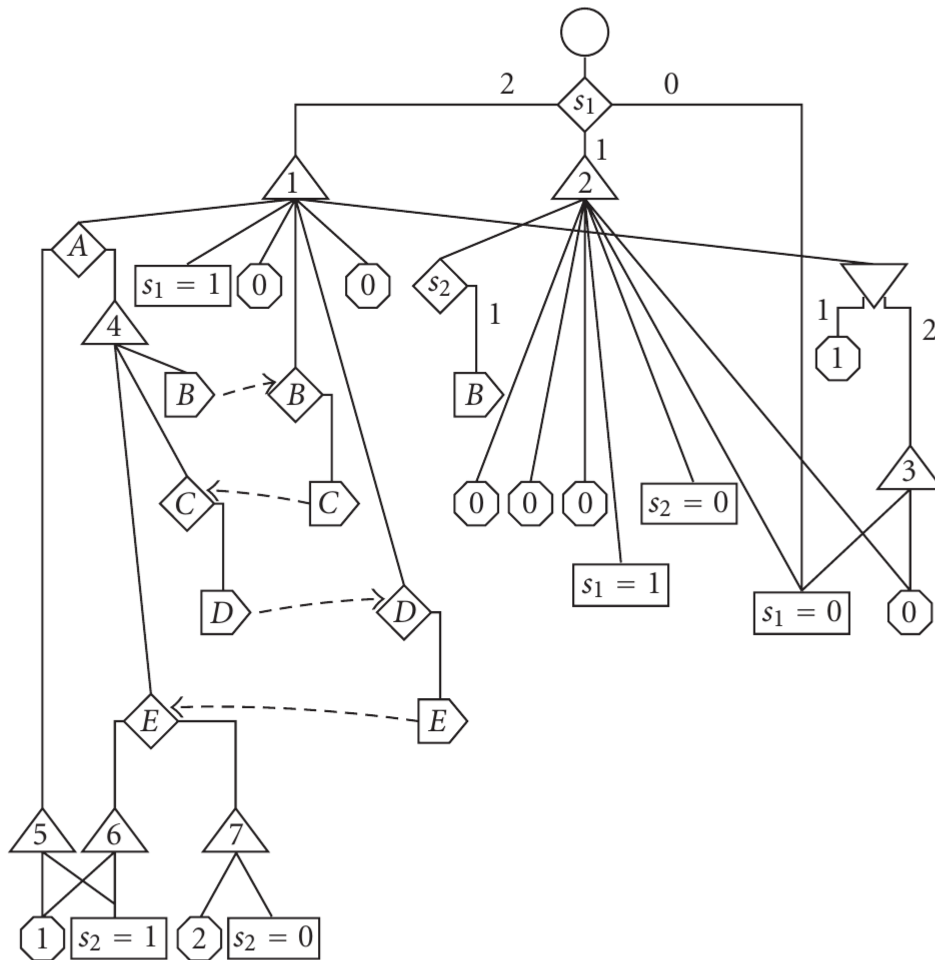


Figure 2.5. Program Dependence Graph [EZ07]

Transforming the GRC into the PDG illustrated in Figure 2.5 may not seem very intuitive at first, as even though all unnecessary control dependencies are eliminated, much more concurrency is produced. However, this procedure allows a more flexible reordering of statements for reducing scheduling overhead by building larger atomic blocks.

As concurrent statements have to be executed in the right order to ensure that writes are executed before reads, in the restructuring phase *guards* are added. Guards ensure that the control is handed to the right statement when the PDG is sequentialized. The actual sequentialization is done by an approach presented by B. Simons and J. Ferrante [SF93]. For sequential PDGs, several code generation techniques are known. In the CEC, an approach by S. A. Edwards is used [Edw02].

2.4.3 Dynamic List Code Generation

In contrast to, for instance, the code generation via PDGs or netlists, in the dynamic list approach, code that does not need to execute should not do so. The GRC is analyzed for clusters, in this case, a collection of statements that can be executed atomically. These clusters are further assigned to specific *levels*, where a level contains clusters that can be scheduled in any order. A linked list for each level specifies which clusters in the current tick should be executed. A cluster's code contains information about which cluster should be scheduled afterwards.

2.4.4 Virtual Machine Code Generation

The virtual machine approach targets on generating short, compact code. However, it does not consider performance very much. Its core is a virtual machine preserving similar semantics as Esterel and is based upon the so called Bytecode Assembly Language (BAL). The idea derives from the differences of Esterel to languages such as C or Java, which do not provide preemption or concurrency in the same way as Esterel.

The virtual machine provides registers for general purposes, signal presence states and completion codes. For arithmetic operations a stack is provided. The BAL code is derived from the – slightly modified and topologically sorted – GRC by depth-first search. Finally, the CEC generates C code around the virtual machine to make it executable without further work.

2.5 Freesterel

As the INRIA Esterel compilers are not open source and the CEC does not support Esterel v7, T. Coadou and B. Ferrero of INRIA started developing Freesterel[3] in 2010. Their aim is a free, open source Esterel v7 compiler. Therefore, they try to develop a clone of the genuine Esterel compiler by making use of existing literature and reverse

[3] http://freesterel.sourceforge.net

engineering the compiler. The status of the project is unclear as there are currently neither publications nor source code available.

This approach is directly connected to the INRIA compiler and aims on developing a complete compilation chain from Esterel to target code. In contrast, the compilation via a transformation to SCL makes use of an existing compilation chain for SCL to target code.

Chapter 3

Used Technologies

In this chapter, the technologies used for the implementation are presented to allow a better comprehensibility of the following chapters. Firstly, Eclipse and the utilized frameworks are presented. Afterwards, the components of KIELER that have been used are examined.

3.1 Eclipse

Eclipse[1] is a popular open source programming tool developed by the *Eclipse Foundation*. Starting as an Integrated Development Environment (IDE) for Java, Eclipse benefits from its easy extensibility and is nowadays used in many different domains [dRB06]. It is especially well suited to build arbitrary IDEs.

The Eclipse Rich Client Platform (RCP) is a minimal set of Eclipse modules that are necessary to build a platform application with a User Interface (UI).

3.1.1 Plug-ins

The core of Eclipse is its plug-in architecture [Bol03]. Eclipse plug-ins are intended to be small and only carry little functionality. Larger projects are commonly split over several plug-ins to allow a modular and extensible project structure.

The communication between the plug-ins is regulated by *extension points*, which can be defined by a plug-in and used by others. For example, Eclipse itself provides extension points for menu contributions that can be extended by custom plug-ins to add new functionality to the IDE. The interaction is defined in a *manifest*, that is, an XML file, which is contained in each plug-in.

For specifying which plug-ins should be loaded when starting an Eclipse instance, a *product* is used. It provides basic information about the application and allows to customize the appearance of Eclipse. Therefore, Eclipse provides an extension point, which has to be extended by a product.

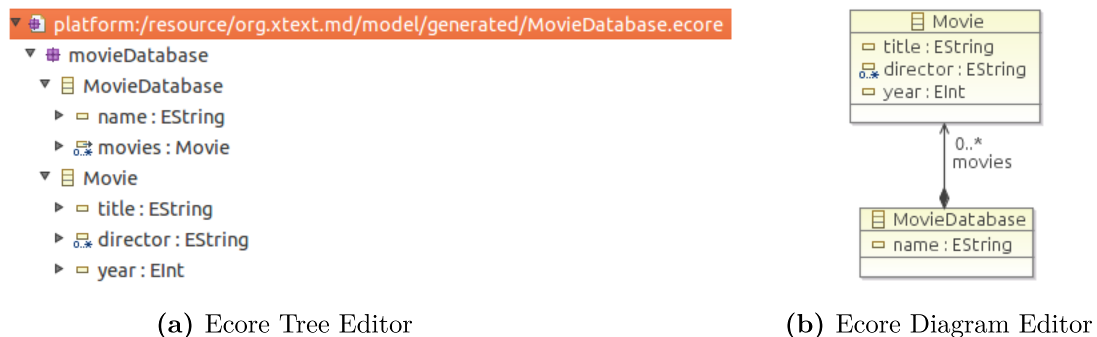
[1] http://www.eclipse.org

3.1.2 Eclipse Modelling Framework

The Eclipse Modeling Framework (EMF)[2] provides additional functionality for model-driven development, i.e., the generation of source code from abstract models.

To describe the structure of these models, *meta-models* are used. The organization of these meta-models is, again, specified by the Ecore-meta-model, which is specified by itself. To create and edit a meta-model, EMF provides two editors. Figure 3.1 exemplifies how a meta-model for a movie database may look in the *tree editor* and in the *diagram editor*.

Based on the meta-model, several Java classes are generated that provide an editor for models, a validator, serializer and JUnit tests. These classes can be customized according to the use case.



(a) Ecore Tree Editor **(b)** Ecore Diagram Editor

Figure 3.1. The movie database meta-model in the tree editor and in the diagram editor.

3.1.3 Xtext

Xtext[3] is an open source framework for the development of programming languages and Domain Specific Languages (DSLs), which are programming languages developed for a specific domain or area of application. The development started in 2006 in the scope of the Open Architecture Ware (OAW) project. Since 2008, it is developed by itemis AG[4] and is a part of EMF.

The grammar for a language is defined in a similar way to the Extended Backus-Naur Form (EBNF). Listing 3.1 shows the definition of a simple grammar for the movie database. The definition of a movie database starts with the keyword movieDatabase followed by a name. A movie database consists of several, but at least one, movies, which is indicated by the plus sign at the end of line 6. A movie entry starts with the keyword movie followed by the title. After the keyword by, the directors are listed. This may be a single

[2] http://eclipse.org/modeling/emf
[3] http://eclipse.org/Xtext
[4] http://www.itemis.de

one or several comma separated, implied by the asterisk at the end of line 9. Followed by a comma and the keyword year, the release year can be given as an optional argument, which is indicated by the question mark at the end of line 10.

Based on the grammar, a parser is generated by using Another Tool for Language Recognition (ANTLR). Additionally, a full Eclipse integration is provided, including an editor, syntax highlighting, auto completion and validation. In Figure 3.2, the generated editor can be seen.

```
1  grammar de.cau.cs.kieler.mb.MovieDatabase with org.eclipse.xtext.common.Terminals
2
3  generate movieDatabase "http://www.cau.de/cs/kieler/mb/MovieDatabase"
4
5  MovieDatabase:
6    "movieDatabase" name=ID ":" movies+=Movie+;
7
8  Movie:
9    "movie" title=STRING "by" director+=STRING ("," director+=STRING)*
10     ("," "year" year=INT)?;
```

Listing 3.1. The movie database grammar in Xtext.

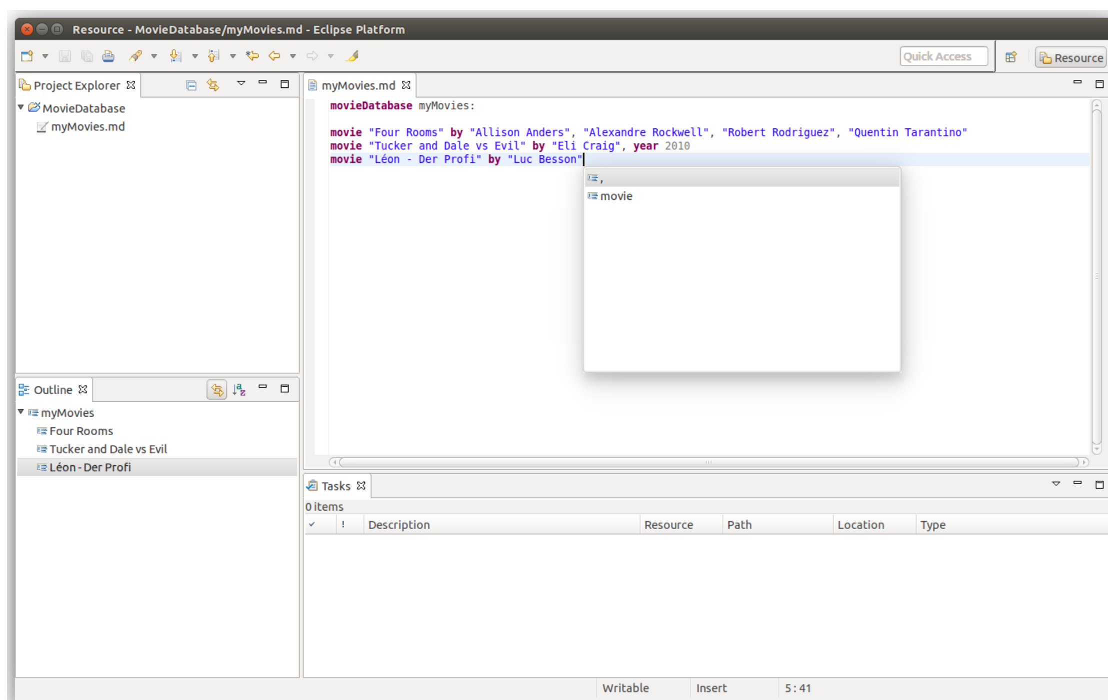


Figure 3.2. Code editor generated by Xtext.

3.1.4 Xtend

Xtend[5] is a programming language which can be seen as a Java dialect. It started as a part of Xtext in 2011 and is also developed by itemis AG. Initially, the intention was to ease Model-to-Model (M2M) transformations. However, Xtend is used in other domains, too. A main advantage in contrast to Java is the omission of unnecessary syntax overhead.

Listing 3.2 shows how a method that inserts a new movie to the movie database can be implemented with Xtend.

```
1  def addSharktopus(MovieDatabase md) {
2    val sharktopus = MovieDatabaseFactory::eINSTANCE.createMovie => [
3      title = "Sharktopus"
4      director += "Declan O'Brian"
5      year = 2010
6    ]
7    md.movies += sharktopus
8
9    md
10 }
```

Listing 3.2. Adding a movie to the database with Xtend.

In line 1, a method is declared by the keyword def. As long as the return type can be inferred, it is not required to explicitly declare it. Similar, in line 2 a new value is declared by using the keyword val. The type is inferred by the right-hand side, that is, an object of type movie created by a method generated by the movie database meta-model. The newly created movie entry is directly initialized: The *with* operator, denoted by the arrow, passes the movie object into the lambda expression defined by the square brackets. In line 7, the new movie is added to the database and in line 9, the changed database is returned. The return keyword used in Java is omitted.

Xtend is maximally compatible to Java as it compiles to readable Java code. This leads to good interoperability with Java, i.e., it is possible to use Java classes in Xtend and *vice versa*. Listing 3.3 shows how a simple Xtend method, which checks if a movie with a specific title is contained in the database, is transformed to Java.

3.2 KIELER

As mentioned in Section 1.3, the Kiel Integrated Environment for Layout Eclipse Rich Client (KIELER) is a collection of Eclipse plug-ins developed by the Real-Time and Embedded Systems group at the Christian-Albrechts-Universität zu Kiel, Germany. In this section, the components that were used in the scope of this thesis are examined briefly.

[5] http://eclipse.org/xtend

```
1  def isElement(MovieDatabase md, String movieTitle) {
2    md.movies.exists[ title.equals(movieTitle) ]
3  }
```

(a) Xtend

```
1  public boolean isElement(final MovieDatabase md, final String movieTitle) {
2    EList<Movie> _movies = md.getMovies();
3    final Function1<Movie, Boolean> _function = new Function1<Movie, Boolean>() {
4      public Boolean apply(final Movie it) {
5        String _title = it.getTitle();
6        return Boolean.valueOf(_title.equals(movieTitle));
7      }
8    };
9    return IterableExtensions.<Movie>exists(_movies, _function);
10 }
```

(b) Java

Listing 3.3. Simple transformation method in Xtend (a) and compiled to Java (b).

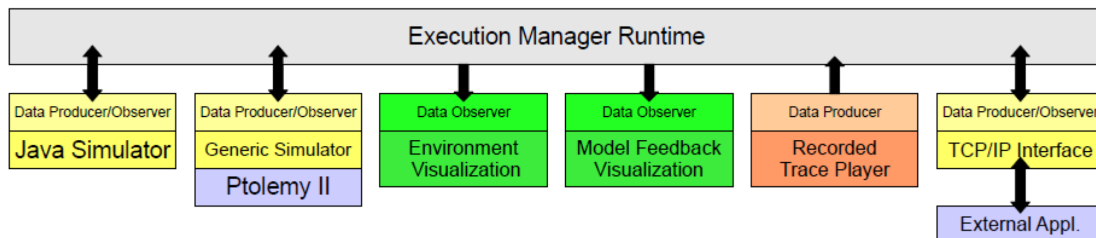


Figure 3.3. Schematic overview of KIEM.

3.2.1 KIEM

The KIELER Execution Manager (KIEM) provides the infrastructure for simulating domain specific models [Mot09, MFvH09]. It provides a visualization and a user interface to control the simulation, however, KIEM does not do any simulation computations.

To actually simulate programs with KIEM, so called *DataComponents* have to be implemented for different tasks. Figure 3.3 illustrates how they communicate through the *Execution Manager Runtime*. DataComponents have to implement an interface specified by KIEM and may observe data, produce data, or both.

Figure 3.4 shows how KIEM can be used in KIELER for simulation purposes on the example of ABRO. The *Execution Manager View*, located at the bottom, is used to compose the different DataComponents that should be used for the simulation. The green arrows are used to control the execution by either going back or forth step-wise or letting the execution continue automatically in the time interval specified in the text-field left of the buttons.

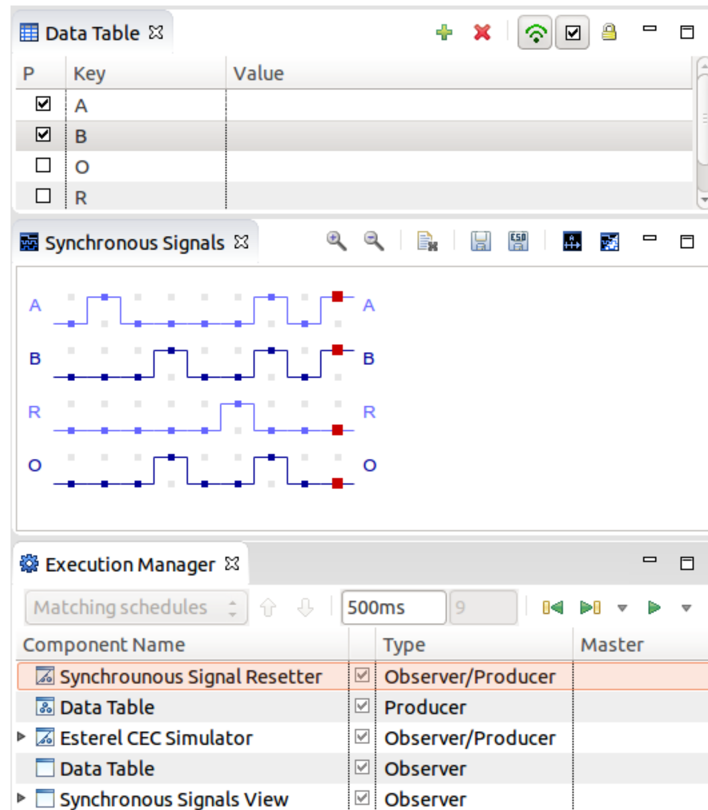


Figure 3.4. Simulation via KIEM in Eclipse.

The *Synchronous Signals View* in the middle visualizes the signals in a time-line and also allows to save execution traces. The *Data Table View* at the top is used to set the presence state of signals. By marking a signal with the checkbox and, in case of valued signals, entering a value in the designated field, the signal is present in the next tick.

3.2.2 KART

KIELER Automated Regression Testing (KART) is a framework for regression tests on simulation results. For this purpose, it is possible to save simulation traces in the Esterel Simulator Output (ESO) format. Initially developed as a format for saving Esterel traces, it can also be adopted for sequentially constructive languages.

The regression testing is done on basis of a set of models and the corresponding ESO files containing the expected reactions for given inputs. When KART is started, it simulates the models and compares the actual outputs with the expected ones. The test suite is started via a JUnit plug-in run configuration. When the model reactions differ from the expected ones, the affected model, as well as the affected tick, is denoted.

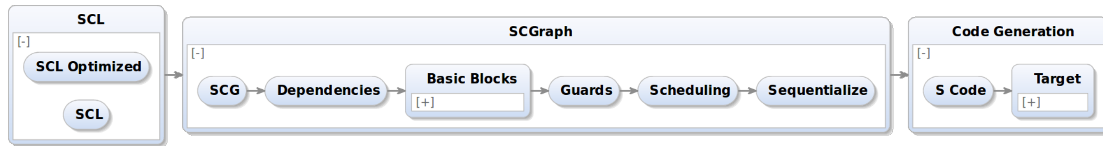


Figure 3.5. Kieler Compiler View for the Esterel compilation via SCL.

3.2.3 KIELER Compiler

The Kieler Compiler (KiCo) is a framework used to perform transformations on instances of the EObject class, which is the root class of all model objects in the EMF framework. Therefore, KiCo, which is implemented as an Eclipse plug-in, provides an extension point to register transformations either implemented in Java or in Xtend.

This is especially useful when step-by-step transformations should be applied or transformations should be reused. Also, dependencies between transformations can easily be expressed.

Figure 3.5 shows the *KiCo View*, a graphical interface for KiCo integrated to KIELER. It can easily be adjusted for different compilation chains, in this case it allows the compilation of Esterel via SCL to target code. For increasing readability, single transformations can be grouped. Grey arrows indicate dependencies, e.g., it is necessary to first transform an Esterel program to SCL before transforming the result to an SCG. Unconnected nodes imply choice, e.g., in the SCL group, alternatively to the normal Esterel to SCL transformation, an optimized variant can be chosen.

3.2.4 KIELER Model View

The KIELER Model View allows to directly inspect the graphical representation of a model next to the editor. When transformations are applied via KiCo, the resulting model representation, which can also be code if no graphical representation is defined, can be seen in the KIELER model view. This allows a direct observation of intermediate results and can therefore be used for verification and bug tracking.

Figure 3.6 shows the KIELER Model View for the ABO example, as presented in Section 1.2, in SCL. As the *Dependencies* transformation is selected in KiCo, the Model View to the right shows the SCG with overlaid concurrent dependencies and is updated every time changes on the SCL code are saved.

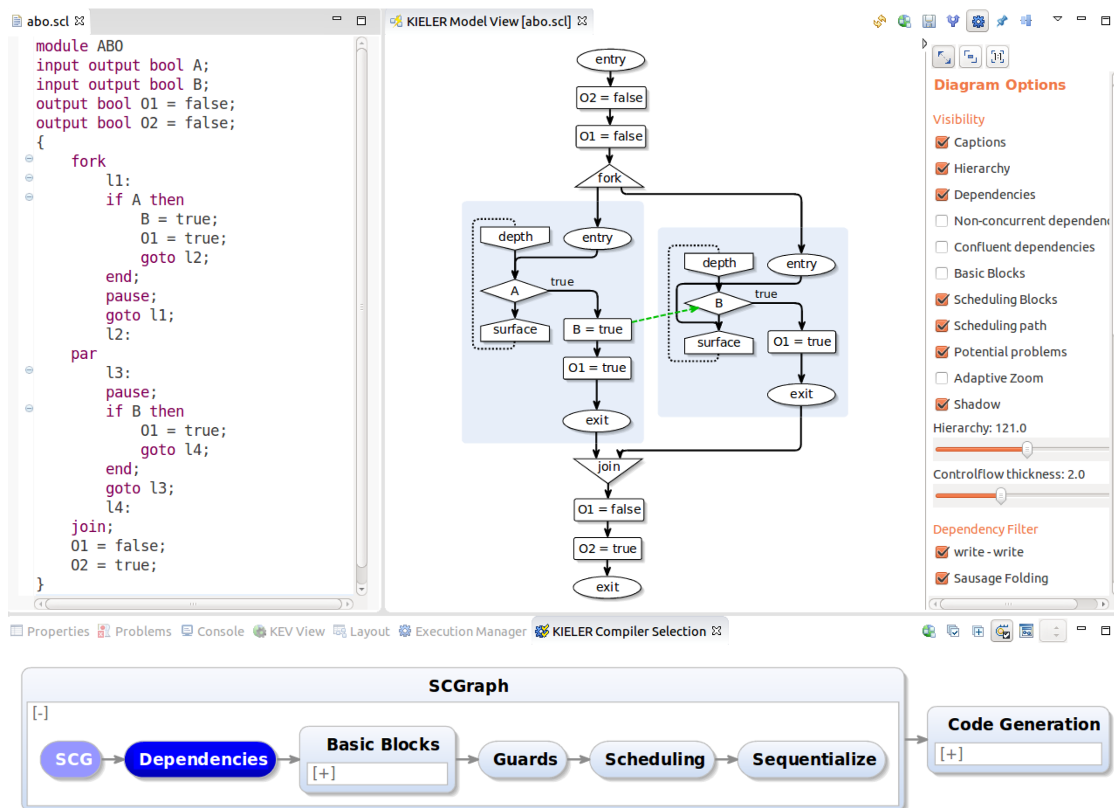


Figure 3.6. Kieler Compiler View and Model View

Chapter 4

Esterel to SCL Transformation

Model-to-Model (M2M) transformations can be classified as *endogenous* and *exogenous*. Whilst in endogenous transformations the source and the target meta-models are the same, they differ in exogenous transformations [MG06]. Hence, the transformation from Esterel to SCL presented in this chapter is exogenous.

Whilst some statements can be transformed in a trivial way, the main challenge is to transform preemptive statements correctly, for instance, suspend or abort. They require the insertion of additional code at tick boundaries. As SCL makes heavy use of goto statements, these have to be placed with care. It is not admissible to jump directly from one thread to another [vHMA+13].

4.1 Definitions

To keep the transformation rules simple and comprehensible, some abbreviations are used.

p
: A statement that is possibly compound.

l, l1, l2, . . .
: Labels that are not already in use.

l_exit
: A label at the end of the current thread. goto l_exit accordingly means to go to the end of the immediate surrounding thread.

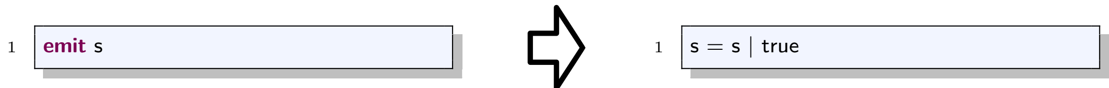
gotoj l
: goto l if l is in the current thread and goto l_exit otherwise.

p $[s_1 \rightarrow s_2 \mid s_3 \rightarrow s_4]$
: All occurrences of s_1 in p are replaced by s_2 and those of s_3 by s_4.

4.2 Transformation Rules

In the following, the main transformation rules from Esterel to SCL are described. To ease the comprehensibility of the transformations, besides the resulting SCL code also the SCG, as presented in Section 1.2.2, is given when adequate.

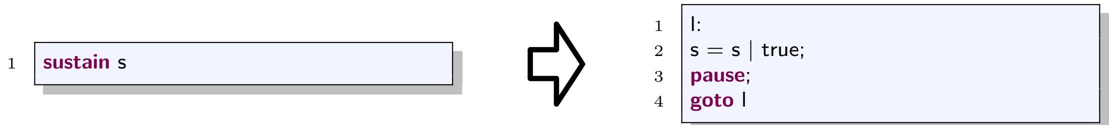
4.2.1 Emit



Listing 4.1. emit Transformation

Since SCL uses variables instead of signals, they are translated to boolean values. When a signal is emitted, the corresponding boolean value is set to true. As shown in Listing 4.1, this happens via a relative write, which always writes true into the variable. The idea behind using a relative write might get clearer when looking at the transformation of output and local signals in the following section.

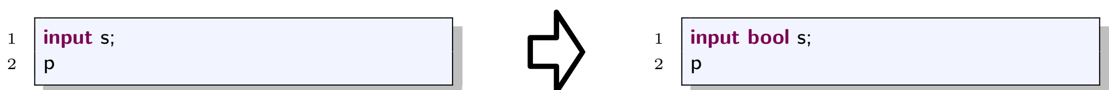
Sustain



Listing 4.2. sustain Transformation

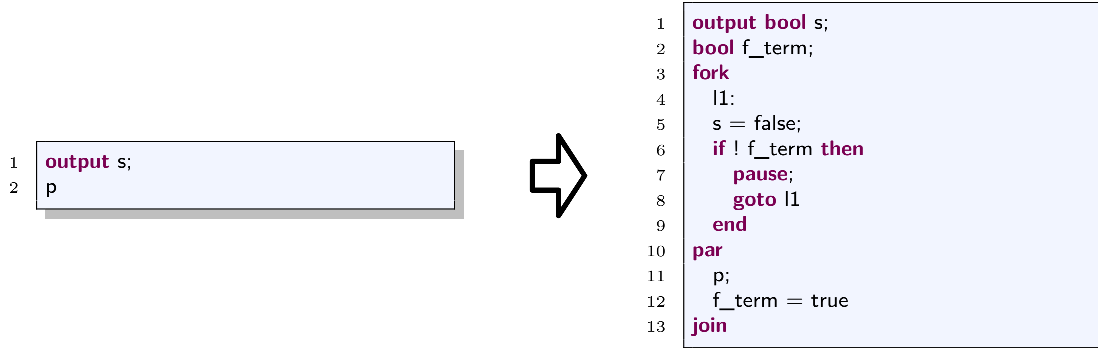
The sustain statement emits a signal in every tick. In SCL, the variable corresponding to the signal is set to true with a relative write in every tick by a goto loop as depicted by the rule given in Listing 4.2.

4.2.2 Signals



Listing 4.3. Input Signal Transformation

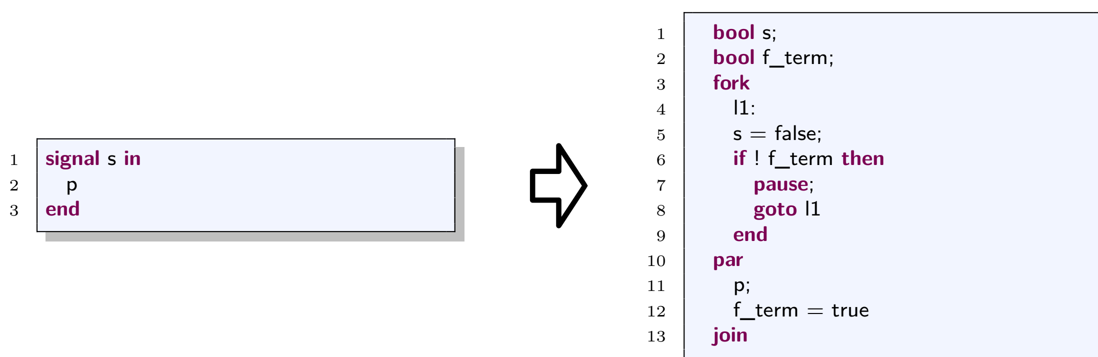
Since in the SC MoC signals are considered as syntactic sugar, signals in Esterel are translated to boolean variables. Listing 4.3 shows the transformation rule for input signals. These are signals whose state is determined by the environment.



```
1 output s;
2 p
```

```
1  output bool s;
2  bool f_term;
3  fork
4    l1:
5    s = false;
6    if ! f_term then
7      pause;
8      goto l1
9    end
10 par
11   p;
12   f_term = true
13 join
```

Listing 4.4. Output Signal Transformation



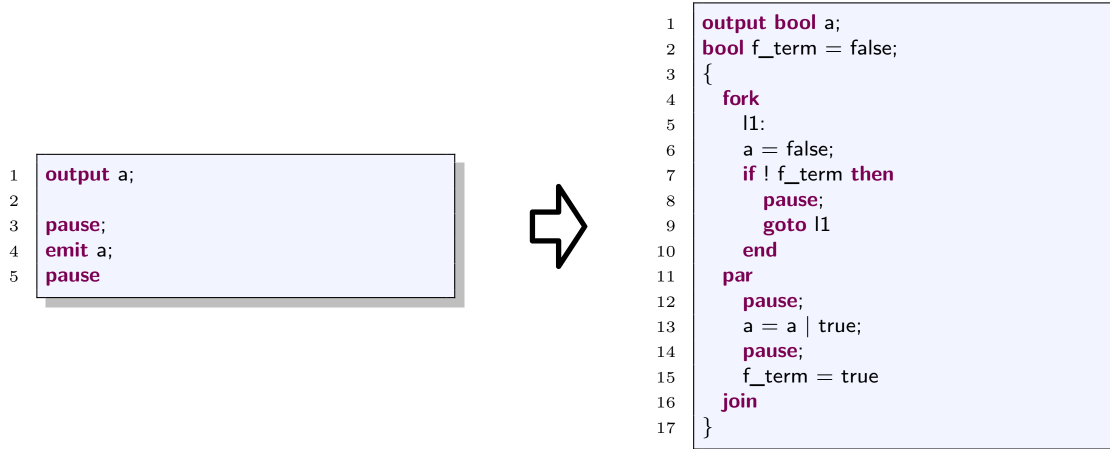
```
1 signal s in
2   p
3 end
```

```
1    bool s;
2    bool f_term;
3    fork
4      l1:
5      s = false;
6      if ! f_term then
7        pause;
8        goto l1
9      end
10   par
11     p;
12     f_term = true
13   join
```

Listing 4.5. Local Signal Transformation

Listing 4.4 and Listing 4.5 present the transformation rule for output and local signal declarations. As signals are only present in a tick if they are emitted, the corresponding boolean value has to be false initially in every tick. This is achieved by introducing a thread running in parallel to the statements for which the signals are declared. In this thread, the boolean variables representing signals are set to false by an absolute write. As described in Section 1.2, when executed in parallel, absolute writes are scheduled before relative writes. Therefore, a variable representing a signal is set to false in every tick at first. When a signal is emitted, according to the initialize-update rule, it is set to true by a relative write afterwards. Thus, the variable is false by default and only true in ticks in which it is emitted.

The additional boolean variable f_term indicates the termination of the statement p. Thus, the thread setting all variables corresponding to a signal to false terminates when p does.



```
output a;

pause;
emit a;
pause
```

```
output bool a;
bool f_term = false;
{
  fork
    l1:
    a = false;
    if ! f_term then
      pause;
      goto l1
    end
  par
    pause;
    a = a | true;
    pause;
    f_term = true
  join
}
```

Listing 4.6. Output/Local Signal Transformation Example

Example

As an example, Listing 4.6 shows the transformation of output signals and Figure 4.1 the corresponding SCG. The Esterel program does nothing in the first tick, emits a in the second, and terminates in the third. As described in Section 1.2.2, the dashed arrows in the corresponding SCG in Figure 4.1 denote dependencies. In the second tick a is set to false at first by the absolute write in the left thread and subsequently to true by the relative write in the other thread. The initialize-update dependency is indicated by the blue arrow. At the beginning of the next tick, a is again set to false and f_term is set to true before it is read by the other thread. This write-read dependency is indicated by the green arrow.

Reincarnation

A notorious phenomenon in Esterel is the *reincarnation problem* [Ber93]. That is, a local signal scope is left and re-entered in the same tick. Listing 4.7 exemplifies the reincarnation problem. In the second tick, s is emitted and afterwards the scope of the

```
loop
  signal s in
    present s then
      emit o
    end;
    pause;
    emit s
  end
end
```

Listing 4.7. Reincarnation Example in Esterel

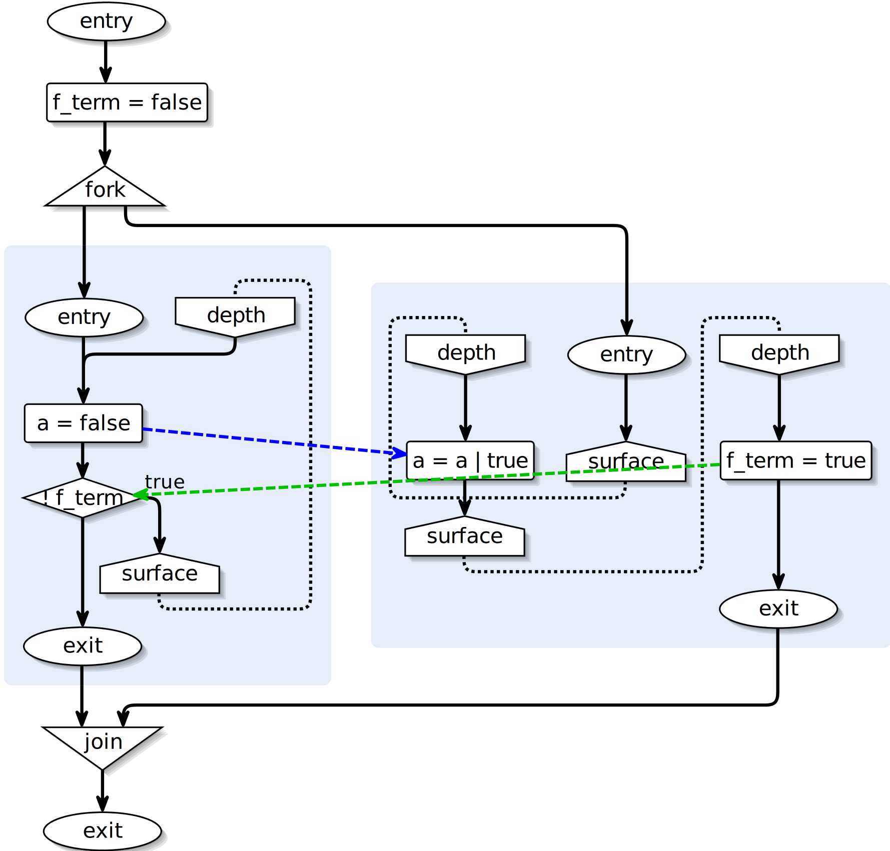


Figure 4.1. The SCG corresponding to the example for the output/local signal transformation in Listing 4.6.

local signal declaration is left. Due to the surrounding loop, it is re-entered in the same tick and, in line 3, the presence state of s is checked. The expected behavior is that o is not emitted since a *new* scope was entered, whilst s was emitted in the *old* scope.

Different approaches exist on how to handle the reincarnation problem [Tar04]. However, the transformation rule for local signal declarations to SCL handles this inherently. Figure 4.2 shows the SCG corresponding to the program in Listing 4.7. The left-most thread sets the global variable o to false by an absolute write in every tick and originates from the global signal transformation. In the other thread, according to the local signal rule, another thread is created to set the local variable s to false in every tick.

The thread in the middle is the transformed local signal body. At first, s is initialized to false before being read due to the initialize-read dependency. This read represents the present statement in the originating Esterel program. o, which was initialized with false by the left-most thread, is not set to true and the tick ends.

In the next instance, s is set to false by the absolute write and subsequently to true by the relative write, which represents the emission in line 7 of the originating Esterel

program in Listing 4.7. Afterwards, the parallel statement is left and re-entered. Again, the absolute write is executed prior to the read and `o` is not not set to `true`, i.e., the emission in the original Esterel program in line 4 is not executed.

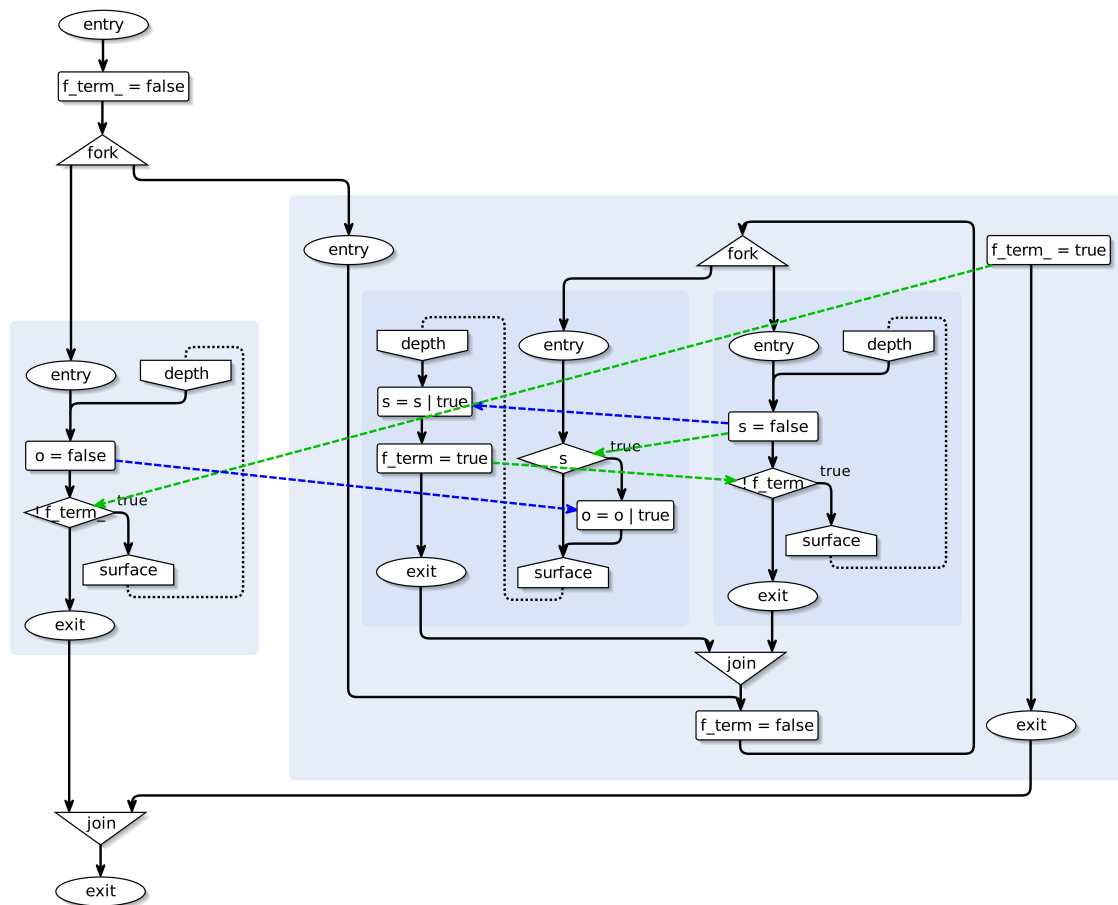
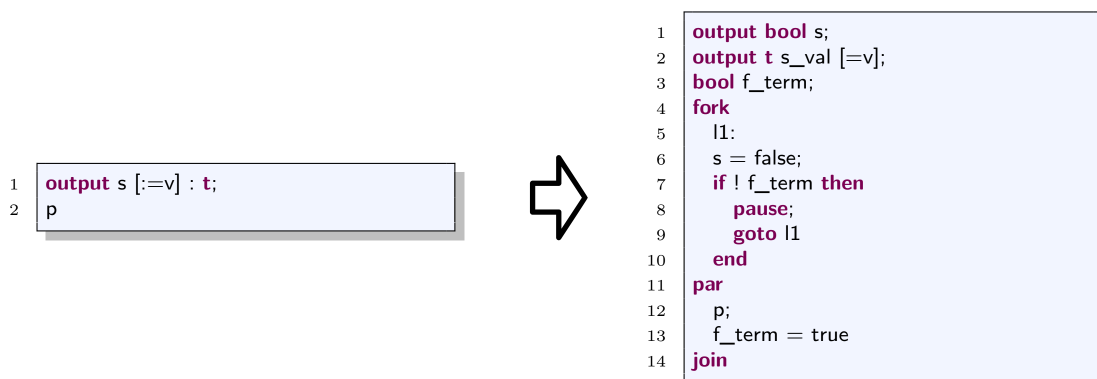


Figure 4.2. The SCG corresponding to the program in Listing 4.7 with overlaid dependencies.

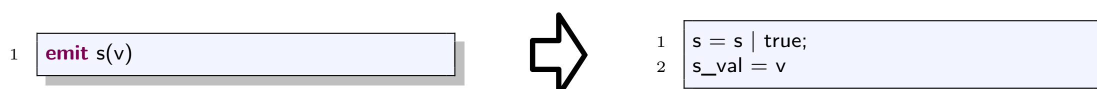
4.2.3 Valued Signals



```
output s [:=v] : t;
p
```

```
output bool s;
output t s_val [=v];
bool f_term;
fork
  l1:
  s = false;
  if ! f_term then
    pause;
    goto l1
  end
par
  p;
  f_term = true
join
```

Listing 4.8. Valued Output Signal Transformation



```
emit s(v)
```

```
s = s | true;
s_val = v
```

Listing 4.9. Valued emit Transformation

Besides *pure signals*, which only have a presence state, *valued signals* additionally allow a value of a given type. As seen in Listing 4.8 and Listing 4.9, output valued signal declarations are transformed similarly to pure signal declarations. Besides a boolean variable representing the presence state, a variable of the corresponding type is created. Local signal declarations are transformed analogous. A valued emit is represented by setting the corresponding boolean variable to true and setting an associated variable to the given value.

Even though a single Esterel statement is split into two SCL statements, no unintentional behavior due to the scheduler is introduced. Since Esterel restricts valued signals to have a unique value in each tick, there should not be a concurrent emission with a different value unless a resolution function is specified, as described below. However, if there was such a construct, the SCL/SCG dependency analysis, as described in Section 1.2.2, would recognize a write-write conflict.

Furthermore, when a conditional depends on the presence state of a signal and, depending on it, uses the value of the signal, the initialize-update-read approach ensures the intended behavior. As Figure 4.3 illustrate, the relative write to the variable holding the presence state has to be scheduled prior to the conditional representing the present statement in Esterel due to the update-read dependency. Additionally, the read access to the variable holding the value is scheduled after the assignment of the variable due to the initialize-read dependency.

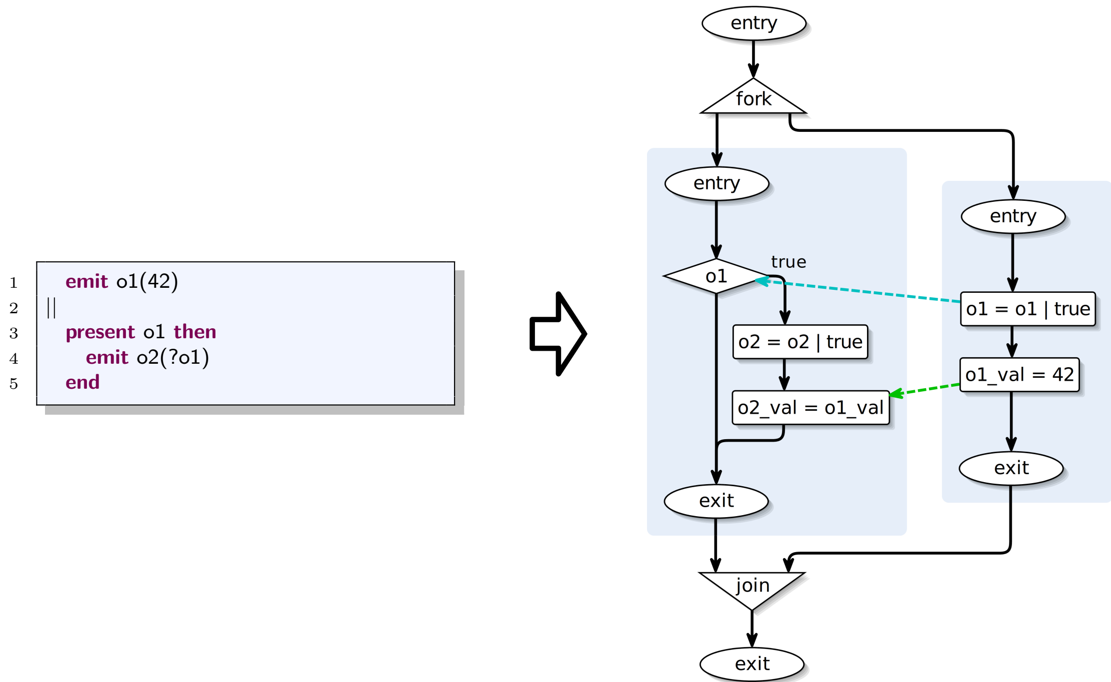


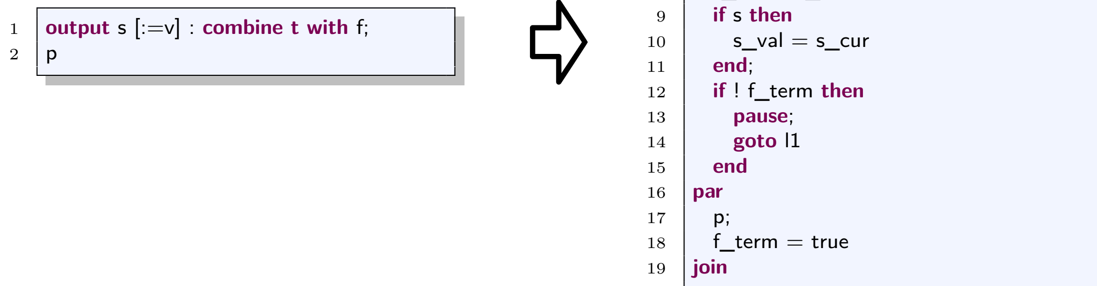
Figure 4.3. Even though the valued emit statement is split to two SCL statements, no unintended behavior is introduced.

Resolution Functions

As every signal must have a well-defined value in each tick, it is not allowed to emit a signal with different values in one tick by default. This can be resolved by a *resolution function*, which is a commutative and associative function to combine the different emitted values. The requirement for the function to be commutative and associative is necessary as the order in which the values are combined must not have influence on the result [Ber00].

Esterel provides a basic construct to declare resolution functions for signals. As seen in Listing 4.10, compared to the transformation of valued signals without a resolution function, an additional variable s_cur is introduced to hold the value of a signal for the current tick. This variable is set to the neutral element of the resolution function, denoted by f_e, at the beginning of each tick by an absolute write in line 8. As shown in Listing 4.11, the value of a signal emission is now stored in s_cur instead of s_val.

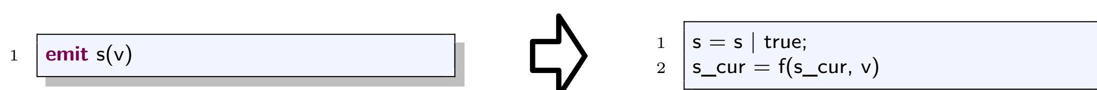
As the value of a signal is persistent across ticks, s_val should only be set to a new value if the signal is emitted, which is done in the parallel region in line 10. Without s_cur, concurrent absolute writes on s_val would be possible, since the variable needs to be initialized in every tick in which it is emitted [vHMA+13].



```
1 output s [:=v] : combine t with f;
2 p
```

```
1  output bool s;
2  output t s_val [=v];
3  t s_cur;
4  bool f_term;
5  fork
6    l1:
7    s = false;
8    s_cur = f_e;
9    if s then
10     s_val = s_cur
11   end;
12   if ! f_term then
13     pause;
14     goto l1
15   end
16 par
17   p;
18   f_term = true
19 join
```

Listing 4.10. Definition of a resolution function in Esterel and in SCL. f_e is the neutral element of the function f.



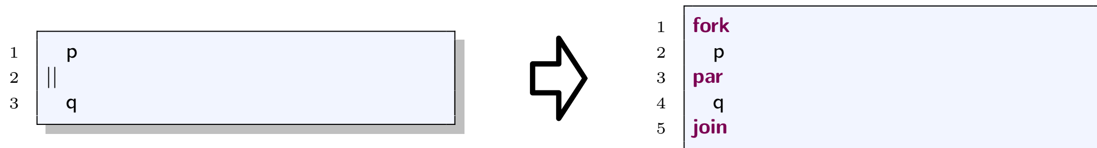
```
1 emit s(v)
```

```
1 s = s | true;
2 s_cur = f(s_cur, v)
```

Listing 4.11. emit Transformation with Resolution Function

4.2.4 Parallel

Listing 4.12 presents the transformation rule for concurrency. An Esterel parallel statement is translated straightforwardly to an SCL parallel statement. To avoid confusion with the logical *or* operator in C (||), SCL uses a fork-par-join construct. Moreover, the scope of the statement is clearer. In Esterel, an additional block statement, indicated by square brackets, may be needed to clarify the statement's range.

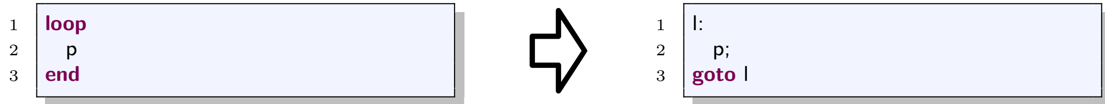


```
1   p
2 ||
3   q
```

```
1 fork
2   p
3 par
4   q
5 join
```

Listing 4.12. Parallel Transformation

4.2.5 Loop



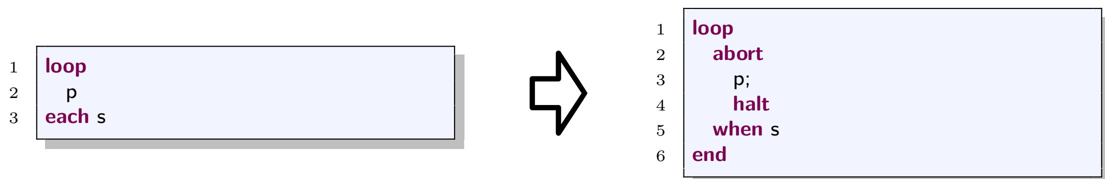
```
1 loop
2   p
3 end
```

```
1 l:
2   p;
3 goto l
```

Listing 4.13. loop Transformation

A loop statement can be transformed by placing a label at the start of the statement body and a goto targeting this label at the end as seen in Listing 4.13.

Loop Each



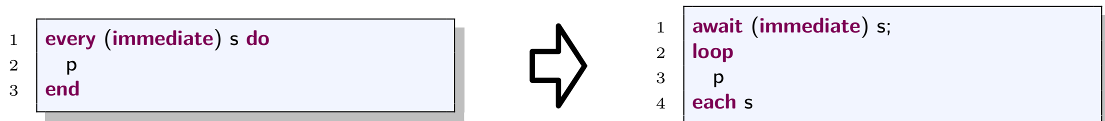
```
1 loop
2   p
3 each s
```

```
1 loop
2   abort
3     p;
4     halt
5   when s
6 end
```

Listing 4.14. loop each is transformed to a combination of loop, abort and halt, which is subsequently transformed to SCL.

Listing 4.14 depicts the transformation of the loop each statement, which restarts its body whenever a specified signal is present. This behavior can also be expressed by nested loop and abort statements [BG92, Ber00] that can further be transformed to SCL.

Every Do



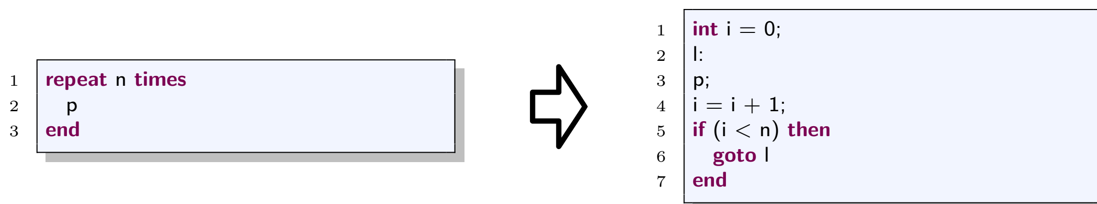
```
1 every (immediate) s do
2   p
3 end
```

```
1 await (immediate) s;
2 loop
3   p
4 each s
```

Listing 4.15. The every do statement can be expressed in Esterel by a loop and a preceding await.

The every do statement is similar to loop each but does not start the execution until the specified signal event occurs. Again, the statement can be expressed by a combination of other statements as seen in Listing 4.15 [BG92, Ber00], which are subsequently transformed to SCL.

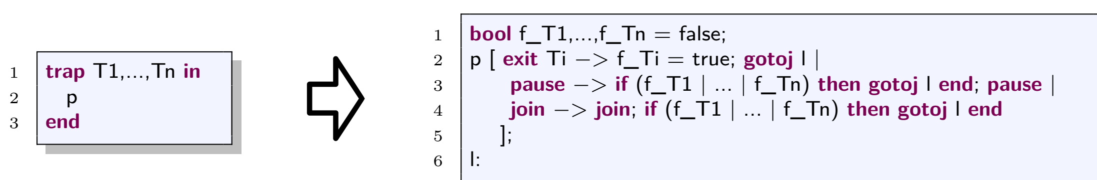
Repeat



Listing 4.16. repeat n times Transformation

The repeat statement executes its body a specified number of times. In SCL, a counting variable keeps track of how many iterations have been done. Depending on this variable, the statement body is executed again or not. The transformation rule is given in Listing 4.16.

4.2.6 Trap



Listing 4.17. trap Transformation

trap statements in Esterel preempt the execution of the body when an exit statement with the corresponding variable is triggered. These exit statements can only be executed within the trap statement for that they were defined. Parallel running threads in the statement body execute the current tick, even if the trap statement is triggered and are preempted when the control reaches a tick boundary. Therefore, traps are neither strong nor weak preemptive.

As depicted by Listing 4.17, this behavior is translated to SCL by setting a flag when an exit statement is triggered to signal concurrently running threads to not continue the execution when a tick boundary is reached. Therefore, the flag is checked before each pause statement. If it is evaluated to true, the control continues at the end of the trap statement.

Exception Handling

The trap statement in Esterel can be seen as an exception handling mechanism. When an exit statement is executed, this can be interpreted as a thrown exception. These exceptions can optionally be caught by the surrounding trap. With regard to this point

of view, additional handle statements can react depending on which exception, i.e., trap variable, caused the preemption. Basically, these are comparable to *cases* for abort or await statements. A difference is that if more than one exception is thrown at once in the same scope, all of them are handled, i.e., all do-blocks are executed in parallel. Additionally, exceptions may be valued, similar to signals. The value can only be accessed in the handle statements [BC84].

This behavior is adopted to SCL by starting a thread after the transformed trap statement for every handler. As an example, Figure 4.4 shows how handle blocks are translated to parallel threads.

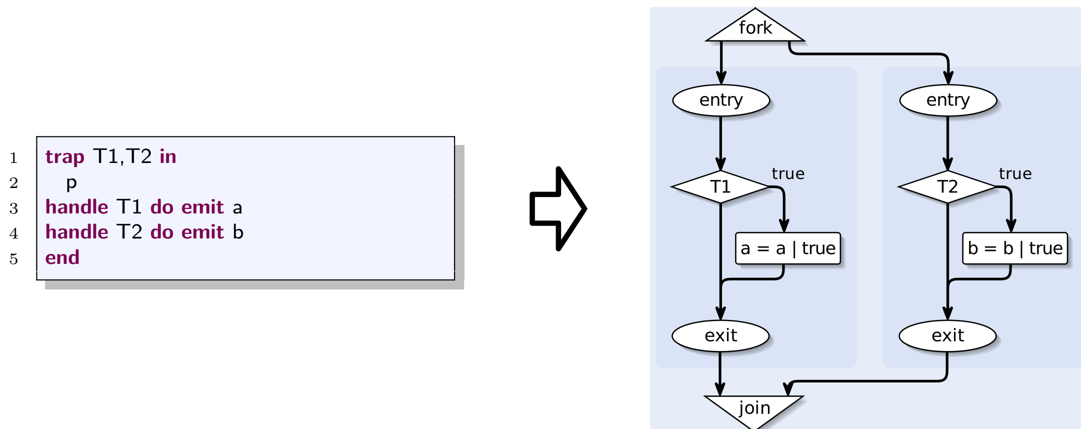


Figure 4.4. Exception handling blocks in Esterel are transformed to concurrent threads executing the do-block if the trap was activated.

Potentially Instantaneous Loops

In general, loops in SCL, as well as in Esterel, are not allowed to be instantaneous as this would mean the execution of an infinite amount of micro steps within one tick. Whether a loop is potentially instantaneous or not has to be decided statically, as the program should not produce run-time errors [Ber00].

How potential instantaneous loops are detected depends on the compiler, thus, a program rejected by some compiler may be accepted by another one. As a complete analysis to decide whether a statement may execute instantaneously would mean to do an exhaustive check of all possible execution paths, many compilers may rather decline correct programs than performing such an expensive analysis [TdS03].

For example, the program in Listing 4.18a is rejected by the CEC due to a potentially instantaneous loop. A full analysis would have shown that the exit statement in line 6 is never executed as the conditional in line 5 is a constant with the value false. In contrast, the program in Listing 4.18b is accepted, as the CEC recognizes that no exit statement is reachable in the first tick of the loop body.

Applying the transformation rules on the program in Listing 4.18b, the SCL program

```
1  loop
2    trap T in
3      pause
4    ||
5      if false then
6        exit T
7      end
8    end
9  end
```

(a) Esterel program that is rejected by the CEC.

```
1  loop
2    trap T in
3      pause
4    ||
5      if false then
6        pause;
7        exit T
8      end
9    end
10 end
```

(b) Esterel program that is accepted by the CEC.

Listing 4.18. Rejection of potentially instantaneous loops in the CEC.

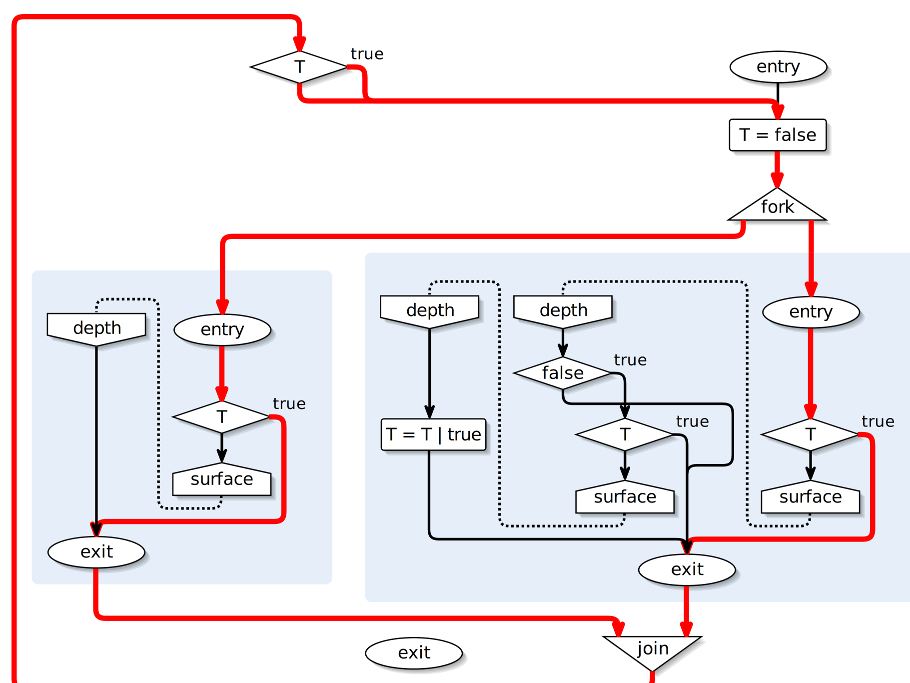


Figure 4.5. The SCG corresponding to the SCL program in Listing 4.19a. A potentially instantaneous loop is generated by a trap nested within a loop in the source Esterel program in Listing 4.18b.

presented in shortened form in Listing 4.19a is generated. The corresponding SCG can be seen in Figure 4.5. It contains a potentially instantaneous loop, indicated by the red line, and thus is rejected by the SCG compiler. This instantaneous path originates by the transformation of pause statements in the trap body as shown in Listing 4.17. Since in concurrently running threads the execution continues until the next tick boundary, when some other thread executes an exit statement, the check whether an exception has

been raised is done prior to each pause. As it would be admissible if a trap statement terminates instantaneously as long as there is no surrounding loop, the check in the initial tick is mandatory.

To increase the set of Esterel programs that can be scheduled using SCL/SCG, an analysis whether it can be assured that no exit statement is reachable in the initial tick is done. If so, each pause statement that can only be reached in the initial tick is not enriched by the goto to the end of the trap statement. With this extension, the program in Listing 4.18b can be compiled correctly via SCL.

Listing 4.19b illustrates how the SCL program looks after applying this extension to the SCL program in Listing 4.19a. The conditional gotos in lines 5–7 and lines 11–13 in Listing 4.19a can be removed since T cannot be set to true in the first tick.

Even though weak immediate abortion may cause the same issue, most Esterel compilers would refuse a weak immediate abortion statement within a loop. Since the triggering signal is not scoped by the statement, an analysis whether the statement may terminate instantaneously is more expensive [PTvH06].

```
1  l1:
2  {
3    bool T = false;
4    fork
5      if T then
6        goto l3
7      end;
8      pause;
9      l3:
10   par
11     if T then
12       goto l4
13     end;
14     pause;
15     if false then
16       if T then
17         goto l4
18       end;
19       pause;
20       T = T | true;
21       goto l4
22     end;
23     l4:
24   join;
25   if T then
26     goto l2
27   end;
28   l2:
29 };
30 goto l1
```

(a) Before Optimization

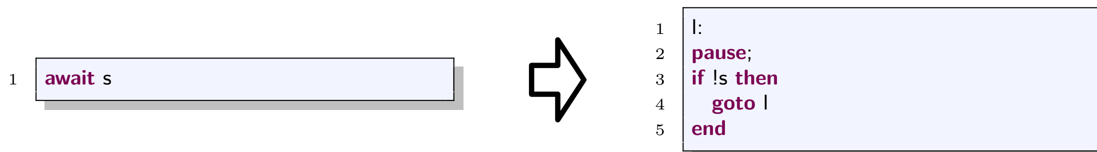
```
1  l1:
2  {
3    bool T = false;
4    fork
5      pause;
6      l3:
7    par
8      pause;
9      if false then
10       if T then
11         goto l4
12       end;
13       pause;
14       T = T | true;
15       goto l4
16     end;
17     l4:
18   join;
19   if T then
20     goto l2
21   end;
22   l2:
23 };
24 goto l1
```

(b) After Optimization

Listing 4.19. By removing gotos originating by the trap transformation in the initial tick, a larger set of Esterel programs can be scheduled after the transformation to SCL.

4.2.7 Await

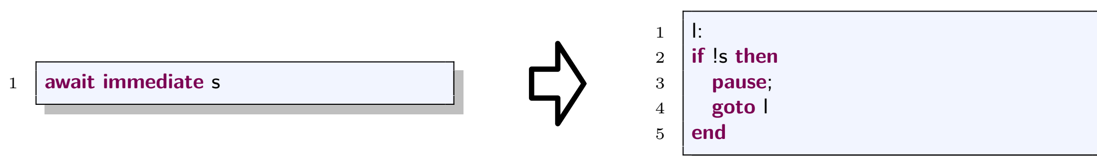
Delayed Await



Listing 4.20. Delayed await Transformation

A delayed await statement stops the execution until the specified signal is present after the initial tick. According to the transformation rule in Listing 4.20, it is transformed to a loop statement that is left when the signal becomes present.

Immediate Await



Listing 4.21. Immediate await Transformation

In contrast to the delayed await statement, the immediate variant also considers the initial tick. In the transformed SCL program, this is realized by moving the pause statement into the conditional as depicted in Listing 4.21.

Await Cases

Esterel provides *await cases* that allow to wait for several signals at the same time and react depending on which of them has been present. In case of multiple signals being present in a tick, the order of the cases determines the priorities.

Figure 4.6 shows an Esterel await case statement and the corresponding SCG in abbreviated form. Immediate cases are checked prior to the pause statement, afterwards all cases are checked in the given order to respect the priorities.

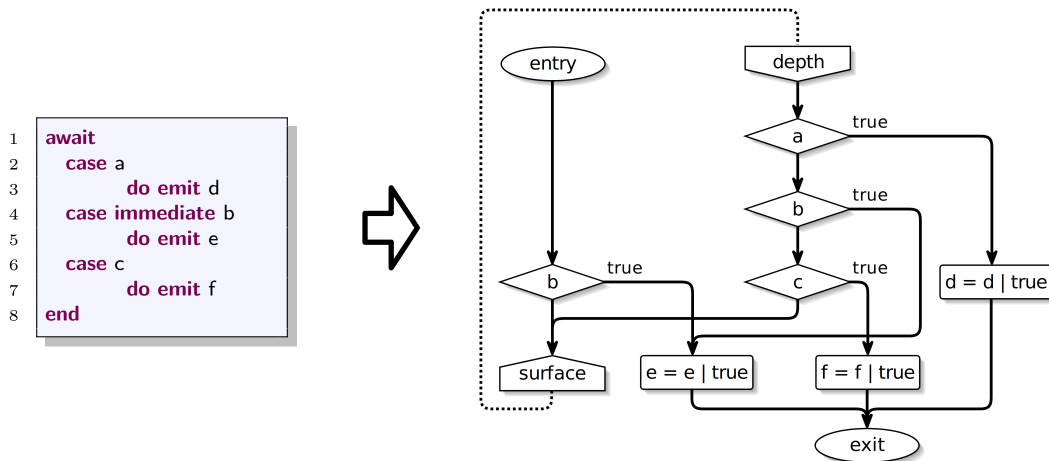
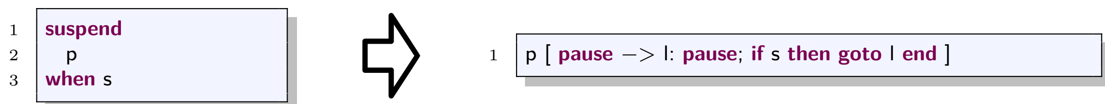


Figure 4.6. await Cases Transformation to SCG

4.2.8 Strong Suspend

Strong Delayed Suspend

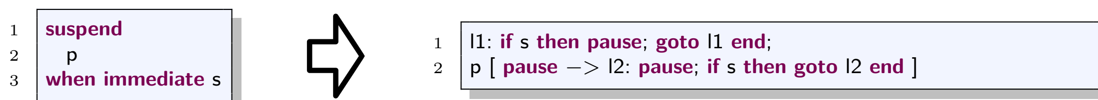


Listing 4.22. Strong Delayed suspend Transformation

A strong delayed suspend statement stops the execution when a specified signal is present after the initial tick. The current tick's micro steps are not executed and the execution continues in the next tick in which the specified signal is absent.

In SCL, the strong delayed suspend statement is realized by a conditional goto loop surrounding every pause in the body of the statement, as shown in Listing 4.22. Hence, at the beginning of each tick, i.e., after each pause, the condition is checked and, when evaluated to true, the pause is repeated.

Strong Immediate Suspend



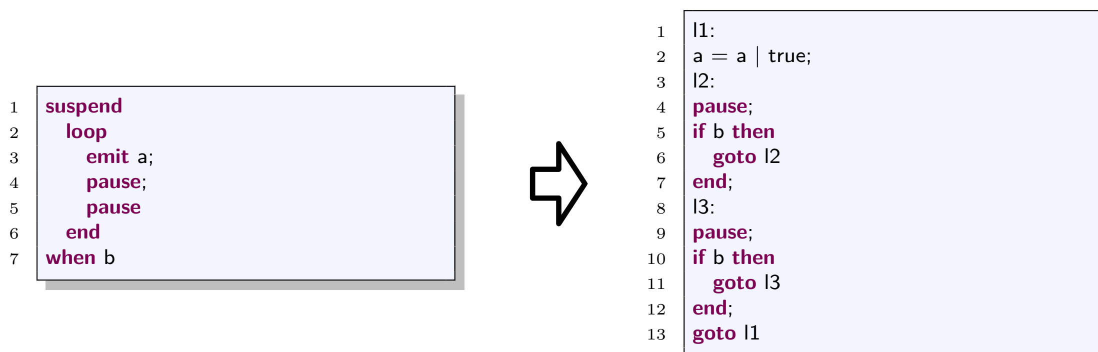
Listing 4.23. Strong Immediate suspend Transformation

The immediate variant of suspend also considers the statement's initial tick. When the corresponding signal is present, the execution of the body of the statement does not start.

The transformation to SCL is similar to the delayed case and is presented in Listing 4.23. An additional label followed by a conditional checking whether the execution should be suspended is added at the statement's start. Again, a goto loop realizes the suspension. Afterwards, the transformation rule for delayed suspension is applied.

Example

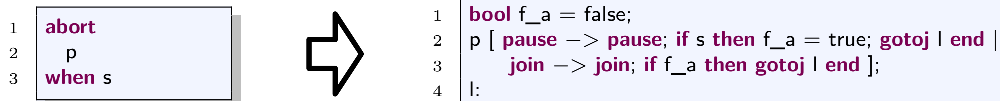
As an example, Listing 4.24 shows the transformation of an Esterel program with a delayed suspend statement. The label in line 1 and the goto in line 13 of the resulting SCL program originate from the Esterel loop statement. In the SCL program, each pause statement is tagged by a unique label located in line 3 and line 8. After the pauses, the signal triggering the suspension is checked and, if evaluated to true, the preceding pause is executed again.



Listing 4.24. Strong Delayed suspend Example (Signal Transformation Omitted)

4.2.9 Strong Abort

Strong Delayed Abort



```
abort
  p
when s
```

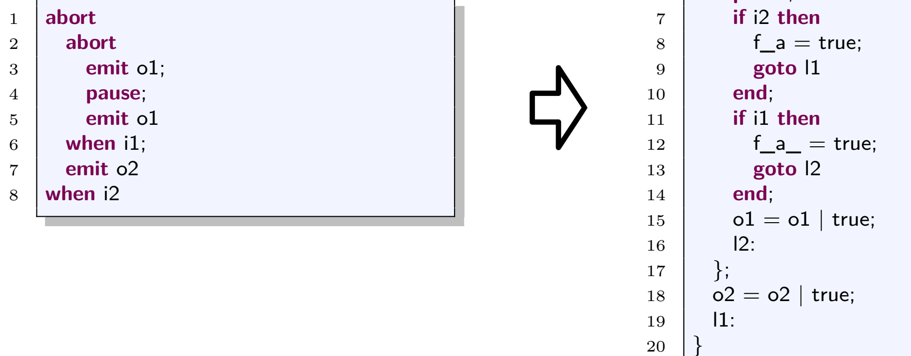
```
bool f_a = false;
p [ pause −> pause; if s then f_a = true; gotoj l end |
    join −> join; if f_a then gotoj l end ];
l:
```

Listing 4.25. Strong Delayed abort Transformation

A strong delayed abort statement preempts the execution when a specified event occurs after the initial tick. As seen in Listing 4.25, in the corresponding SCL code after each pause the condition is checked. If it evaluates to true, a flag f_a is set and, depending on in which thread the control flow is, it continues at the thread end or at the end of the abort statement.

If within the abort body parallel threads have been started, subsequently to each join the corresponding flag is checked. The control flow continues at the end of the current thread or the at end of the abort statement if the flag is evaluated to true.

Example



```
abort
  abort
    emit o1;
    pause;
    emit o1
  when i1;
  emit o2
when i2
```

```
{
  bool f_a = false;
  {
    bool f_a_ = false;
    o1 = o1 | true;
    pause;
    if i2 then
      f_a = true;
      goto l1
    end;
    if i1 then
      f_a_ = true;
      goto l2
    end;
    o1 = o1 | true;
    l2:
  };
  o2 = o2 | true;
  l1:
}
```

Listing 4.26. Example for the strong delayed abort transformation (signal transformation omitted).

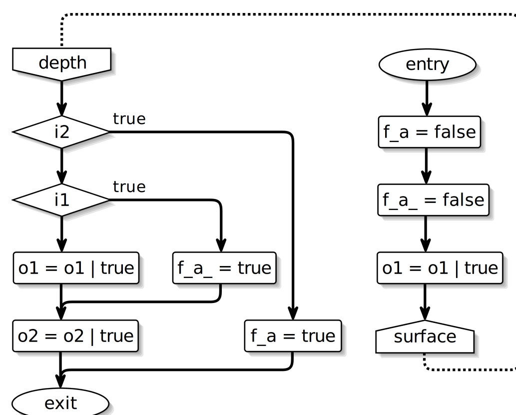


Figure 4.7. The strong delayed abort example in Listing 4.26 illustrated as an SCG (signal transformation omitted).

As an example, Listing 4.26 shows the transformation of the strong delayed abort statement to SCL. Figure 4.7 illustrates the resulting SCG. The Esterel code emits o1 in the first tick and o1 as well as o2 in the second tick. Nested abort statements preempt this behavior when the corresponding input signals are present.

If only i1 is present in the second tick, o2 should be emitted and o1 should be absent. If i2 is present in the second tick, nothing should be emitted. The precedence of the outermost abort statement can be seen in the SCL code in the conditionals following each pause in lines 7–14. At first, it is checked whether i2 is true and afterwards if i1 is true. Depending on the evaluated condition, o2 is set to true or not.

Listing 4.27 exemplifies why the additional flag f_a is used. After a join, a jump to the next thread exit or the label marking the end of the abort body may be necessary. In case a direct check on the abort condition would be used, the execution of line 13

```
1  input i;
2  output o1, o2, o3;
3
4  abort
5    [
6      emit o1
7    ||
8      emit o2
9    ];
10   emit o3
11 when i
```

```
1  {
2    bool f_a = false;
3    fork
4      o1 = o1 | true;
5      l2:
6    par
7      o2 = o2 | true;
8      l3:
9    join;
10   if f_a then
11     goto l1
12   end;
13   o3 = o3 | true;
14   l1:
15 }
```

Listing 4.27. Without the additional flag, the strong delayed abort transformation may not behave as aspected (signal transformation omitted).

would be omitted when the abort condition holds in the initial tick. Since this flag is only necessary in the special case where the abort body contains a potentially instantaneous parallel region, it can be omitted if there is none.

Strong Immediate Abort

```
1 abort
2   p
3 when immediate s
```

```
1 if s then goto l end;
2 p [ pause -> pause; if s then gotoj l end |
3     join -> join; if s then gotoj l end ];
4 l:
```

Listing 4.28. Strong Immediate abort Transformation

A strong immediate abort, in contrast to a delayed abort, also considers the initial tick. Accordingly, the transformation seen in Listing 4.28 is similar, since only an initial check for the specified signal is added at the start of the statement. As also the initial tick is considered, the additional flag f_a for the case of an instantaneous parallel statement can be omitted.

Additional Cases

Both abort variants allow additional cases, that is, preemption may be triggered by different signals. Depending on which of them actually causes the abortion, a specified do-block is executed. This is realized by transforming the cases into nested aborts, whereas the nesting depends on the ordering of the cases and thus the priorities. In case of preemption, the outermost abort, representing the first case, is prioritized.

In the resulting SCL program, after the transformed nested abort statement, it is checked which signal causes the abortion. Thereby, the conditionals are ordered by the priorities to ensure that the right do-block is executed. An additional depth flag, which is set to true when a tick boundary is executed in the abort body, enables to decide whether also delayed or only immediate cases may be taken.

Listing 4.29 shows an example for the transformation of abort cases. In the resulting SCL program, in line 7, it is checked whether i2 is true and potentially the execution is aborted. As the other case is delayed, it is first checked in line 15. As the first case is prioritized, the second one is checked afterwards in line 19. After the abort statement's body terminated, in line 28 and the following, the possible do-blocks are executed. The do-block for the first case is only executed if i1 is present and it is not the initial tick, i.e., f_depth is set. The second case is immediate and therefore can be executed only depending on i2. Since only one case should be executed, after the do-block, in lines 30 and 34 the control is handed to the end of the abort statement.

Even though an optimization may be conceivable, since instead of checking i2 in line 7 and in line 19, the label l4 could be moved before line 7, this optimization would only

target a small set of structures. An additional delayed case with lower priority that the i2 case would not allow to reuse the check in line 7 without additional gotos and conditional, since the priorities should be respected. However, the do-blocks located in lines 28–35, which may have an arbitrary size, are not duplicated.

```
1  abort
2    sustain o1
3  when
4    case i1 do
5      emit o2
6    case immediate i2 do
7      emit o3
8  end
```

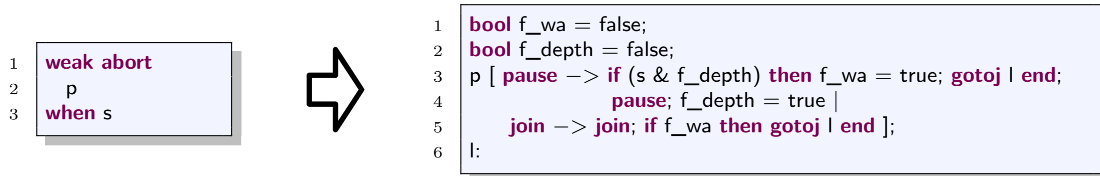
```
1  {
2    bool f_depth = false;
3    {
4      bool f_a = false;
5      {
6        bool f_a_ = false;
7        if i2 then
8          f_a_ = true;
9          goto l3
10       end;
11       l4:
12       o1 = o1 | true;
13       pause;
14       f_depth = true;
15       if i1 then
16         f_a = true;
17         goto l2
18       end;
19       if i2 then
20         f_a_ = true;
21         goto l3
22       end;
23       goto l4;
24       l3:
25     };
26     l2:
27   };
28   if (i1 & f_depth) then
29     o2 = o2 | true;
30     goto l1
31   end;
32   if i2 then
33     o3 = o3 | true;
34     goto l1
35   end;
36   l1:
37 }
```

Listing 4.29. Example for the transformation of abort with cases (signal transformation omitted).

4.2.10 Weak Abort

In contrast to strong abortion, weak abortion allows the execution of the micro steps of the current tick even if abortion is triggered. This can be seen as a *last wish* of the body of the statement.

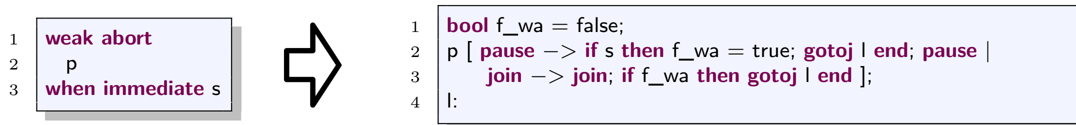
Weak Delayed Abort



Listing 4.30. Weak Delayed abort Transformation

Listing 4.30 shows the transformation rule. As weak abortion in general allows the current tick to execute its micro steps, at the end of each tick, thus directly before each pause statement, the corresponding signal is checked. Additionally, the f_depth flag is checked which indicates that it is not the initial tick. This is necessary since delayed weak abortion does not react in the first tick. As also done in the case of strong abortion, after a join, it is checked whether abortion is triggered. If so, the control continues at the end of the current thread or if it is in the current thread, at the end of the weak abort statement.

Weak Immediate Abort



Listing 4.31. Weak Immediate abort Transformation

The transformation for the immediate case of weak abort, which can be seen in Listing 4.31, is similar to the delayed case. As also the initial tick is considered, the f_depth flag is omitted.

Potentially Instantaneous Loops

As for the trap transformation described in Section 4.2.6, also weak abortion may produce potentially instantaneous loops, as a conditional goto is placed before instantaneously reachable tick boundaries. Whilst these loops are tolerable for the immediate case since

most Esterel compilers would refuse them [PTvH06], for the delayed case the same refinement as for the trap transformation is done. If a pause is only reachable in the initial tick, there is no need to insert the goto in the delayed case. Even though this procedure does not completely avoid the creation of potentially instantaneous loops, it allows the compilation of a reasonably higher amount of Esterel programs.

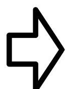
Listing 4.32 shows a simple example. As the delayed weak abort in the source Esterel code in Listing 4.32a is only active in the second tick, it has no effect. Regardless of s, the program should be paused forever.

Listing 4.32b shows the corresponding SCL program without the refinement. The goto in line 7 is potentially reachable in the initial tick since no exhaustive analysis is done whether the conditional in line 5 may evaluate to true. As after the label l2 a goto restarts the program and line 7 is again instantaneously reachable, the program contains a potentially instantaneous loop.

Since the pause statement is only reachable in the initial tick with respect to the scope of weak abort, in Listing 4.32c the preceding conditional goto is omitted. Thus, there is no potentially instantaneous loop anymore.

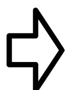
```
1  loop
2    weak abort
3      pause
4    when s
5  end
```

(a) Source Esterel Program



```
1  l1:
2  {
3    bool f_wa = false;
4    bool f_depth = false;
5    if (s & f_depth) then
6      f_wa = f_wa | true;
7      goto l2
8    end;
9    pause;
10   f_depth = true;
11   l2:
12 };
13 goto l1
```

(b) Without Refined Transformation

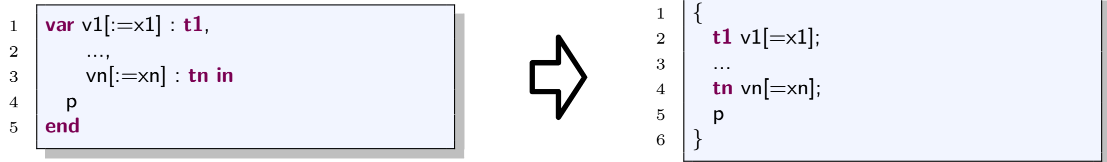


```
1  l1:
2  {
3    bool f_wa = false;
4    bool f_depth = false;
5    pause;
6    f_depth = true;
7    l2:
8  };
9  goto l1
```

(c) With Refined Transformation

Listing 4.32. The transformation rule for weak abortion may produce unnecessary potentially instantaneous loops. With the refined transformation, this is avoided in many cases.

4.2.11 Local Variable Declarations

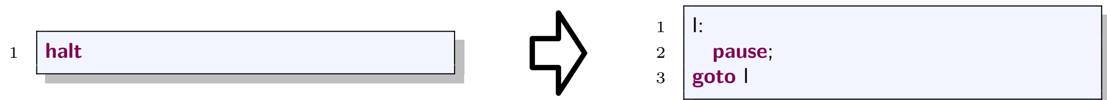


Listing 4.33. Local Variable Transformation

Besides local signals, Esterel provides local variable declarations of arbitrary types. Again, if the locally declared variable name is already declared, it is shadowed. Within the body of the statement, variables may be assigned and read, e.g., to emit a signal with a value stored in a variable.

As SCL provides statement scopes with the same behavior, the transformation is straightforward as illustrated in Listing 4.33.

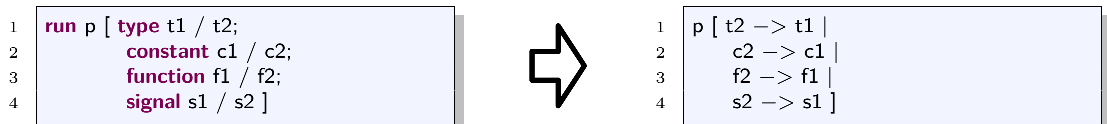
4.2.12 Halt



Listing 4.34. halt Transformation

Esterel's halt statement stops the execution by pausing forever. Similar to its translation to kernel statements, in the resulting SCL statement, a goto-loop repeats a pause statement. Listing 4.34 illustrates the transformation of the halt statement.

4.2.13 Module Instantiation



Listing 4.35. run Transformation

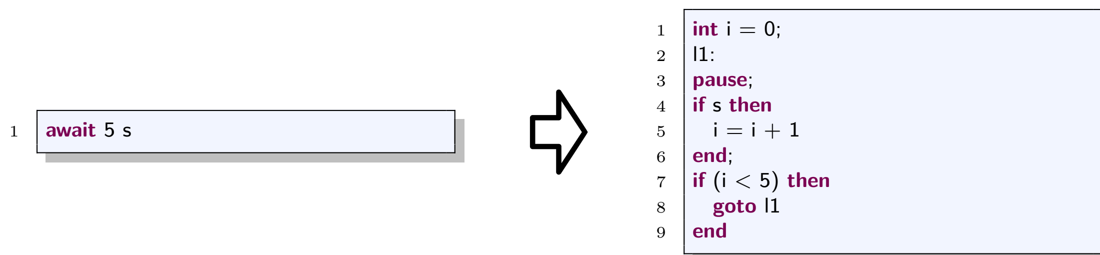
Esterel allows the instantiation of a module within another module by the run statement, where the module to instantiate is specified by its name. The specified module must not declare any signals that are not also declared in the parent module as long as no renaming to globally defined objects is explicitly specified [Ber00].

Since SCL does not provide such a statement, when a run statement is transformed in the resulting SCL program it is replaced by the transformed module. Variables may be renamed with respect to an optional list after the module name as showen in Listing 4.35.

4.2.14 Count Delays

Besides delayed and immediate expressions, Esterel provides *count delays* [Ber00]. These are delayed expression that require the specified event to occur in a particular number of ticks. For instance, await 5 s means that the execution is stopped until s has been present five times after the initial tick. Count delays can also be used to specify how often an event in preemptive statements, such as abort, has to occur before preemption takes place.

In SCL, these delays can be realized by using an integer variable for counting the ticks in which the specified event occurs. In contrast to the previously defined rules, the predicate that states if the statement is triggered changes to a comparison of the given count delay and the state of the counting variable. As a statement may be executed more than once by the use of loops, it has to be ensured that counting variables are set to zero at the start of the corresponding statement. As an example, Listing 4.36 shows the transformation of an await statement that is triggered by a count delay.



Listing 4.36. Example for the Count Delay Transformation

4.2.15 Additional Language Features

Some language features of Esterel can be transformed straightforwardly, as briefly described in the following.

pause
: Since pauses indicate tick boundaries in Esterel, as well as in SCL, they are directly transformed.

present s **then** p **else** q
: Esterel's present statement is transformed to a conditional in SCL that is triggered by the variable holding the presence state of the corresponding signal.

pre(s)
The pre expression returns the presence state or the value of a signal in the previous tick. The same operator is part of the SCL expressions and returns the value of a variable in the previous tick.

function $f(t_1, \cdots, t_n)$ **:** t_m
In Esterel, functions defined in the host language can be used. SCL provides a similar feature.

procedure $p(t_1, \cdots, t_n)(t_{n+1}, \cdots, t_m)$
Also host language procedure calls, which have no return value but allow to pass arguments by call-by-reference, are possible in Esterel. Again, SCL provides a similar feature.

4.3 Step-Wise Transformation of ABRO

Figure 4.8 shows the step-wise transformation of ABRO as introduced in Section 1.1. At first, the await statements are translated as described in Section 4.2.7. Afterwards, the parallel statement and the scope marking square brackets are transformed to the SCL fork-par-join construct.

Further, the emission of O becomes a relative write to true in SCL and the halt statement is translated according to the rule given in Section 4.2.12. Then, the abort statement is transformed as stated in Section 4.2.9 leading to inserted conditionals after pauses and joins. Since there is no potentially instantaneous parallel statement within the abort body, the additional flag is omitted.

Afterwards, the loop statement is transformed by using a label and a goto. Finally, the signal declarations are transformed and a thread for resetting output variables at the start of each thread is added as described in Section 4.2.2.

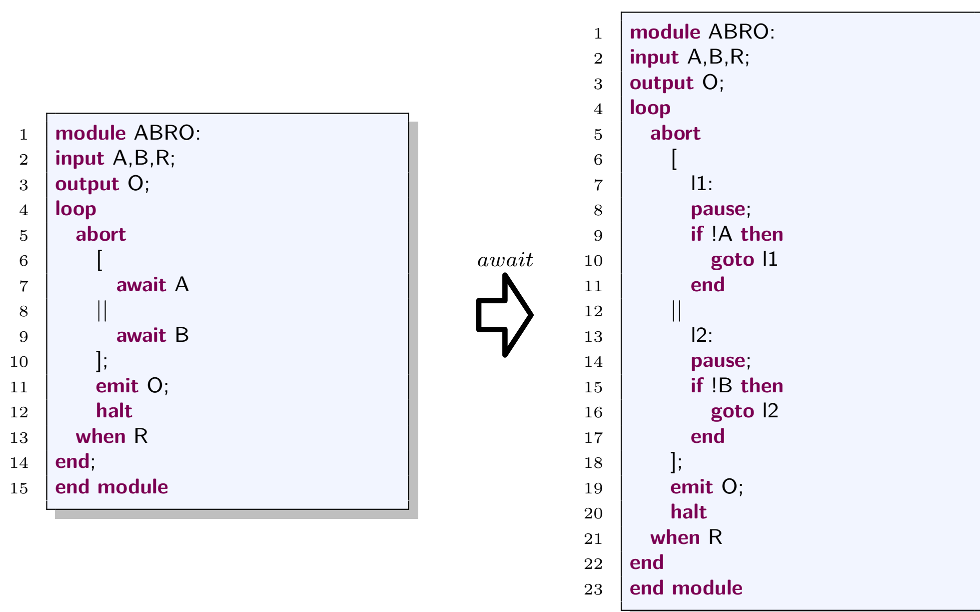


Figure 4.8. Step-wise transformation of ABRO (a).

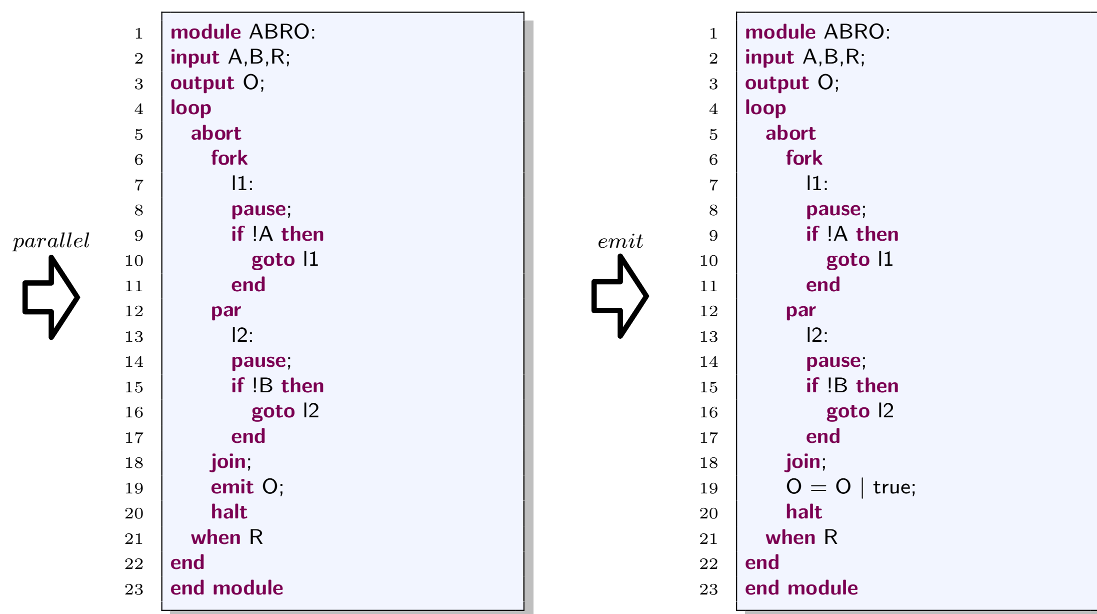


Figure 4.8. Step-wise transformation of ABRO (b).

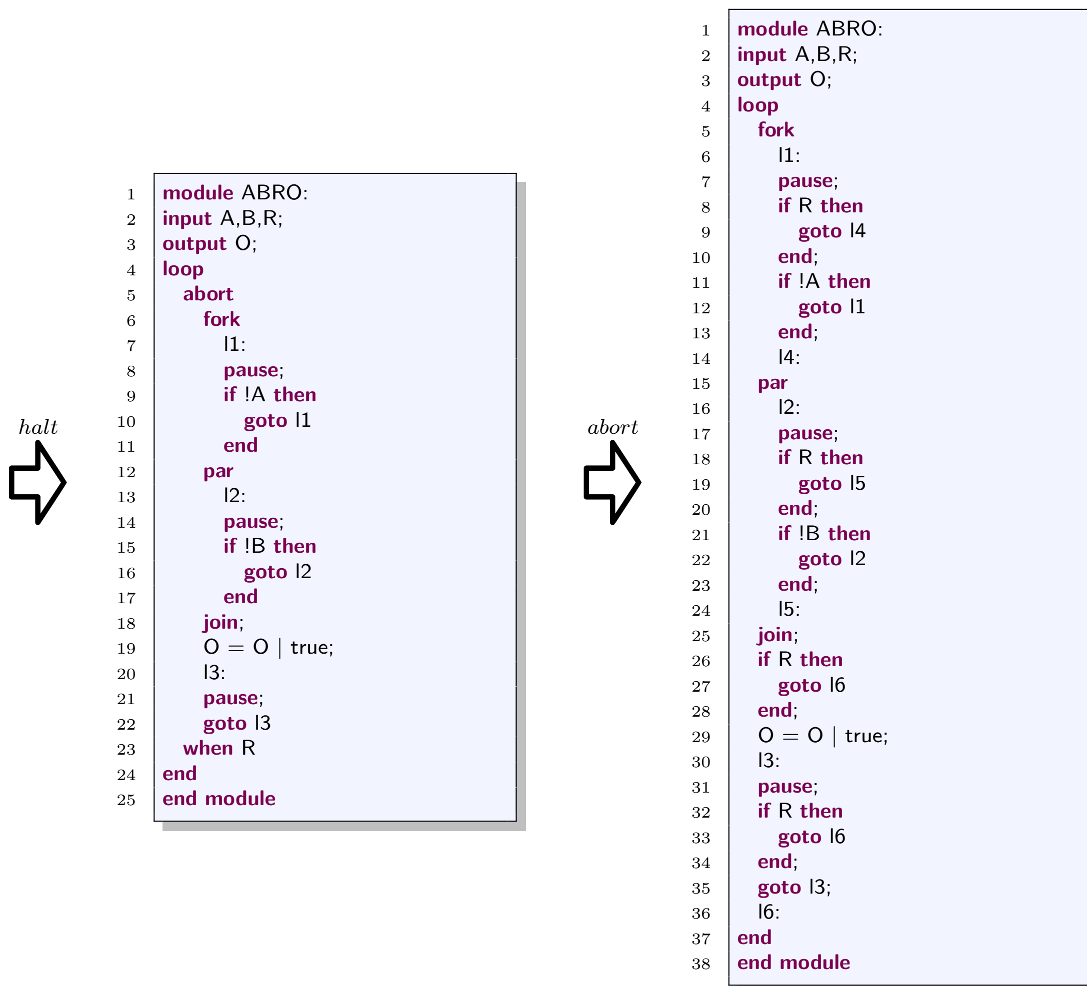


Figure 4.8. Step-wise transformation of ABRO (c).

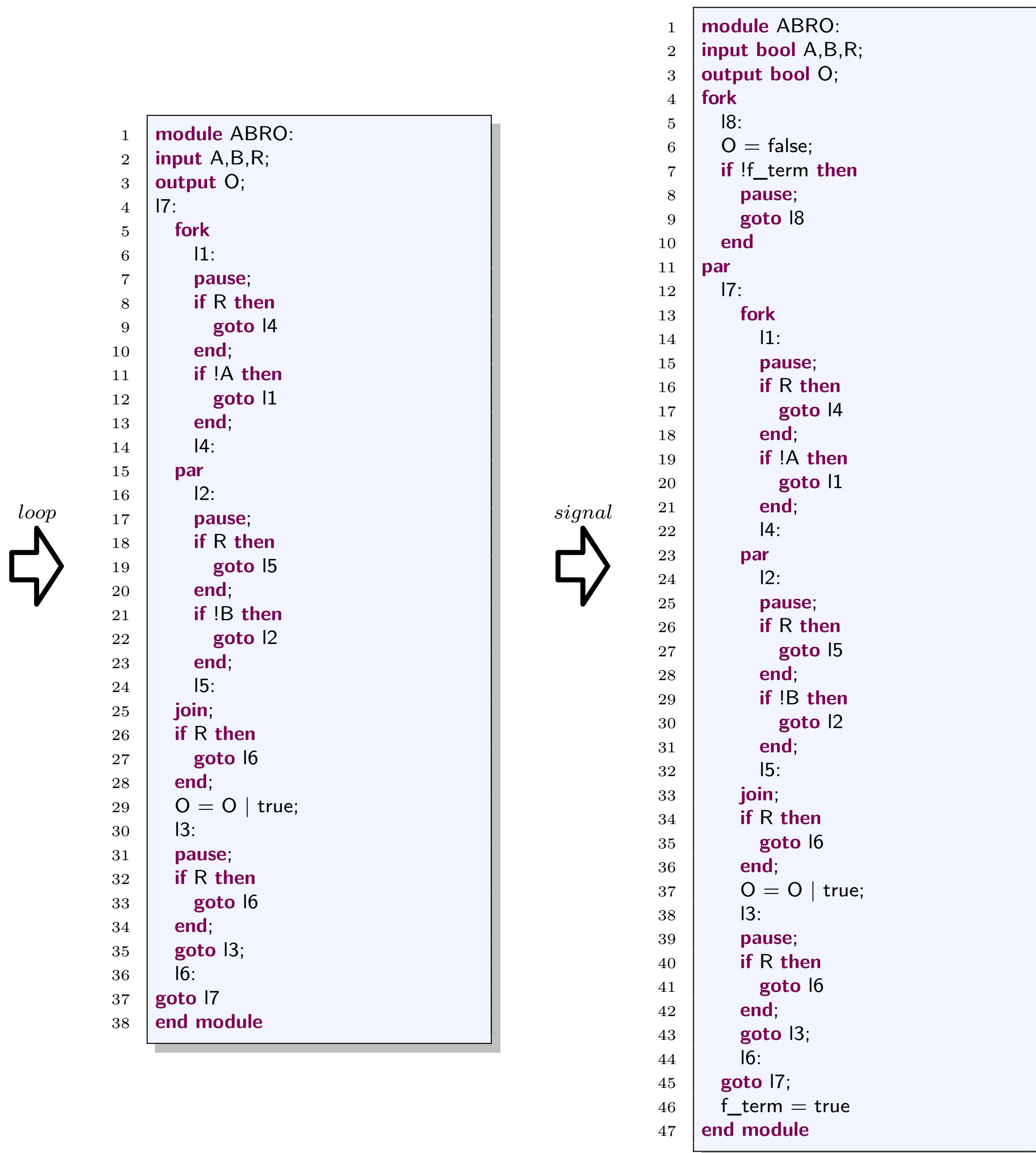
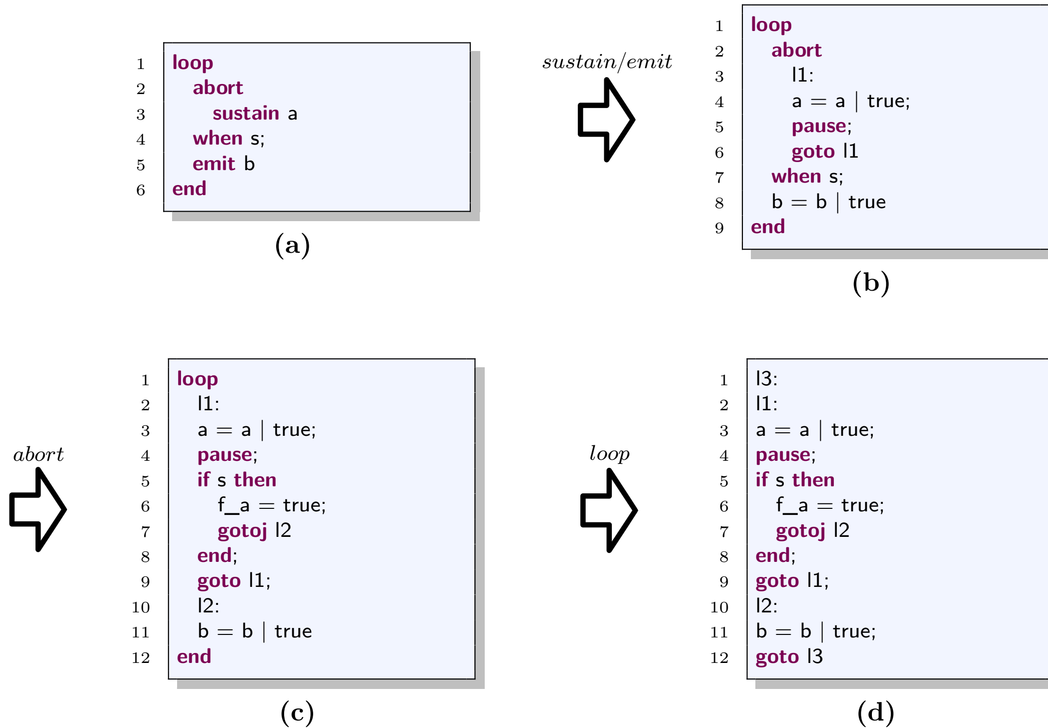


Figure 4.8. Step-wise transformation of ABRO (d).



Listing 4.38. Generation of redundant code when transforming Esterel to SCL.

4.4 Optimization

As the transformation rules should be simple and intelligible, the resulting code may contain overhead and redundant statements. Listing 4.38 illustrates the generation of redundant statements. The behavior of the program in Listing 4.38a is that the Esterel program emits `a` as long as `s` is absent. When `s` gets present, `b` is emitted and the `loop` body starts again. The corresponding SCL program in Listing 4.38d contains redundant code. In lines 1 and 2, two subsequent labels mark the same position in the code.

This section presents some methods to avoid and remove redundant code. Besides an optimized transformation rule for the translation from Esterel to SCL, also general SCL optimizations are presented.

4.4.1 Optimized Signal Transformation

The transformation rules for output and local signals presented in Section 4.2.2 may produce unreachable code and unused variables. The introduction and assignment of `f_term` is needless when the program does not terminate. A simple, although conservative, analysis can determine whether `f_term` should be produced or not.

The analysis is based on inductive reasoning. An `abort` statement terminates, whilst a `loop` statement does not. A parallel statement terminates when all its threads terminate

and so on.

The conservative nature of this analysis becomes clear when examining the conditional. Only when none of the conditional branches terminates, the conditional is considered to never terminate. As this decision may also be effected by the condition, which might always be evaluated to true, not all never terminating programs are recognized and f_term might be produced but remains needless.

4.4.2 SCL Optimization

When transforming Esterel to SCL, the resulting code may contain redundancy which can be removed by optimizations. Some approaches have been proposed by S. Smyth [Smy13] and were reused, others were improved or newly invented in the scope of this thesis.

Removing Unused Labels

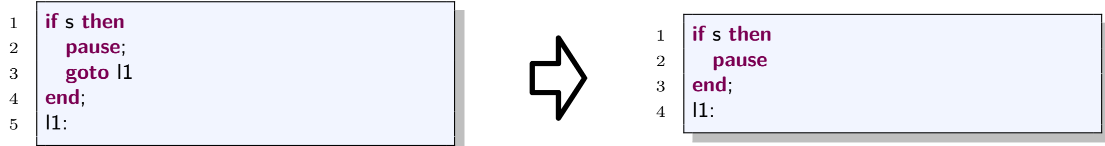
A simple label optimization was implemented prior to this thesis work that removes unused labels, i.e., labels which are not targeted by any goto.

Removing Subsequent Labels

Considering the example in Listing 4.38, also another label optimization is conceivable to remove subsequent labels. Therefore, all labels following another label are deleted and the referencing goto target labels are changed correspondingly.

Removing Redundant Gotos

In case the target label of a goto is a successor statement, the goto is superfluous and can be removed. This optimization, which was implemented prior to this thesis work, is extended to also consider superfluous jumps out of conditional branches. As an example, Listing 4.39 shows this behavior. Therefore, when the goto that should be optimized is at the end of a StatementSequence, it is checked whether the StatementSequence is a Conditional. In that case, it is inspected whether the statement following the Conditional is the targeted label.



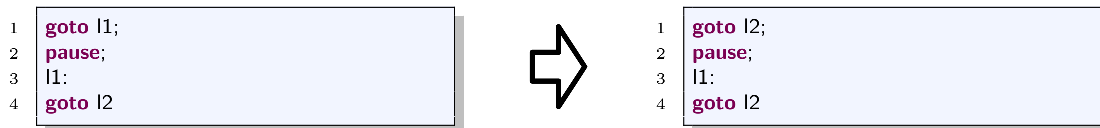
```
1 if s then
2   pause;
3   goto l1
4 end;
5 l1:
```

```
1 if s then
2   pause
3 end;
4 l1:
```

Listing 4.39. Removing Redundant goto Optimization

Removing Double Jumps

A *double jump* describes a goto targeting a label which is followed by another goto. Instead of jumping twice, the first goto might jump directly to the target of the second one. As an example, Listing 4.40 shows this issue and its optimization. Similar to the superfluous goto optimization also labels in a conditional branch which are followed by a goto are considered.



```
1 goto l1;
2 pause;
3 l1:
4 goto l2
```

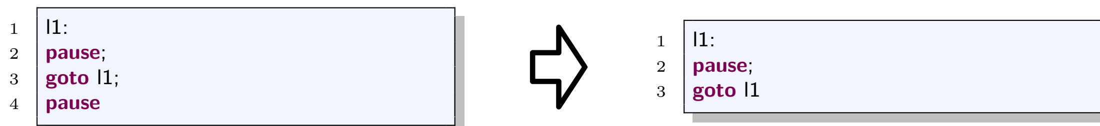
```
1 goto l2;
2 pause;
3 l1:
4 goto l2
```

Listing 4.40. Removing Double Jumps Optimization

Removing Dead Code

The Esterel to SCL transformation may produce dead code. As mentioned earlier in this section, some dead code generation can be avoided directly in the transformation rules. Further optimizations can be made in SCL by removing all code following a goto statement until a label is reached. Listing 4.41 exemplifies this. The pause statement in line 4 is unreachable since the goto in line 3 always jumps to line 1.

This optimization is a conservative dead code elimination. For example, a condition in an if-then-else statement might always evaluate to true, or even be a constant, resulting in dead code in the else branch. However, an exhaustive search would be too expensive.



```
1 l1:
2 pause;
3 goto l1;
4 pause
```

```
1 l1:
2 pause;
3 goto l1
```

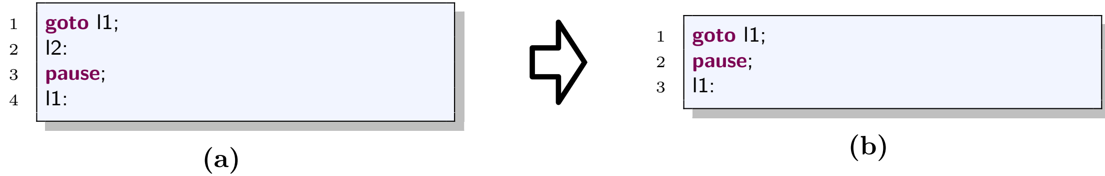
Listing 4.41. Removing Dead Code

4.4.3 Cross Dependencies of Optimizations

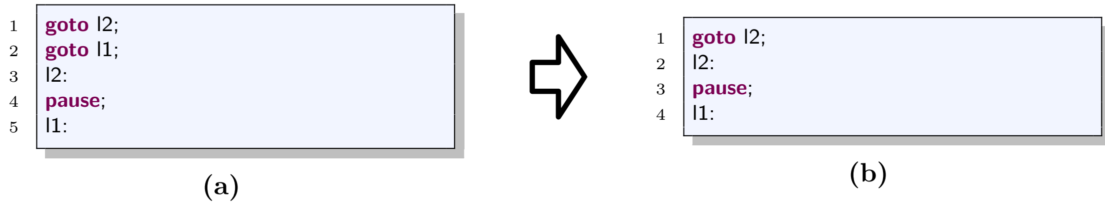
As one optimization may produce code that should be optimized by another one, the order in which they are applied matters. The presented optimizations are cross dependent, i.e., there is no clear order to apply them.

For example, in the program in Listing 4.42a there is nothing to do for the dead code elimination, as the label in line 2 and every statement thereafter is considered reachable. Applying the label optimization, the result seen in Listing 4.42b contains dead code.

Listing 4.43 illustrates that also applied in another order, the result could be further optimized. Eliminating unused labels in Listing 4.43a does nothing, as there is a goto for



Listing 4.42. The label optimization may produce dead code.



Listing 4.43. Removing dead code may produce unused labels.

every label. Removing dead code results in the program shown in Listing 4.43b, where unused labels could be removed.

It is possible to do a fix-point iteration to gather full optimization. This means, all optimizations have to be applied as long as at least one of them changes something.

Figure 4.9 illustrates the dependencies of the different optimizations. The graph reveals that the unused label optimization is cross dependent to two other optimizations. To avoid the fix-point iteration and provide a clear ordering in which the optimizations can be applied, a possible solution would be to combine the dead code elimination and the unused label optimization. Labels that are not targeted by any goto can also be seen as dead code. Further, the optimization removing redundant gotos can be extended to also check if a redundant and thus removed goto was the only one targeting a specific label. In that case, the label can be removed. Figure 4.10 illustrates the resulting, acyclic optimization chain.

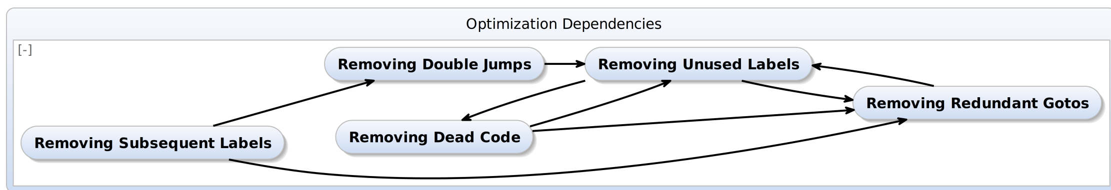


Figure 4.9. Cross Dependencies of SCL Optimizations

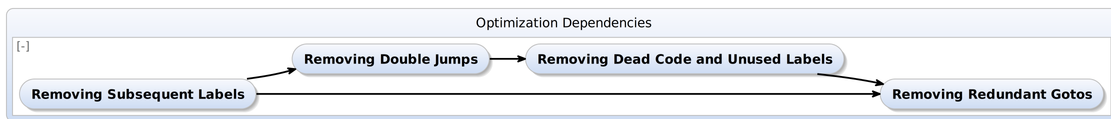
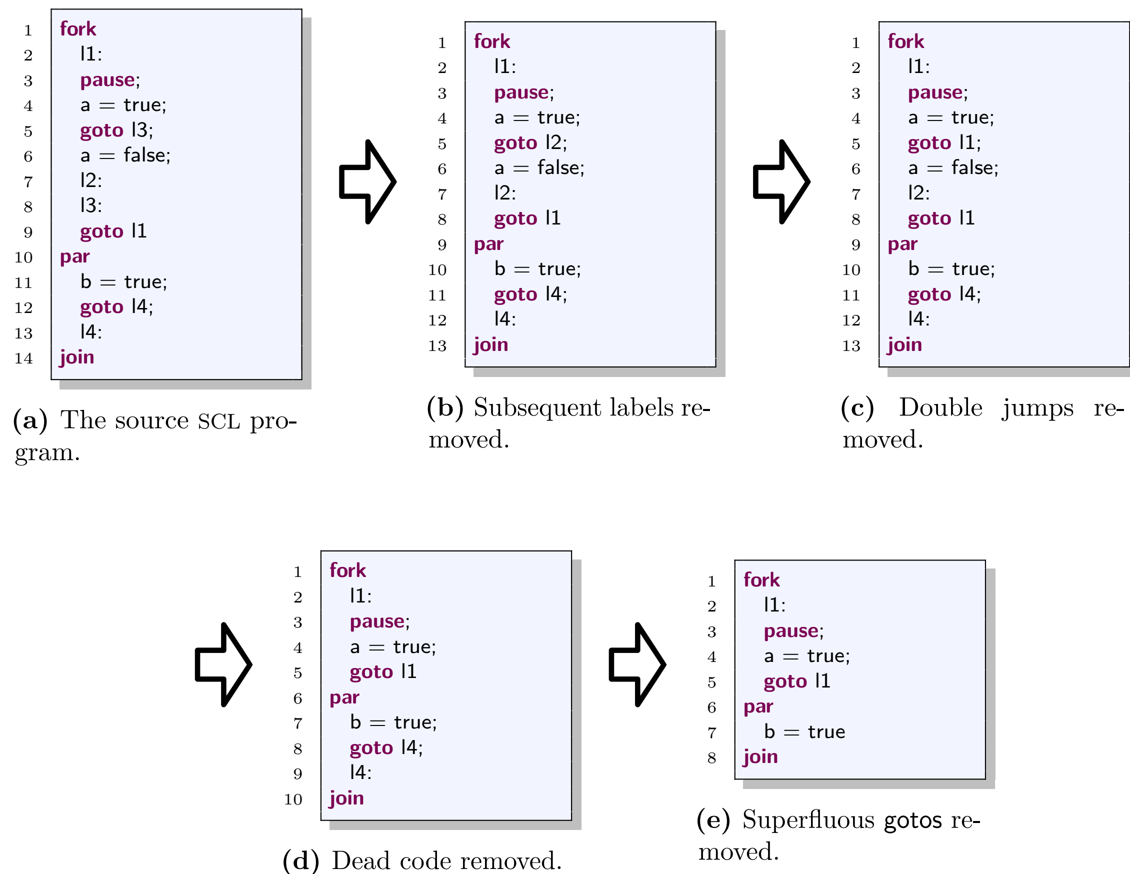


Figure 4.10. SCL Optimizations without Cross Dependencies

4.4.4 Optimization Example

Listing 4.44 shows the step-wise appliance of the optimizations as given in Figure 4.10. In the first step, the labels in line 7 and line 8 are combined and the target of the goto statement in line 5 is changed accordingly. Afterwards, in Figure 4.44c, instead of jumping from line 5 to line 7 and then to line 2, the goto is changed to directly jump to label l1. In Figure 4.44d dead code is removed, that is, everything following the unconditional goto in line 5. Finally, in Listing 4.44e, the superfluous goto in the second thread is removed.



(a) The source SCL program.

(b) Subsequent labels removed.

(c) Double jumps removed.

(d) Dead code removed.

(e) Superfluous gotos removed.

Listing 4.44. Step-wise appliance of the SCL optimizations.

Chapter 5

Implementation

In this chapter, the mandatory implementations for the transformation described in Chapter 4 are examined. At first, the integration of SCL and Esterel into KIELER is presented. Afterwards, the implementation of the transformation from Esterel to SCL is described. Finally, the realization of the regression testing is discussed.

5.1 SCL Implementation

Based on the KExpressions, SCL is implemented in KIELER using Xtext and the EMF framework as presented in Section 3.1. By adjusting the classes that Xtext generates, the SCL editor can be enriched with automatic formatting and custom validation rules. Additionally, variable scopes can be implemented.

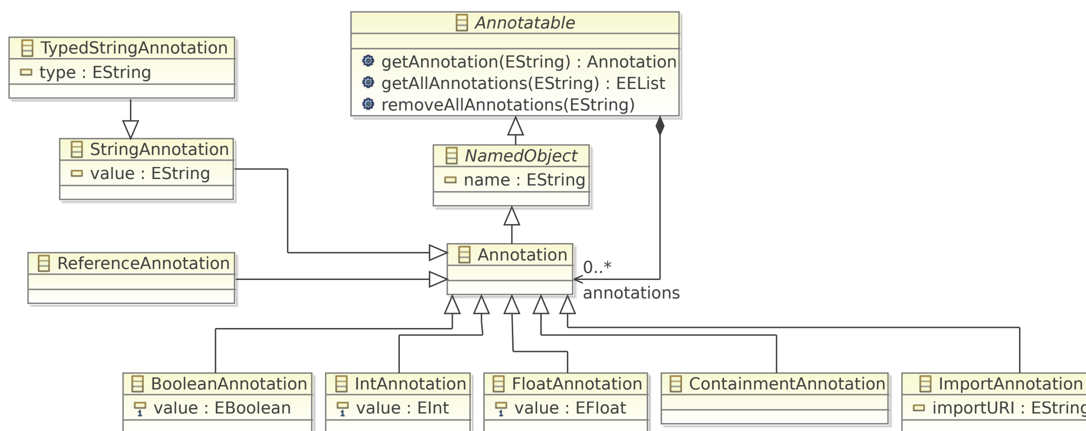


Figure 5.1. The annotations Meta-Model

5.1.1 Expression Language

With *Xbase*[1], itemis AG offers a *partial programming language* [EEK+12]. That is, Xbase is not a full language but may be used as a generic base. It provides basic constructs that are commonly used in programming languages. However, KIELER provides an own partial language to allow flexibility and a good conformance to sequentially constructive

[1]http://wiki.eclipse.org/Xbase

languages. For example, the pre operator returns the value of a signal or variable in the previous tick. It is a common construct used in synchronous languages and can be found in the KExpressions but is not a part of Xbase. Additionally, using Xbase would produce a lot of overhead since it is tightly coupled to Java and Xtend. For instance, it provides mechanisms for static methods and object instantiation, which are not beneficial for sequentially constructive languages.

Therefore, KIELER provides an own partial programming language that can be used as a base for other languages. The most general language features can be found in the annotations meta-model, which is illustrated in Figure 5.1. It provides a basic mechanism for annotating objects to store additional information for the layout, or semantic information such as properties of objects.

Extending the annotations meta-model, the KExpressions meta-model provides basic expressions, such as values or operator expressions. Besides for SCL, they are used for the *SyncCharts*, *SCCharts* and *S* language implementations, *inter alia*.

Figure 5.2 shows the KExpressions meta-model. All concrete expressions inherit from the Expression entity. For example, an OperatorExpression is an Expression with an additional attribute specifying its operator type. This should be one from the enumeration OperatorType located at the bottom of the diagram. Further, an OperatorExpression consists of zero or more subExpressions, which are again Expressions.

5.1.2 Meta-Model

Figure 5.3 shows the SCL meta-model. As indicated by the unfilled arrow, an SCLProgram is a StatementSequence with a name. Further, a StatementSequence contains zero or more Statements which is denoted by the black diamond. A Statement may be an EmptyStatement, i.e., a label, or an InstructionStatement and so on.

Further model elements are provided by the KExpressions. For example, an Assignment consists of a ValuedObject to be assigned and an Expression denoting the assigned value.

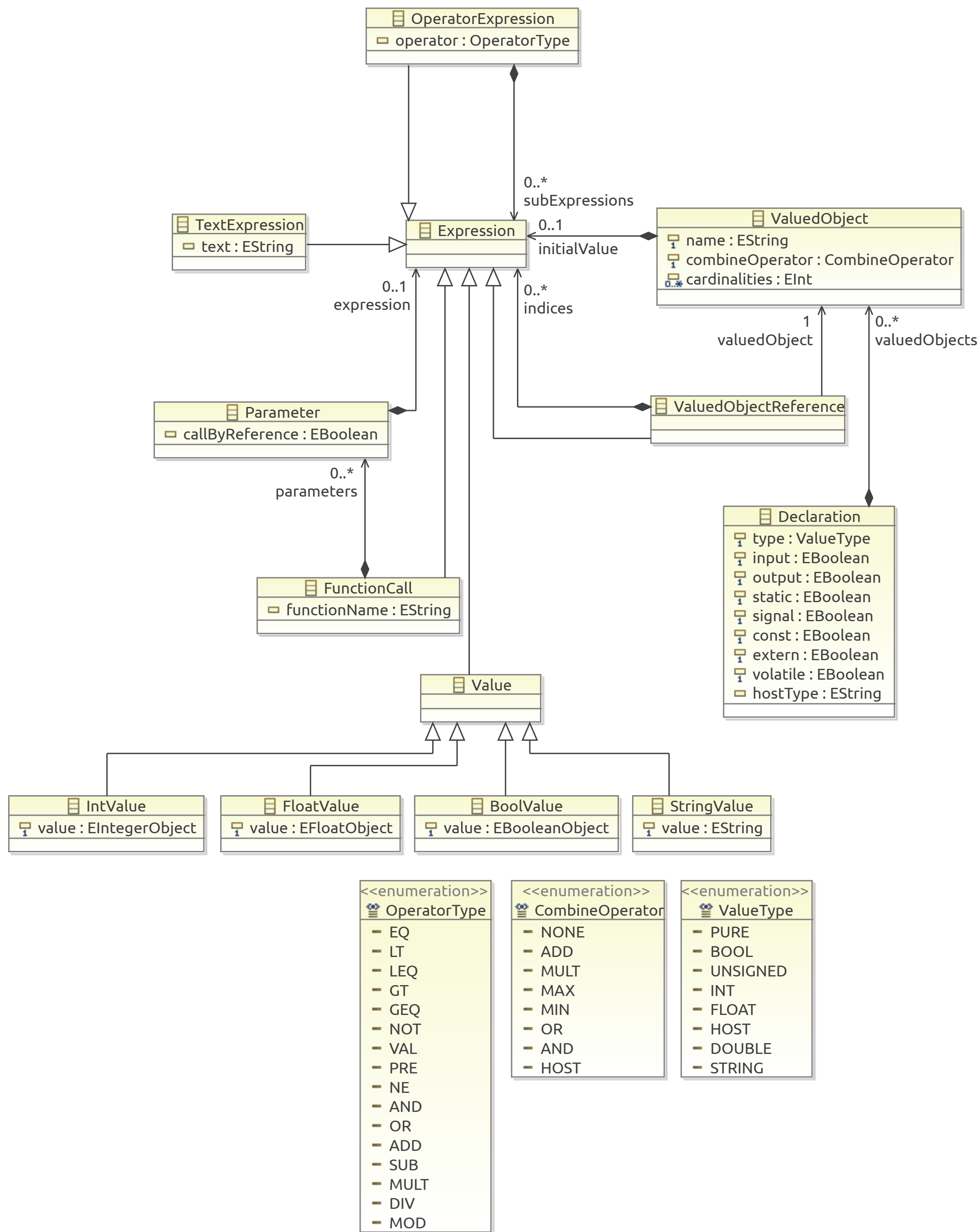


Figure 5.2. The KExpressions Meta-Model

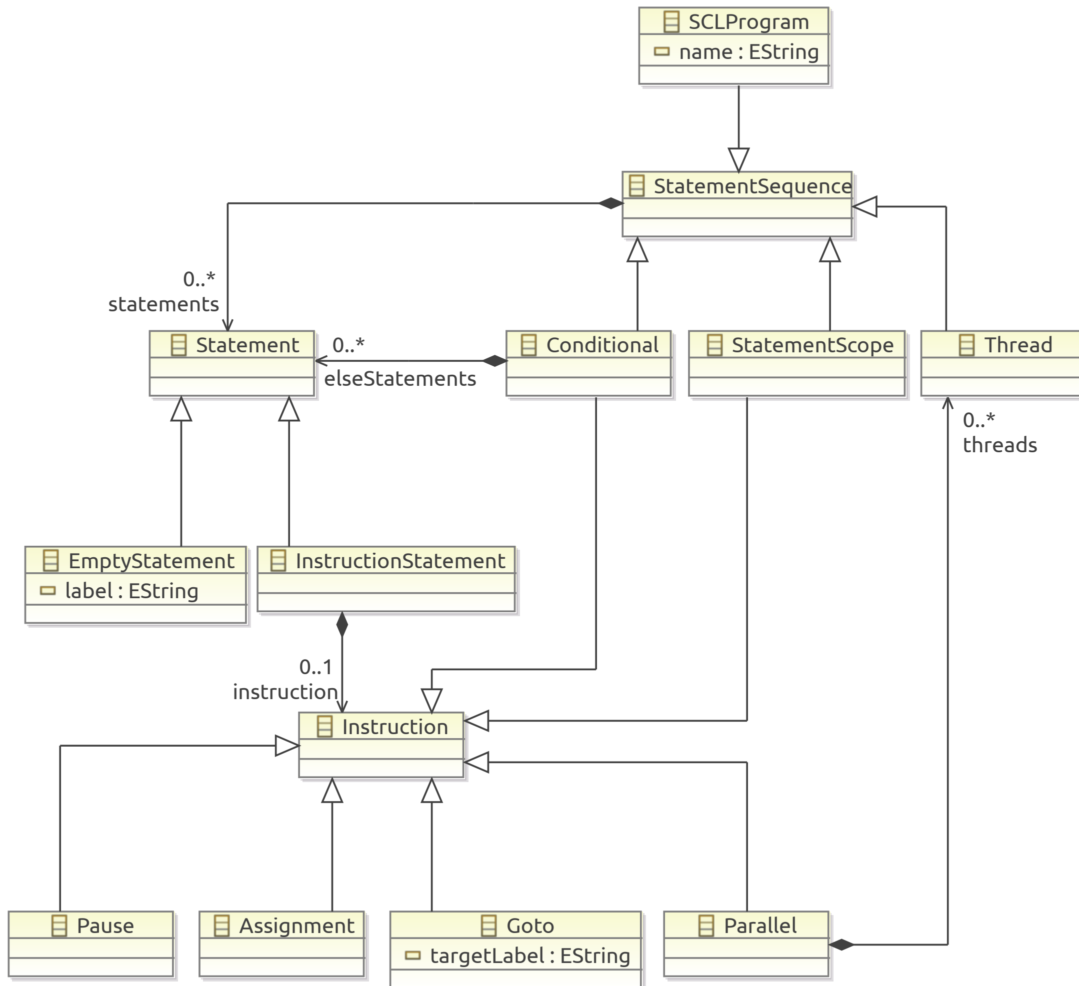


Figure 5.3. The SCL Meta-Model

5.1.3 Grammar

As described in Section 3.1.3, Xtext uses a syntax similar to EBNF. Listing 5.1 shows the definition of an SCL program. As denoted by the meta-model illustrated in Figure 5.3, the root is an SCLProgram object. The definition of an SCL program starts with the keyword module as defined in line 3 and may be annotated. Afterwards, variables are declared and stored in the declarations attribute of the SCLProgram. In line 7 and line 8, the program body is realized as a sequence of statements that are separated by semicolons.

Listing 5.2 shows another extract of the SCL grammar. A statement may either be an EmptyStatement or an InstructionStatement. EmptyStatements, which are labels, may have several annotations and an ID specifying the label's name followed by a colon. Also an InstructionStatement may be annotated. Its instruction attribute indicates which kind

```
1  SCLProgram:
2    (annotations+=Annotation)∗
3    'module' name = ID
4    (declarations+=Declaration)∗
5    '{'
6      (
7        ((statements += InstructionStatement';') | statements += EmptyStatement)∗
8        (statements += InstructionStatement statements += EmptyStatement∗)?
9      )
10   '}'
11 ;
```

Listing 5.1. An SCL program starts with the keyoword module followed by variable declarations.

of instruction it is. In case of Parallel, after the keyword fork several concurrent threads are defined, separated by the keyword par. Therefore, it is possible to start more than two threads concurrently without nesting several parallel statements. The end of the instruction is indicated by join.

The separation of labels and InstructionStatements arises from the use of the semicolon as a sequence operator. A label should be followed by a colon to signalize that the label does not do anything itself. It tags the statement that follows and allows the control to directly continue at this point. Additionally, several labels may appear consecutively and should be separated.

```
1  Statement:
2    EmptyStatement | InstructionStatement;
3
4  EmptyStatement:
5    (annotations += Annotation)∗
6    (label = ID ':');
7
8  InstructionStatement:
9    (
10     (annotations += Annotation)∗
11     instruction = (Assignment | Conditional | Goto |
12                    Parallel | Pause | StatementScope)
13   );
14
15 Parallel:
16   'fork'
17   (threads += Thread
18   ('par'
19    threads += Thread)∗)
20   'join';
```

Listing 5.2. Extract of the SCL grammar implementation in Xtext.

5.1.4 Formatting

Without a specification on how the generated SCL code should be formatted, just a string without line-breaks or additional white-spaces emerges. To allow a better comprehension of the SCL code generated from Esterel, automatic formatting, sometimes also referred to as *pretty-printing*, restructures the generated code.

For this task, Xtext allows the definition of a declarative formatter by extending the AbstractDeclarativeFormatter class. A FormattingConfig instance is used to specify how the code should be formatted. Further, an instance of a language specific GrammarAccess class, which is generated by Xtext, provides methods to find language elements, such as keywords.

Listing 5.3 shows how semicolons are formatted in SCL. At first, the findKeywords method provided by the GrammarAccess class collects all semicolons in line 1 and a for-loop iterates over them. Afterwards, the formattingConfig is instructed to set no white-space before and a line-break after semicolons in lines 2 and 3. Similarly, methods to specify indentation are used. For example, threads in the fork-par-join construct should be indented.

Listing 5.4 illustrates the utility of automatic formatting. The program in Listing 5.4a is a correct SCL program that contains no additional line-breaks or white-spaces. After invoking automatic formatting, the program in Listing 5.4b emerges.

```
1  for (Keyword semicolon: grammarAccess.findKeywords(";")) {
2    formattingConfig.setNoSpace().before(semicolon);
3    formattingConfig.setLinewrap().after(semicolon);
4  }
```

Listing 5.3. Automatic formatting for SCL should make sure that there is no white-space before and a line-break after each semicolon.

5.1.5 Code Validation

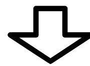
Static code validation is used to indicate programming errors in the editor. After the static analysis, the malformed code is emphasized by an error marker to allow easy debugging. For SCL, the AbstractSCLJavaValidator, which is generated automatically by Xtext, is extended in Java to detect and indicate errors.

As an example, Listing 5.5 shows the implementation of a validation method. The target label of a goto statement should be in the same scope, i.e., it is not possible to jump from one thread to another [vHMA+13]. The @Check annotation indicates that the method should be used for validation and invoked for every goto instance. In lines 3–6, the immediate surrounding thread or program is sought. Afterwards, in line7 and line 8, it is checked whether in this scope the target label of the goto is placed. In case the label is undefined, the error method is invoked. The first argument specifies the error message and the second argument states which statement should be marked as malicious. The

other arguments allow to specify an EStructuralFeature and an index. Since in this case the whole goto statement should be highlighted, they are not defined.

```
1 module formatMe input bool a;output bool b;{fork if a then b=true end;pause par l1:pause;goto l1 join}
```

(a) Before Automatic Formatting



```
1  module formatMe
2  input bool a;
3  output bool b;
4  {
5    fork
6      if a then
7        b = true
8      end;
9      pause
10   par
11     l1:
12     pause;
13     goto l1
14   join
15 }
```

(b) After Automatic Formatting

Listing 5.4. An SCL program before and after invoking auto-formatting.

```
1  @Check
2  public void checkLabelExisting(Goto gotoStatement) {
3    EObject parent = gotoStatement.eContainer();
4    while (!(parent instanceof Thread) && !(parent instanceof SCLProgram)) {
5      parent = parent.eContainer();
6    }
7    if (!labelExisting(((StatementSequence) parent).getStatements(),
8                       gotoStatement.getTargetLabel())) {
9      error("Label not in scope", gotoStatement, null, −1);
10   }
11 }
```

Listing 5.5. This static validation method for SCL should make sure that the target label of a goto statement is in the same scope.

```
1  module localScope
2  output bool a;
3  output bool b;
4  {
5    a = true;
6    b = true;
7    {
8      bool a;
9      a = false;
10     b = false
11   }
12 }
```

Listing 5.6. The global variable a should be shadowed in the statement scope since a is redefined. Thus, the output variable a is not set to false whilst b is.

5.1.6 Scoping

As an example, Listing 5.6 shows an SCL program that demonstrates the intention of scoping. At first, the global output variables a and b are set to true in lines 5 and 6. Afterwards, a new variable scope is entered in line 7. b is redefined in line 8 and subsequently a and b are set to false in lines 9 and 10. Since a was redefined, the output variable is shadowed and not effected. As a result, after the execution, the output variable a is true whilst b is false.

The scoping API provided by Xtext allows the definition of scopes, i.e., which elements can be referenced from where. As illustrated in Figure 5.3, a StatementScope is a StatementSequence. Therefore, it contains zero or more Statements. Additionally, it may have several declarations that are defined in the KExpressions meta-model seen in Figure 5.2. When statements in the StatementScope reference a variable, the declarations of the StatementScope are prioritized. That is, only if the variable is not declared in the StatementScope, outer declarations will be considered.

For SCL, the AbstractDeclarativeScopeProvider is extended and the getScope(EObject context, EReference reference) method is overridden to define custom scopes. Therefore, the context specifies for which element the scope should be calculated and the reference which EObject is referenced.

Listing 5.7 shows the implementation of the scope provider. The loop starting in line 8 iterates until the global variable declarations are reached, which is indicated by setting the boolean variable continue to false. In lines 10–12, the next surrounding variable scope is figured out by setting currentScope to its EContainer until a StatementScope or the root node is reached.

In lines 14–19, the declarations are extracted and, if the root SCLProgram is reached, setting the continue flag to false indicates to stop after this iteration. In lines 23–32, for each declared ValuedObject it is checked whether a variable with the same name has already been defined. In case the variable is undeclared, an EObjectDescription for it is added to the valuedObjectsInScope list and the name is added to the HashMap containing

```
public override IScope getScope(EObject context, EReference reference) {
  val valuedObjectsInScope = <IEObjectDescription>newLinkedList
  val declaredVariableNames = new HashSet<String>
  var currentScope = context
  var continue = true
  var EList<Declaration> declarations

  while (continue) {
    // Get the next surrounding scope
    while (!(currentScope instanceof SCLProgram) && !(currentScope instanceof StatementScope)) {
      currentScope = currentScope.eContainer
    }
    // Get the declarations
    if (currentScope instanceof StatementScope) {
      declarations = (currentScope as StatementScope).declarations
    } else {
      declarations = (currentScope as SCLProgram).declarations
      continue = false
    }

    // Check for each declared variable if it is already on the list and add an
    // EObjectDescription for it if not
    declarations.forEach [
      valuedObjects.forEach [
        if (!declaredVariableNames.contains(it.name)) {
          valuedObjectsInScope.add(
            new EObjectDescription(QualifiedName.create(it.name), it,
                                    Collections.<String, String>emptyMap()))
            declaredVariableNames.add(it.name)
        }
      ]
    ]
    currentScope = currentScope.eContainer
  }

  // Return a SimpleScope containing the list of all reachable variables
  new SimpleScope(valuedObjectsInScope)
}
```

Listing 5.7. An implementation of a scope provider for SCL.

all defined variable names in line 29. When all declared variables have been collected, in line 37, a new SimpleScope with all collected EObjectDescriptions is returned.

This scope provider is necessary for the Esterel to SCL transformation. As described in Chapter 4, some of the transformation rules make use of StatementScopes. To allow the serialization of the SCL program after the transformation, Xtext has to connect a variable used in the resulting program to an unambiguous declaration. Therefore, the scope of the variable in the specific context is needed.

In contrast to SCL, SCGs do not provide local variable scopes. Therefore, for the transformation from the SCL meta-model to the SCG meta-model, the locally declared variables have to be lifted to global declarations. To avoid the multiple declaration of the same variable name, locally defined variables may have to be renamed. Therefore, the SCLExtensions class provides a method removeLocalDeclarations(SCLProgram sclProgram)

that can be used to remove local statement scopes. If locally defined variables are already defined, they will be renamed and added to the global declarations.

5.2 Esterel Implementation

The Esterel language is, as also SCL, implemented in KIELER by using Xtext as described in Section 3.1. In contrast to SCL, Esterel is not designed as a minimal language. Thus, the meta-model is more comprehensive.

5.2.1 Expression Language

Esterel's expressions are different to the expressions provided by the KExpressions. The KExpressions are primary designated to be used in sequentially constructive languages, thus they do not provide all components needed by Esterel. For example, besides signal definitions, in Esterel also *sensors*, which are variables intended to be set by the environment, may be declared in the interface. Therefore, a different meta-model is used for the expressions in Esterel.

5.2.2 Grammar

Different grammars for Esterel exist [Ber00, PBEB07]. Even though these grammars are commonly written in EBNF, which is similar to the notation in Xtext, they cannot be adopted directly. Since the ANTLR parser generator used by Xtext produces LL-parsers, *left-recursion* has to be eliminated that is used a lot in the existing grammars for Esterel.

A left-recursive rule means that the non-terminal symbol on the left-hand side of the rule appears on the left-most position of the right-hand side [Pow99]. Therefore, a left-recursive grammar contains a rule of the form

$$A \rightarrow A\alpha_1 \mid \ldots \mid A\alpha_n \mid \beta_1 \mid \ldots \mid \beta_m,$$

where each α_i and β_j may be a composition of non-terminal and terminal symbols, where no β_j starts with A.

LL-parsers are not able to parse left-recursive grammars since they read from left to right. Therefore, it cannot be decided which of the rules should be applied. A possible solution to remove left-recursion is *left-factorization*. An equivalent grammar to the left-recursive rule denoted above without left-recursion would be

$$\begin{aligned} A &\rightarrow \beta_1 A' \mid \ldots \mid \beta_m A' \\ A' &\rightarrow \alpha_1 A' \mid \ldots \mid \alpha_n A' \mid \varepsilon. \end{aligned}$$

```
1 ModuleInterface:
2     (intSignalDecls+=InterfaceSignalDecl
3     | intTypeDecls+=TypeDecl
4     | intSensorDecls+=SensorDecl
5     | intConstantDecls+=ConstantDecls
6     | intRelationDecls+=RelationDecl
7     | intTaskDecls+=TaskDecl
8     | intFunctionDecls+=FunctionDecl
9     | intProcedureDecls+=ProcedureDecl)+;
```

Listing 5.8. Left-recursion can be avoided by using the list assignment mechanism of Xtext.

Even though left-recursion can be removed this way, additional syntax is introduced. Therefore, Xtext provides list assignments. As an example, Listing 5.8 shows the list assignment mechanism for the interface of an Esterel module. The plus-equal sign indicates that the left argument is interpreted as a list. Zero or more elements of the type given on the right side of the equation may be added to this list.

5.3 Esterel to SCL Transformation

As stated in Section 1.3, KIELER is a collection of Eclipse plug-ins. For making use of this modular concept and providing a good integration into KIELER, also the implementation of the Esterel to SCL transformation is implemented as an Eclipse plug-in, namely de.cau.cs.kieler.esterel.scl.

For the actual M2M transformation classes Xtend is used, as introduced in Section 3.1.4. Figure 5.4 shows an overview of the different components of the transformation plug-in and how they communicate. Whilst most transformation rules are implemented in the EsterelToSclTransformation class, in the EsterelToSclExtensions class some helper methods and abbreviations are located. The class EsterelDeclarationsTransformation provides methods to transform Esterel interfaces and local declarations to SCL and the TransformExpressions class allows the transformation between the different meta-models that provide the expression languages of Esterel and SCL.

5.3.1 Main Transformation Class

The EsterelToSclTransformation class extends the abstract class Transformation in order to be used by KiCo, introduced in Section 3.2.3. To invoke the transformation to SCL, two methods can be used as entry points. Both fulfill the signature requirements of KiCo and are presented in Listing 5.9. Whilst the first method performs the transformation without applying optimizations, as described in Section 4.4, the second method does. The first argument of type EObject, which is the root class of all EMF model objects, is the source Esterel program. The returned EObject is the corresponding SCL program. The KielerCompilerContext provides additional information from KiCo.

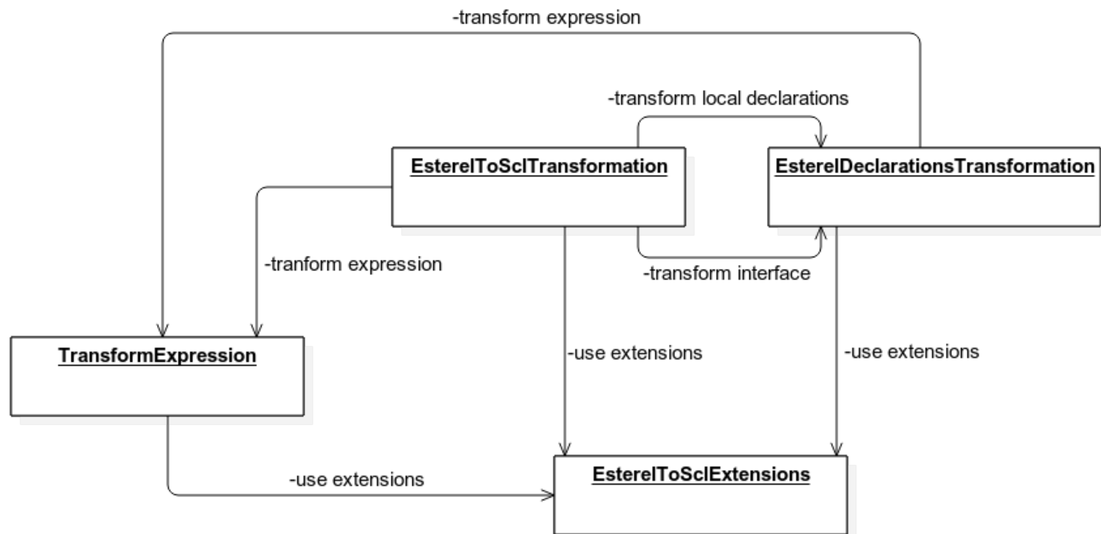


Figure 5.4. Diagram showing the communication between the different components of the Esterel to SCL transformation implementation.

The transformProgram method starts the actual transformation by initializing global variables and calling further transformation methods. The Esterel module body is transformed by several *dispatched* methods. Dispatched methods are a feature provided by Xtend and allow to define several methods with the same name but different argument types. Depending on the arguments, the matching method is chosen at run-time. Hence, for each Esterel statement, a dispatched transformStatement method handles the actual transformation.

As an example, Listing 5.10 shows how the transformation of the Esterel loop statement is implemented, as described in Section 4.2.5. In the Esterel meta-model, a loop may have different end entities. If the end is an instance of the EndLoop entity, the loop is a simple loop. Otherwise, if the end is an instance of the LoopEach entity, the loop is a loop each statement.

```
override EObject transform(EObject eObject, KielerCompilerContext contex) {
  optimizeTransformation = false;
  return transformProgram(eObject as Program) as EObject
}

def EObject transformOpt(EObject eObject, KielerCompilerContext contex) {
  optimizeTransformation = true;
  return transformProgram(eObject as Program) as EObject
}
```

Listing 5.9. Two methods can be called by KiCo to invoke the transformation either with or without optimization.

```
1  def dispatch StatementSequence transformStatement(
2    Loop loop, StatementSequence targetStatementSequence) {
3    if (loop.end instanceof LoopDelay) {
4      return handleLoopEach(loop, targetStatementSequence)
5    }
6
7    val loopStartLabel = createNewUniqueLabel
8    targetStatementSequence.addLabel(loopStartLabel)
9    transformStatement(loop.body.statement, targetStatementSequence)
10   targetStatementSequence.addGoto(loopStartLabel)
11
12   targetStatementSequence
13 }
```

Listing 5.10. Xtend Implementation of the loop Transformation

Accordingly, in line 3, it is checked whether the loop statement is a loop each. In that case, the responsible handler is called. Otherwise, it is a normal loop statement and in line 7 a new, unused label is created and added to the SCL StatementSequence in line 8. Afterwards, in line 9, the body of the statement is transformed. In line 10, a goto to the former added label is inserted and in line 12, the transformed statement is returned.

Class Variables

In the following, the major variables of the EsterelToSclTransformation class are presented.

String *currentThreadEndLabel*
: Since some transformation rules use the artificial gotoj statement, as defined in Section 4.1, the label at the end of the transformed thread may be needed. During the transformation, *currentThreadEndLabel* is always set to the label at the end of the currently transformed thread.

Multimap<String, String> *labelToThreadMap*
: To actually transform the gotoj statement, each label has to be related to a specific thread. The unique label at the end of each thread can be used to check whether a label targeted by a gotoj is reachable. Therefore, the key for the *labelToThreadMap* is the label at the end of the currently transformed thread, the dedicated values are the labels within this thread.

LinkedList<Pair<String, ValuedObject>> *signalToVariableMap*
: Each signal in Esterel has to be connected to a variable in SCL. For this purpose, a LinkedList is used. It associates a string, i.e., a signal name, with a variable. When a new scope is entered, the locally declared signals are attached at the end of the list and when the scope is left they are removed. This allows to respect local declarations and shadowing declarations with the same name on higher hierarchical layers. The ValuedObject, i.e., the SCL variable, associated with a name found as

far at the end of the list as possible represents the ValuedObject in the currently transformed scope.

HashMap<ValuedObject, ValuedObject> *signalToValueMap*
Valued signals have – besides a presence state – a value. As described in Section 4.2.3, one variable indicates the presence state and another the associated value in SCL. The signalToValueMap allows to find a variable representing the value by the variable representing the presence state.

HashMap<ValuedObject, ValuedObject> *signalToNeutralMap*
As stated in Section 4.2.3, an additional variable is introduced for resolution functions. This variable should hold the neutral element of the variable's resolution function at the start of each tick. The *signalToNeutralMap* allows to find the variable that holds the neutral element of this resolution function by the variable representing the presence state of the signal.

Stack<(StatementSequence)=>StatementSequence> *pauseTransformation*
Many of the rules presented in Section 4.2 rely on inserting code before or after pause statements. To avoid searching for every pause in the transformed statement, the lambda function mechanism of Xtend is used to allow the correct transformation of pauses right when they appear. Each Esterel statement that invokes transformations on pause statements pushes a lambda function on the stack which inserts the additional statements. When a pause is transformed, the functions from the stack are applied in the given order. When the body of a statement is transformed, the corresponding lambda functions are removed from the stack.

Stack<(StatementSequence)=>StatementSequence> *joinTransformation*
Similar to the transformations on pause statements, some of the rules given in Section 4.2 insert additional statements after joins. Again, a stack structure is used to assert that the functions are applied in the right order.

5.3.2 Interface Transformation

The class EsterelDeclarationsTransformation provides the functionality to transform an Esterel interface to SCL declarations. This covers the transformation of signals, constants and sensors. The resulting SCL variables are directly inserted into the class variables of the main transformation class and are added to the target SCL program. Additionally, this class provides a method to transform local variable declarations.

5.3.3 Expression Transformation

The class TransformExpression is used to transform Esterel expressions to SCL expressions. Similar to the transformation of statements, dispatched methods are used. Whilst many expressions can be translated straightforwardly, e.g., operator expressions, some differences

have to be handled. For example, SCL expressions do not provide the double data type or unsigned integers. Also references to, among others, signals and sensors in Esterel expressions have to be transformed to variable expressions in SCL.

5.3.4 Extensions

The class EsterelToSclExtension provides helper methods for the actual transformation to make the code more comprehensible. For example, to allow the usage of gotoj, which is not an actual SCL statement, the extensions class provides the method in Listing 5.11. In line 3, by using the labelToThread map, it is checked if the currently transformed thread contains the targeted label. If so, a statement with a goto to the label is returned, if not, the created goto targets the thread end.

```
1 def createGotoj(String label, String currentThreadEndLabel,
2                 Multimap<String, String> labelToThreadMap) {
3   if (labelToThreadMap.get(currentThreadEndLabel).contains(label)) {
4     return createGotoStm(label)
5   } else {
6     return createGotoStm(currentThreadEndLabel)
7   }
8 }
```

Listing 5.11. Xtend Implementation of gotoj

The extension methods provided by Xtend allow to add new methods to existing types without modifying them directly. This mechanism is used to implement abbreviations for frequently used cases. For example, for adding an Instruction to a StatementSequence, it has to be wrapped into an InstructionStatement first, as denoted by the SCL meta-model presented in Section 5.1.2. Listing 5.12a shows how this has to be done with the methods generated by the EMF framework. A new InstructionStatement, which is instantiated by the SclFactory, is added to the list of statements of the StatementSequence in line 1. Afterwards, a new goto is added to this InstructionStatement in line 3 and the target label is set accordingly in line 4.

Listing 5.12b shows how an extension method addGoto can be used as an abbreviation. Even though this method is not defined in the StatementSequence class, it can be called in the same way. Listing 5.13 shows how addGoto is implemented in the EsterelToSclExtensions class. As the StatementSequence is the first argument, it can be used as an extension method for a StatementSequence with only the second argument given as a parameter.

Additionally, this class contains a method to analyze whether an Esterel program may terminate or not. This information is used for the optimized transformation as described in Section 4.4.1. Further, a function to check whether a trap or a weak abort statement may terminate instantaneously is provided, as described in Section 4.2.6 and Section 4.2.10.

For many transformation rules unused labels are necessary. Therefore, a method

that creates unique, i.e., not yet in use, labels by numbering them is located in the EsterelToSclExtensions class.

5.4 SCL Optimizations

The SCL optimizations presented in Section 4.4 are implemented in the de.cau.cs.kieler.scl plug-in in the SCLExtensions class. All optimizations are implemented using Xtend. As an example, Listing 5.14 shows the implementation of the dead code elimination as introduced in Section 4.4.2.

The loop starting in line 3 iterates over every goto statement in the StatementSequence that should be optimized. As gotos are instructions that are wrapped into an Instruction-Statement, in line 4 the actual statement is extracted. In line 5, the StatementSequence that contains the InstructionStatement is figured out. Afterwards, the loop in line 8 iterates through this StatementSequence, starting from the index where the goto is, and stores every statement into the toDelete list. When a label is reached, the following code is considered as reachable, since a goto may target it. Accordingly, the loop is left and the statements before the label are deleted, as they are not reachable.

```
1 statementSequence.statements.add(
2   SclFactory::eINSTANCE.createInstructionStatement => [
3     instruction = SclFactory::eINSTANCE.createGoto => [
4       targetLabel = label
5     ]
6   ]
7 )
```

(a) Without Extension Methods

```
1 statementSequence.addGoto(label)
```

(b) With Extension Methods

Listing 5.12. Adding a goto to a StatementSequence

```
1 def addGoto(StatementSequence statementSequence, String label) {
2   statementSequence.statements.add(
3     SclFactory::eINSTANCE.createInstructionStatement => [
4       instruction = SclFactory::eINSTANCE.createGoto => [
5         targetLabel = label
6       ]
7     ]
8   )
9 }
```

Listing 5.13. Definition of the addGoto Extension Method in Xtend

```
def StatementSequence removeUnreachableCode(StatementSequence statementSequence) {
  val toDelete = <Statement>newLinkedList
  for (goto : statementSequence.eAllContents.toList.filter(typeof(Goto))) {
    val statement = goto.eContainer
    val parent = statement.eContainer as StatementSequence
    var index = parent.statements.indexOf(statement)
    var noLabel = true
    while (parent.statements.size > index + 1 && noLabel) {
      val nextStatement = parent.statements.get(index + 1) as Statement
      if (nextStatement instanceof EmptyStatement) {
        noLabel = false
      } else {
        toDelete.add(nextStatement)
      }
      index = index + 1
    }
  }
  toDelete.forEach[ it.remove ]

  statementSequence
}
```

Listing 5.14. Xtend implementation of the dead code elimination.

5.5 KART Regression Testing

As described in Section 3.2.2, KART is a framework integrated into KIELER that can be used for regression testing. Figure 5.5 illustrates how traces are validated. For a set of Esterel programs that is included in the CEC[2] and covers a wide spectrum of different statements and program structures, traces have been created by the compilations chain via SCL and were validated by the CEC.

Figure 5.6 shows the regression testing process. When executed, KART simulates the Esterel programs in the test suite via the transformation to SCL and compares the outputs with the given traces. For the simulation, the same inputs as prescribed by the validated traces are used.

Therefore, the class EsterelToSclAutomatedTests extends KiemAutomatedJUnitTest. Most of the work, for example, executing the test cases and comparing the traces, is done by KART. The EsterelToSclAutomatedTests class only has to specify where these test cases are located, which extension the files have and how to execute them. Subsequently, the KART framework simulates the test programs – in this case by a prior transformation to SCL – and compares the outputs to those found in ESO files with the same names as the source Esterel files.

[2]http://www.cs.columbia.edu/~sedwards/cec

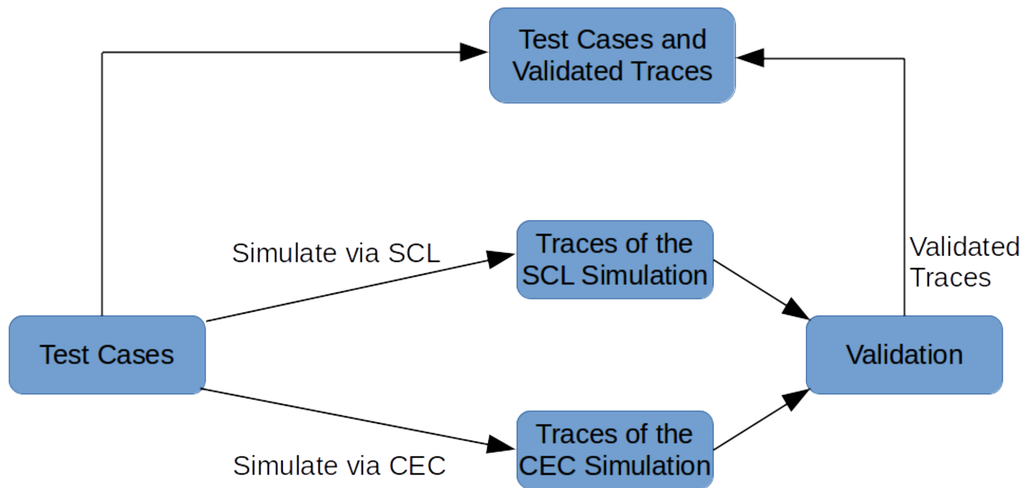


Figure 5.5. The test cases are simulated via SCL and the CEC. The traces are validated against each other and create the regression testing suite together with the corresponding test cases.

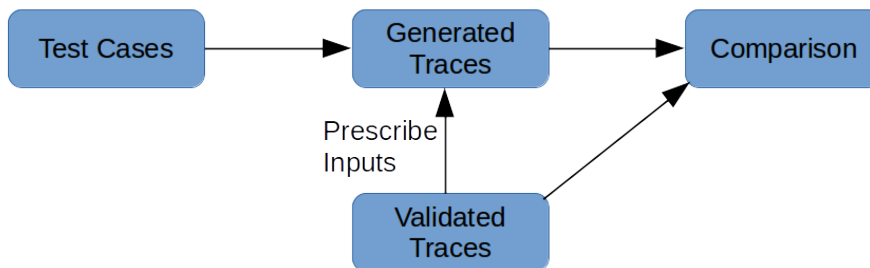


Figure 5.6. For the regression testing, every Esterel program in the testing suite is simulated with the same inputs as in the validated trace. The resulting trace is compared to the validated one.

Chapter 6

Evaluation and Experimental Results

This chapter evaluates the transformation rules and the underlying implementation. After presenting the test set-up, the performance of the Esterel to SCL transformation and the compilation via SCL to target code is tested. Afterwards, the size of the SCL representation is compared to the size of the source Esterel program and the extend in which the optimizations affect it are evaluated. The generated target code is compared to the target code generated by the CEC in both, size and performance. Finally, some examples on how the SC MoC can be adopted for Esterel are presented.

6.1 Test Set-Up

All tests were executed on a system with an Intel Core™ i7-4510, 2.00 GHz quadcore with 8GB RAM running on a 64-bit Ubuntu 14.04. For validation and testing, a subset of the test cases that are include in the CEC[1] were used. They cover a broad spectrum of scenarios and size. Table 6.1 shows an overview of the test cases and gives a short description.

6.2 Compile Performance

This section examines, how efficient the transformation of Esterel to SCL is and how the compilation to target code compares to the CEC. Additionally, the performance of the optimizations is evaluated.

6.2.1 Esterel to SCL

Figure 6.1 shows the efficiency of the transformation from Esterel to SCL, with and without optimizations, depending on the source Esterel program size. The optimized transformation is more expensive than the unoptimized one, especially for larger programs. Since the duration does not increase with the amount of statements in every case, it can be seen that the time needed for the transformation does not only depend on the size of the source Esterel program. This derives from the different program structures. Large programs that use only simple statements may be transformed more efficiently than smaller ones with expressive and deeply nested statements.

[1] http://www.cs.columbia.edu/~sedwards/cec

Table 6.1. Extract of the test cases for the evaluation with a short description.

Name	Statements	Description
test-pause1	1	Simple program with just one `pause` statement.
test-suspend9	4	Tests the `suspend` statement in combination with parallel execution.
test-abro	7	The *Hello World* of Esterel as described in Section 1.1.1.
test-trap12	10	A `trap` statement with an exception handler.
test-trap15	15	A program with nested `trap` statements.
test-weakabort2	38	Long `weak abort` statement with different cases.
test-mult4	56	Deeply nested parallel, preemptive and loop statements.
test-ms-rs-flipflop	108	Implementation of an MS-RS-flipflop.
test-ww	212	A wristwatch implementation developed by G. Berry [Ber88].
test-mejia2	239	The acyclic variant of a popular test case in the *Estbench* [Est], a benchmark suite for Esterel.

6.2.2 Esterel to Target Code

In the following, the compilation of Esterel to C via SCL and subsequent code generation is compared to the CECs code generation by the PDG approach integrated into KIELER. Therefore, every program in the test suite is transformed from Esterel to C target code by each compiler 10 times and the average compilation time is calculated.

Figure 6.2 shows the result. The transformation to SCL and subsequent compilation by the netlist approach takes much less time. Besides the modern technology that exploits the test system's resources more efficiently, also the integration into KIELER potentially slows down the CEC. However, the CEC seems to scale better as the compilation time stays relatively stable also for larger programs.

6.3 Scalability

Since SCL is a minimal language, the transformation of single Esterel statements may result in many SCL statements. In this Section, the code size of the resulting SCL code is compared to the size of the source Esterel code. Additionally, the influence on the SCL code size by the optimizations presented in Section 4.4 is evaluated and the target code size is compared to the size of the target code generated by the CEC.

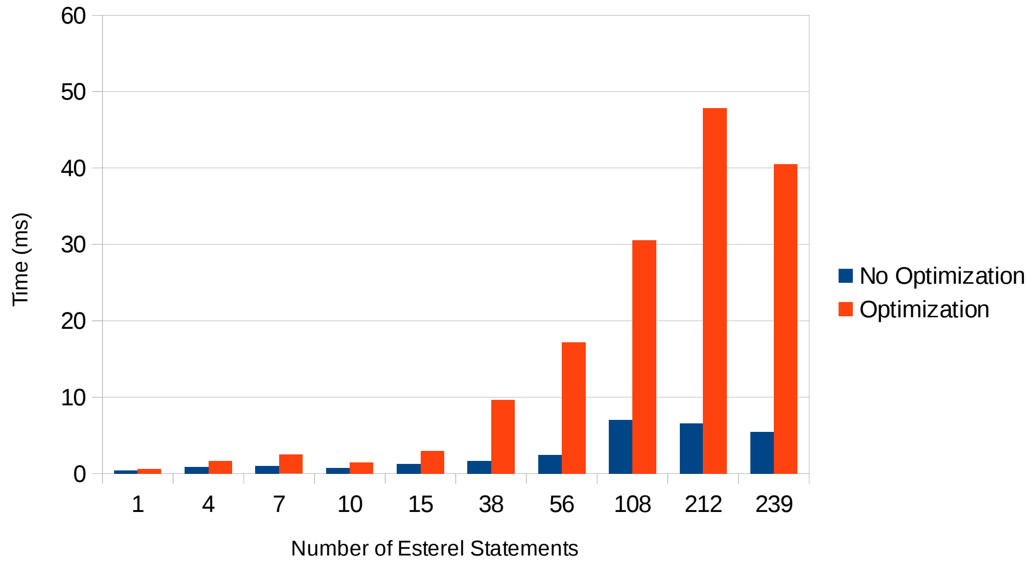


Figure 6.1. Results for the performance analysis of the optimized and the unoptimized transformation.

6.3.1 SCL Code Size

Table 6.2 compares the code size of the source Esterel program and the target SCL program. In average, in SCL 4.99 times more statements are necessary to express an Esterel program. It can be seen that there is no direct connection between the source Esterel program size and the scale factor. For example, the test-ms-rs-flipflop program makes heavy use of the module instantiation mechanism provided by Esterel. As described in Section 4.2.13, SCL does not have a comparable statement. When transformed to SCL, instantiated modules are just copied. Hence, for test-ms-rs-flipflop the scale factor is above average.

6.3.2 Optimizations

The optimizations presented in Section 4.4 are intended to remove redundancy in the generated SCL code. Figure 6.3 shows the results for the test cases presented in Table 6.1. The amount of statements in the target SCL program is reduced by 14.65% in average.

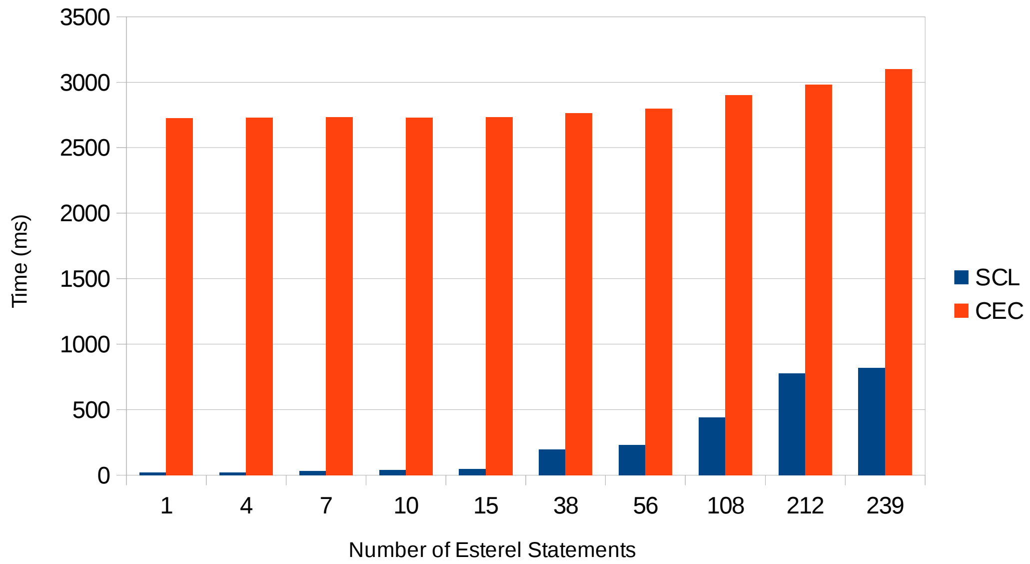


Figure 6.2. Results for the compiler performance comparison of the compilation via SCL and the CEC in KIELER.

Table 6.2. Amount of statements in Esterel programs and the resulting SCL programs.

Test Case	Number of Esterel Statements	Number of SCL Statements	Scale Factor
test-pause1	1	8	8.00
test-suspend9	4	25	6.25
test-abro	7	38	5.43
test-trap12	10	25	2.50
test-trap15	15	47	3.13
test-weakabort2	38	216	5.68
test-mult4	56	295	5.27
test-ms-rs-flipflop	108	663	6.14
test-ww	212	854	4.03
test-mejia2	239	777	3.25
			Ø4.99

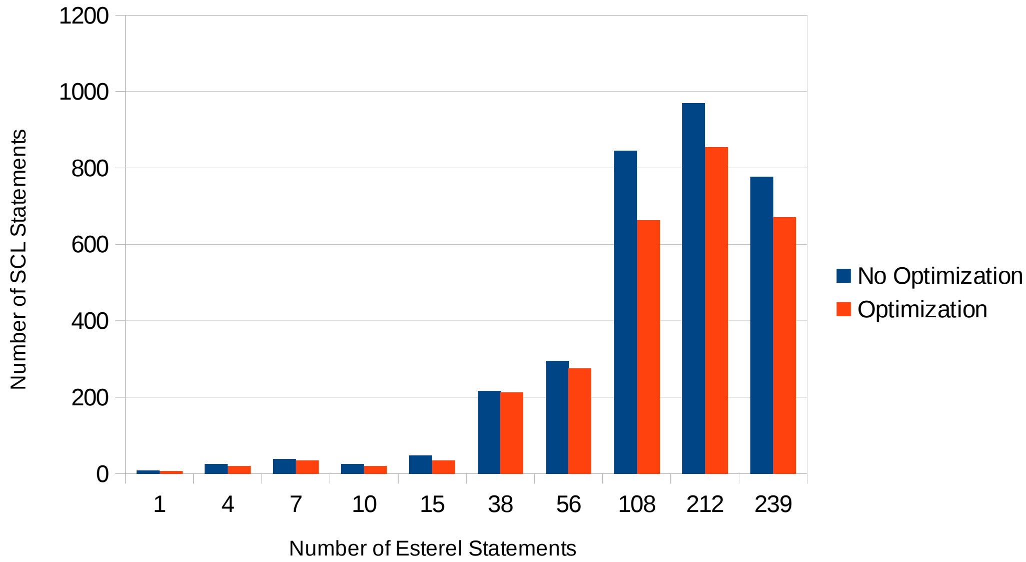


Figure 6.3. Amount of statements after transforming Esterel programs to SCL with optimizations and without.

6.3.3 Target Code

Figure 6.4 shows the result of the comparison between the size of the generated target code produced by the code generation via SCL with optimizations and the PDG approach used by the CEC. Therefore, both compilers have produced C code with a similar format. The compilation via SCL produces more lines of code. Since the Esterel program is transformed to SCL, the structure of the originating Esterel program is not considered in the subsequent compilation. In contrast, the CEC especially targets on generating code from Esterel.

6.4 Target Code Execution Time

Figure 6.5 shows the comparison between the performance of the code generated via SCL and the CEC. Therefore, each of the generated tick functions are executed one million times with pseudo-random inputs.

The CEC generates more efficient code for every test case. This derives from the different compilation approaches. As described in Section 2.4.2, the compilation via PDGs produces C code that maps control dependency in the intermediate format to control dependencies in the C code. This seems to produce more efficient code than considering data dependencies.

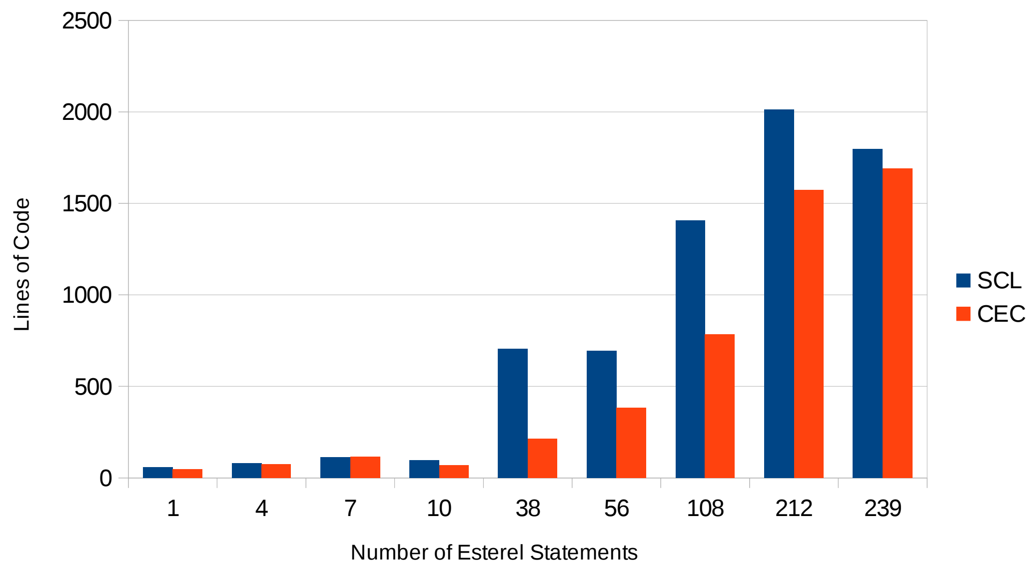


Figure 6.4. Lines of code generated by the compilation to C via SCL and the CEC.

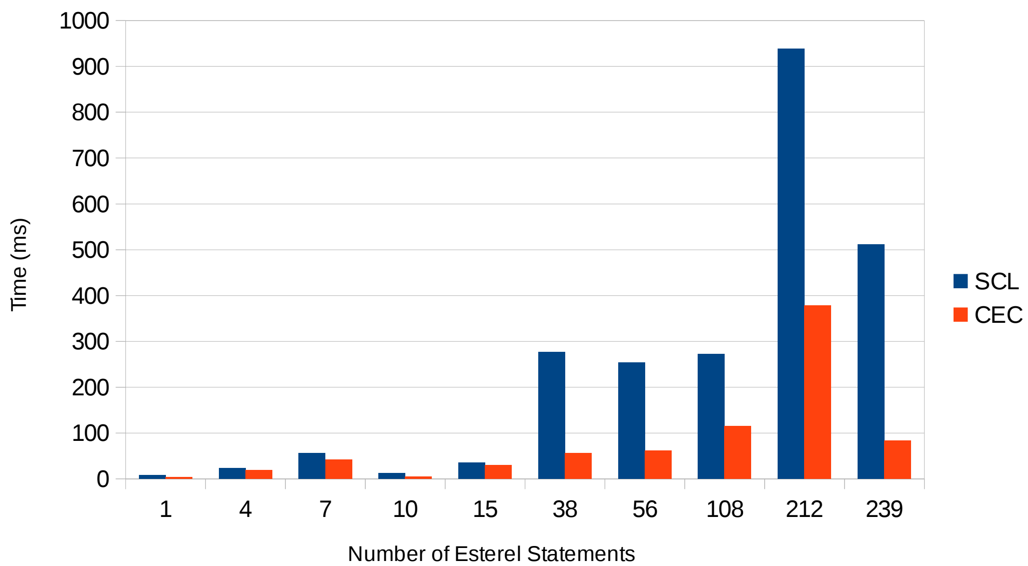


Figure 6.5. Comparison of the performance for compiled Esterel programs via SCL and by the CEC.

It can be seen that the efficiency of the generated code does not only rely on the source Esterel program size. Also the structure of the program has a major influence on the code generation progress. For example, a program that uses only simple statements, such as `pause`, may be compiled to more efficient code than a program with expressive and deeply nested statements.

6.5 Sequentially Constructive Esterel

This section evaluates how the SC MoC can be applied to Esterel, i.e., if the transformation rules presented in Chapter 4 preserve sequentially constructive structures. This would allow to use the vast syntax of Esterel to write sequentially constructive programs. In particular, some of the restrictions of Esterel can be eased and a higher amount of programs could be compiled. For example, the Esterel program in Listing 6.1a is incorrect, however, if Esterel uses the SC MoC, a deterministic execution would be possible.

For the evaluation, different programs, which are invalid in Esterel, but correct in the SC MoC, are transformed to SCL. The resulting SCL program is tested for schedulability.

6.5.1 Sequential Ordering of Statements

At first, a simple, invalid Esterel program is examined. The program given in Listing 6.1a is invalid since the presence state of `s` cannot be determined in a constructive way, as described in Section 1.1.2.

```
1  module simpleExample:
2  output s;
3
4  present s then
5    pause
6  end;
7  emit s
8  end module
```

(a) Esterel

```
1  module simpleExample
2  output bool s;
3  bool f_term = false;
4  {
5    fork
6      l1:
7      s = false;
8      if ! f_term then
9        pause;
10       goto l1;
11     end;
12   par
13     if s then
14       pause;
15     end;
16     s = s | true;
17     f_term = true;
18   join;
19 }
```

(b) SCL

Listing 6.1. The transformation of a simple invalid Esterel program to a valid SCL program.

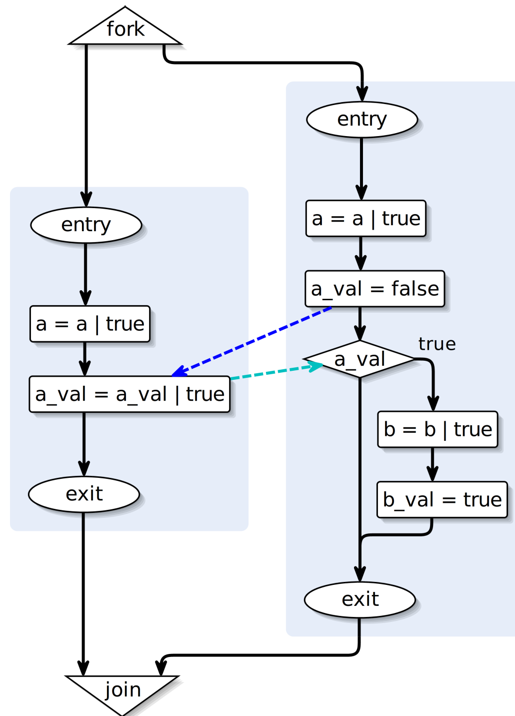


Figure 6.6. Extract of the SCG for the IUR Program

In the SC MoC, the sequential ordering of statements is considered, thus, s is not present at the start of the tick. Therefore, the pause statement in line 5 is not executed and s is emitted in line 7. Using the transformation rules, the SCL program in Listing 6.1b emerges, which is correct in terms of the SC MoC. The thread in lines 6–11 sets a to false at the beginning of the first tick. Afterwards, s is read in line 13 and the pause is not executed. Finally, s is set to true in line 16 and the program terminates. Thus, the resulting SCL program shows the expected behavior for the Esterel program with sequentially constructive semantics.

6.5.2 The Initialize-Update-Read Paradigm

The Esterel program in Listing 6.2a illustrates the initialize-update-read order when scheduling concurrently running threads as described in Section 1.2. The program is invalid, since a is emitted with different values in the same tick without a defined resolution function. The resulting SCL program in Listing 6.2b is schedulable, as the initialize-update-read paradigm gives a clear schedule for the concurrent statements. Figure 6.6 illustrates this as a shortened SCG with overlaid dependencies.

At first, the variable representing the signal's value a_val is set to false by an absolute write since the initialize-update dependency requires absolute writes to be scheduled before relative writes. Subsequently, a_val is set to true by the relative write. Finally, as

depicted by the update-read dependency, a_val is read and therefore b is emitted and b_val is set to true.

```
1  module IUR:
2
3  output a : boolean, b : boolean;
4
5  [
6    emit a(false);
7    if ?a then
8      emit b(true)
9    end
10 ||
11   emit a(?a or true)
12 ]
13 end module
```

(a) IUR in Esterel

```
1  module IUR
2  output bool a;
3  output bool a_val;
4  output bool b;
5  output bool b_val;
6  bool f_term = false;
7  {
8    fork
9      l4:
10     a = false;
11     b = false;
12     if ! f_term then
13       pause;
14       goto l4
15     end
16   par
17     fork
18       a = a | true;
19       a_val = false;
20       if a_val then
21         b = b | true;
22         b_val = true
23       end
24     par
25       a = a | true;
26       a_val = a_val | true
27     join;
28     f_term = true
29   join
30 }
```

(b) IUR in SCL

Listing 6.2. The Transformation of IUR from Esterel to SCL

6.5.3 The ABO Example

Finally, ABO is transformed, as presented in Section 1.2. Listing 6.3a shows a possible Esterel implementation of ABO. Due to the multiple emission of O1 with different values in the same tick in lines 9, 12 and 14, this program is invalid and therefore refused by Esterel compilers. In contrast, the resulting SCL program in Listing 6.3b can be scheduled by the SCL compiler and executed with the expected behavior since the sequential ordering gives a clear schedule.

6.5.4 Results

Even though the examples do not provide a complete analysis of how the SC MoC can be adopted to Esterel by using the presented transformation rules, for most cases it seems possible. The basic concepts, i.e., considering the sequential order of statements and respecting the initialize-update-read protocol, appear to be preserved by the transformation rules.

Thus, compiling Esterel via SCL eases the restrictions of classical Esterel compilers by allowing the compilation of a larger set of programs without introducing non-determinate concurrency.

```
module ABO:

input A : boolean, B : boolean;
output O1 : boolean, O2 : boolean;

[
  await immediate A;
  emit B(true);
  emit O1(true)
||
  await B;
  emit O1(true)
];
emit O1(false);
emit O2(true)

end module
```

(a) ABO in Esterel

```
module ABO
input bool A;
input bool A_val;
input bool B;
input bool B_val;
output bool O1;
output bool O1_val;
output bool O2;
output bool O2_val;
bool f_term = false;
{
  fork
    l5:
    O1 = false;
    O2 = false;
    if ! f_term then
      pause;
      goto l5
    end;
  par
    fork
      l2:
      if ! A then
        pause;
        goto l2
      end;
      B = B | true;
      B_val = true;
      O1 = O1 | true;
      O1_val = true
    par
      l4:
      pause;
      if ! B then
        goto l4
      end;
      O1 = O1 | true;
      O1_val = true
    join;
    O1 = O1 | true;
    O1_val = false;
    O2 = O2 | true;
    O2_val = true;
    f_term = true
  join
}
```

(b) ABO in SCL

Listing 6.3. The Transformation of ABO from Esterel to SCL

Chapter 7

Conclusion

This chapter summarizes how Esterel is transformed to SCL and how the implementation was evaluated. Additionally, some approaches for future work are presented.

7.1 Summary

It has been shown that Esterel can be transformed to SCL. Therefore, less statements than the kernel statements of Esterel suffice for expressing Esterel programs.

Since the nesting of Esterel statements may produce redundancy in the resulting SCL code, optimizations for both, the transformation and the resulting SCL code are beneficial. These can reduce the generated SCL code by approximately 14%.

The compilation chain from SCL/SCG to target code allows the compilation of Esterel via the transformation to SCL, which can be seen as an intermediate format. Even though the generated code is long and slow in comparison to the PDG approach used by the CEC, the code generation runs faster. Other compilation techniques that may be developed for SCL may resolve this disadvantage.

In most cases, the transformation rules allow to apply the SC MoC to Esterel since the transformation rules preserve sequentially constructive structures. This enables the compilation of a larger set of Esterel programs in comparison to classical Esterel compilers.

7.2 Future Work

During the work on the transformation from Esterel to SCL, some ideas for future work emerged. These are discussed in the following and some initial thoughts are announced.

7.2.1 Weak Suspend

Being introduced with Esterel v7, the `weak suspend` statement brings additional expressiveness to Esterel compared to Esterel v5. This means that the `weak suspend` statement cannot be translated to Esterel v5 kernel statements, as introduced in Section 1.1.1 [PBEB07].

When a `weak suspend` is activated by the corresponding event, the control-flow remains where it was at the beginning of that tick. Nonetheless this ticks actions are performed. Another formulation is that the current tick is executed, but the next one starts at the same state.

```
1  input s;
2  output d,e;
3
4  weak suspend
5    present s then
6      pause;
7      emit d
8    else
9      pause;
10     emit e
11   end;
12 when s;
13 pause
```

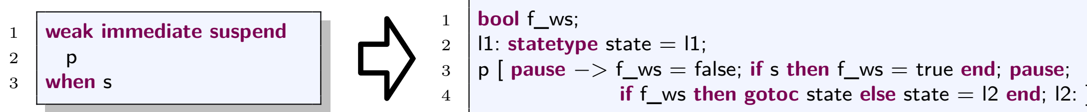
Listing 7.1. Tick boundaries may occur depending on input variables.

One characteristic of the weak suspend statement is that, in contrast to Esterel v5 statements, the starting point of a tick has to be stored. A minimalistic example can be seen in Listing 7.1: As s is an input signal, it cannot statically be decided where a tick ending in line 12 started. This information has to be gained dynamically during the execution.

For returning to a point in the program that is determined during run-time, a computed goto can be used. That is, the target of the goto is given as a variable. SCL does not have a computed goto, but it is possible to emulate this by normal variables and if-then-else statements. In the following, gotoc will be used as an abbreviation. With this in mind, the rule given for the immediate variant of weak suspend in Listing 7.2 could be a solution. A variable state keeps track of where the tick started and is initialized by a label at the start of the statement. Before each pause, a flag f_ws is initialized by false and set to true if the triggering event occurs. In the next tick, the flag is checked and if it evaluates to true, the control continues at the label specified by state. Otherwise, the state is set to a new label that is directly inserted to allow a goto to this point in the next tick in case of weak suspension.

However, considering threads, this solution no longer works, which is illustrated in Listing 7.3. The second tick starts in lines 3 and 5 and ends in lines 8 and 10 in parallel, accordingly only one state variable is not sufficient. As a tick within a weak suspend may start and end in several different threads, this transformation requires some more effort.

Since it is not admissible to jump from one thread to another [vHMA+13], besides



Listing 7.2. A transformation rule for weak suspend that does not work for programs containing concurrency.

```
 1  weak suspend
 2    [
 3      pause;
 4    ||
 5      pause;
 6    ];
 7    [
 8      pause;
 9    ||
10      pause;
11    ];
12  when s
```

Listing 7.3. Esterel programs containing concurrent statements can not be translated by the rule given in Listing 7.2 since ticks may also start in parallel running threads.

more than one state variable, also additional code insertions are necessary. In contrast to other preemptive rules, it is possible that the control has to return to threads that were already left. Hence, also forks, i.e., thread entries, have to be considered for weak suspension.

Listing 7.4 illustrates how parallel threads complicate the transformation. If the weak suspend is triggered in the second tick, b and d are emitted in parallel in lines 5 and 9. Afterwards, the threads join and two new threads are entered. The tick ends as both parallel threads reach a tick boundary in lines 12 and 15. As weak suspension was triggered, the execution starts again in lines 5 and 9 in the next tick.

In SCL, inter thread jumps are not admissible. Therefore, after the pauses in lines 12 and 15 a jump to the thread end would be necessary. After the join in line 17, a jump to the first fork in line 2 has to be executed. Since neither a nor c were emitted in the preceding tick, after the fork, in each thread additional gotos have to target lines 5 and 9.

```
 1  weak suspend
 2    [
 3      emit a;
 4      pause;
 5      emit b
 6    ||
 7      emit c;
 8      pause;
 9      emit d
10    ];
11    [
12      pause;
13      emit a
14    ||
15      pause;
16      emit b
17    ]
18  when s
```

Listing 7.4. For weak suspend, also at entry points of threads additional gotos are necessary.

7.2.2 Transformation of Tasks

```
1 task P(a1,...,an)(b1,...,bn);
2 return R;
3
4 exec P(x1,...,xn)(y1,...,yn) return R
```

Listing 7.5. Definition and Execution of a Task in Esterel

Esterel does not only allow to import functions and procedures from the host language, also *tasks* may be used. Listing 7.5 shows, how tasks are imported and executed. Similar to procedures, the arguments given in the first parenthesis are call-by-reference and, in the second parenthesis, call-by-value arguments are passed. In contrast to procedures, tasks have a return value. Therefore, in the interface a specific return signal is declared.

Tasks are not instantaneous, the statement terminates when the task terminates and therefore they are asynchronous. They run in parallel to the Esterel threads and indicate their termination by the return signal [BdS91].

This is useful, *inter alia*, when commands need time to be executed. For example, if a robot should drive 2 meters forward, it should still be able to react on events that occur while driving or abort the task.

Since tasks are not provided by SCL and the subsequent code generation, they cannot be transformed to SCL. Even though also other compilers are not capable of tasks, it is conceivable to extend the code generation by a comparable feature. This would also allow the use of tasks within SCCharts.

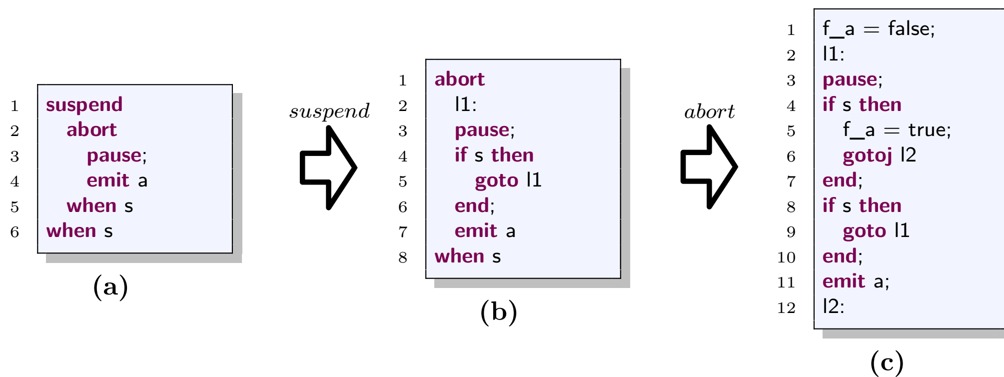
7.2.3 Step-Wise Transformation

Similar to the Single-Pass Language-Driven Incremental Compilation (SLIC) approach used for SCCharts [MSvH14], a modular transformation of Esterel to SCL is imaginable. Such a transformation would allow to compile the program step-by-step and view the intermediate transformation results. However, as briefly described in the following, this is not trivial.

Different Meta-Models

As described in Section 2.2, when compiling SCCharts, the transformation to normalized SCCharts happens within the same meta-model; they are so called *endogenous in-place* transformations. When transforming Esterel to SCL, the target meta-model is different to the source meta-model, which is called an *exogenous* transformation [MG06]. Therefore, it is much more difficult to view intermediate results.

A possible solution could be creating a meta-model for SCL that is enriched by the language features of Esterel. Since Esterel statements can be transformed to SCL, they can be seen as syntactic sugar.



Listing 7.6. Applied in the wrong order, the Esterel to SCL transformation may produce wrong code.

The Relevance of Order

The order, in which the statements are transformed, matters. Listing 7.6 illustrates how the transformation rules generate wrong code when invoked in the wrong order. In this example, if s is present in the second tick, the program should suspend, as the outermost preemptive statement has precedence. The abort statement has no effect, as it also depends on s. Whenever the program terminates, a has to be present since it is emitted.

When the suspend is transformed first, the pause statement is modified with respect to the rule given in Section 4.2.8, resulting in the program seen in Figure 7.6b. If afterwards the suspend transformation, discussed in Section 4.2.9, is applied, the resulting intermediate program in Figure 7.6c is generated. However, the program does not behave as the source program. As abort was translated secondly, the generated conditional statement is located directly after the pause. If now s is present in the second tick, the program will terminate without emitting a. Therefore, as depicted in the SLIC approach, the order of the single transformation steps has to be predefined.

7.2.4 SCL to Esterel Transformation

The transformation of SCL to Esterel can be divided into two tasks:

1. The transformation of SCL programs that are B-constructive.
2. The transformation of SCL programs that are not B-constructive but valid in the SC MoC.

The second task is of major effort. It would require to express the sequentially constructive behavior in valid Esterel, which might be impossible.

Initial thoughts on the first task are presented in the following. The approach makes excessive use of pre-processing the source SCL program and uses only a few Esterel specific

language features to keep things simple. Additionally, much code is duplicated, which might be avoided by a more elaborate transformation. However, the approach seems to work and thus the transformation is possible.

Transformation Rules

Pauses

Since pause has the same semantics in Esterel and SCL, it can directly be translated.

Variables

As SCL considers signals to be syntactic sugar, variables have to be transformed to valued Esterel signals. It should be noted that Esterels variable declarations are local and therefore not suitable for input/output variables. Variables with no influence on the interface may be transformed to local Esterel variables.

Assignments

If variables become valued signals, assignments become valued emits for input/output variables and Esterel assignments for non-input/non-output variables.

Conditionals

SCL conditionals can be translated straightforwardly to Esterel conditionals.

Parallels

The SCL fork-par-join statement can be translated to an Esterel block statement containing the parallel threads. The block statement ensures the correct scope of the parallel statement.

Gotos and Labels

The goto mechanism is the most elaborate SCL aspect to translate to Esterel. To ease the transformation, the SCL program can be pre-processed by eliminating labels within conditionals. This can be done by duplicating code. Further, gotos can be classified into forward- and backward-gotos.

Forward-gotos can be eliminated by producing redundancy. The code following the target label is copied and inserted instead of the goto. Labels and goto targets referencing labels that were copied are modified in order to make them unique. A goto to the end of the thread is added. This step is repeated until a fix-point is reached and the only forward-gotos are those jumping to the end of the thread. The complete thread is then surrounded by a trap statement in Esterel and these last forward-gotos will exit this trap.

```
1  l1:
2  a = !a;
3  l2:
4  b = !b;
5  pause;
6  if I then
7    goto l1
8  end;
9  goto l2
```

Listing 7.7. Overlapping Conditional Loops

It should be noted that a huge amount of duplicated code is produced and there may be more efficient solutions to this problem.

Backward-gotos can be translated by using loops. However, goto-loops may be conditionally, that is, the backward jump is only done if a specific condition holds and sequential control flow continues otherwise. Esterel does not provide this kind of *conditional loop*, thus, a combination of the loop statement and preemptive ones have to be found. Additionally, with backward-gotos unproper nested loops are possible. For example, in the code-snippet in Listing 7.7, there are two conditional loops which overlap. The loop statements of Esterel do not allow this due to structural reasons. Listing 7.8 shows a possible solution by using conditionals to bring backward jumps in order. This procedure gets even more difficult when code is between the gotos.

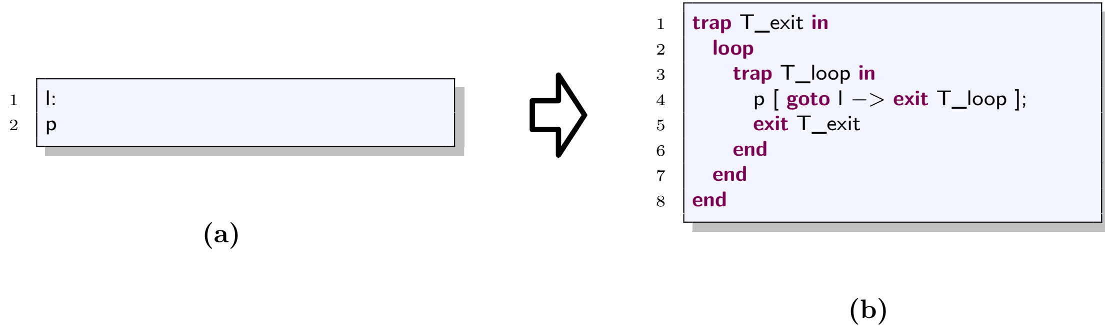
A solution to these problems may be a combination of trap- and loop-statements, as illustrated in Listing 7.9. In the SCL program in Listing 7.9a, the possibly compound statement p is labeled with l and may contain several gotos to that label. In the transformed Esterel program in Listing 7.9b, the outermost trap handles conditional loops, they are exited if the control flow reaches the end of the loop body without executing a backward-goto. Since labels are unique, but several gotos may have the same target label, the innermost trap allows returning to the start of the loop from inside its body. To achieve a proper nesting, the trap-loop-trap starts at the targeted label and ends at the end of the containing thread or program, respectively.

```
1  l1:
2  a = !a;
3  l2:
4  b = !b;
5  pause;
6  if I then
7    goto l1
8  end;
9  goto l2
```

⇨

```
1  l1:
2  a = !a;
3  l2:
4  b = !b;
5  pause;
6  if !I then
7    goto l2
8  end;
9  if I then
10   goto l1
11 end
```

Listing 7.8. Removing Unproper Nested Loops



Listing 7.9. Tranformation of a Backward Jump

7.3 Transformation of Further Synchronous Languages to SCL

Since SCL has proven to be usable as an intermediate format for the compilation of Esterel, a transformation to SCL may also be used to compile other synchronous languages. An example would be *Quartz*, which is a variant of Esterel developed by K. Schneider at the University of Kaiserslautern, Germany [Sch09].

As for Esterel, it can be analyzed how the SC MoC can be adopted for other synchronous languages and the usage of SCL as an intermediate format can further be evaluated.

Bibliography

[And03] Charles André. Semantics of SyncCharts. Technical Report ISRN I3S/RR–2003–24–FR, I3S Laboratory, Sophia-Antipolis, France, April 2003.

[ASU86] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers - Principles, Techniques, and Tools.* Addison-Wesley, Reading, Massachusetts, 1986.

[BC84] 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, 1984.

[BCE+03] Albert Benveniste, Paul Caspi, Stephen A. Edwards, Nicolas Halbwachs, Paul Le Guernic, and Robert de Simone. The Synchronous Languages Twelve Years Later. In *Proc. IEEE, Special Issue on Embedded Systems*, volume 91, pages 64–83, Piscataway, NJ, USA, January 2003. IEEE.

[BdS91] Frédéric Boussinot and Robert de Simone. The ESTEREL language. Another look at real time programming. *Proceedings of the IEEE*, 79(9):1293–1304, September 1991.

[Ber88] Gérard Berry. Programming a digital watch in Esterel v3. Technical Report RR-1032, INRIA, 1988.

[Ber91] Gérard Berry. A hardware implementation of pure ESTEREL. Technical Report de recherche 1479, INRIA, 1991.

[Ber93] Gérard Berry. The semantics of pure Esterel. In *Proceedings of the Marktoberdorf Intl. Summer School on Program Design Calculi*, LNCS. Springer, 1993. `http://citeseer.ist.psu.edu/berry93semantics.html`.

[Ber00] Gérard Berry. *The Esterel v5 Language Primer, Version v5_91.* Centre de Mathématiques Appliquées, Ecole des Mines and INRIA, 06565 Sophia-Antipolis, 2000.

[Ber02] Gérard Berry. *The Constructive Semantics of Pure Esterel.* Draft Book, Version 3.0, Centre de Mathématiques Appliquées, Ecole des Mines de Paris and INRIA, 06902 Sophia-Antipolis, France, December 2002.

[Ber05] Gérard Berry. Esterel v7: From verified formal specification to efficient industrial designs. In M. Cerioli, editor, *Fundamental Approaches to Software Engineering FASE 2005*, volume 3442 of *Lecture Notes in Computer Science*, page 1. Springer, 2005.

[BG92] Gérard Berry and Georges Gonthier. The Esterel synchronous programming language: Design, semantics, implementation. *Science of Computer Programming*, 19(2):87–152, 1992.

[BLL+05] Christopher Brooks, Edward A. Lee, Xiaojun Liu, Stephen Neuendorffer, Yang Zhao, and Haiyang Zheng. Heterogeneous concurrent modeling and design in Java, volume 1: Introduction to Ptolemy II. Technical report, EECS Department, University of California, Berkeley, July 2005.

[Bol03] Azad Bolour. Notes on the Eclipse plug-in architecture. *Eclipse Corner Articles*, July 2003. `www.eclipse.org/articles`.

[Bt00] Gérard Berry and the Esterel Team. *The Esterel v5_91 System Manual*. INRIA, June 2000.

[CFR+91] 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.

[dRB06] Jim des Rivieres and Wayne Beaton. Eclipse platform technical overview. *Eclipse Corner Articles*, April 2006. `www.eclipse.org/articles`.

[Edw02] 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.

[EEK+12] Sven Efftinge, Moritz Eysholdt, Jan Köhnlein, Sebastian Zarnekow, Robert von Massow, Wilhelm Hasselbring, and Michael Hanus. Xbase: Implementing domain-specific languages for java. *SIGPLAN Not.*, 48(3):112–121, September 2012. URL: `http://doi.acm.org/10.1145/2480361.2371419`, `doi:10.1145/2480361.2371419`.

[Est] Estbench Esterel benchmark suite. `http://www1.cs.columbia.edu/~sedwards/software/estbench-1.0.tar.gz`.

[Est05] Esterel Technologies. *The Esterel v7 Reference Manual Version v7_30—initial IEEE standardization proposal*. Esterel Technologies, 06270 Villeneuve-Loubet, France, November 2005.

[EZ07] Stephen A. Edwards and Jia Zeng. Code generation in the Columbia Esterel Compiler. *EURASIP Journal on Embedded Systems*, Article ID 52651, 31 pages, 2007.

[FOW87] Jeanne Ferrante, Karl J. Ottenstein, and Joe D. Warren. The program dependence graph and its use in optimization. *ACM Transactions on Programming Languages and Systems*, 9(3):319–349, 1987.

[Küh06] Lars Kühl. Transformation von Esterel nach SyncCharts. Diploma thesis, Christian-Albrechts-Universität zu Kiel, Department of Computer Science, February 2006. http://rtsys.informatik.uni-kiel.de/~biblio/downloads/theses/lku-dt.pdf.

[Lee06] Edward A. Lee. The problem with threads. *Computer*, 39(5):33–42, May 2006.

[MFvH09] Christian Motika, Hauke Fuhrmann, and Reinhard von Hanxleden. Semantics and execution of domain specific models. Technical Report 0923, Christian-Albrechts-Universität zu Kiel, Department of Computer Science, December 2009.

[MG06] Tom Mens and Pieter Van Gorp. A taxonomy of model transformation. *Electronic Notes in Theoretical Computer Science*, 152:125–142, 2006. Proceedings of the International Workshop on Graph and Model Transformation (GraMoT 2005). URL: http://www.sciencedirect.com/science/article/B75H1-4JFR8K3-B/2/cd561440d16d44082d7b63da61c70252, doi:DOI:10.1016/j.entcs.2005.10.021.

[Mot09] Christian Motika. Semantics and execution of domain specific models—KlePto and an execution framework. Diploma thesis, Christian-Albrechts-Universität zu Kiel, Department of Computer Science, December 2009. http://rtsys.informatik.uni-kiel.de/~biblio/downloads/theses/cmot-dt.pdf.

[MP95] Zohar Manna and Amir Pnueli. *Temporal Verification of Reactive Systems: Safety*. Springer, New York, NY, USA, 1995.

[MSvH14] Christian Motika, Steven Smyth, and Reinhard von Hanxleden. Compiling sccharts—a case-study on interactive model-based compilation. In *Proceedings of the 6th International Symposium on Leveraging Applications of Formal Methods, Verification and Validation (ISoLA 2014)*, volume 8802 of *LNCS*, pages 443–462, Corfu, Greece, October 2014.

[PB02] Dumitru Potop-Butucaru. *Optimizations for faster simulation of Esterel programs*. PhD thesis, Ecole des Mines de Paris, France, November 2002.

[PBEB07] Dumitru Potop-Butucaru, Stephen A. Edwards, and Gérard Berry. *Compiling Esterel*. Springer, May 2007.

[Pow99] James Power. Notes on formal language theory and parsing. Maynooth, Co. Kildare, Ireland, 1999.

[PTvH06] Steffen Prochnow, Claus Traulsen, and Reinhard von Hanxleden. Synthesizing Safe State Machines from Esterel. In *Proceedings of ACM SIGPLAN/SIGBED Conference on Languages, Compilers, and Tools for Embedded Systems (LCTES'06)*, Ottawa, Canada, June 2006.

[Rüe11] Ulf Rüegg. Interactive transformations for visual models. Bachelor thesis, Christian-Albrechts-Universität zu Kiel, Department of Computer Science, March 2011.

[Sch09] Klaus Schneider. The synchronous programming language Quartz. Internal Report 375, Department of Computer Science, University of Kaiserslautern, Kaiserslautern, Germany, December 2009.

[SF93] Barbara Simons and Jeanne Ferrante. An efficient algorithm for constructing a control flow graph for parallel code. Technical Report 03465, IBM, Santa Teresa Laboratory, San Jose, Calif, USA, February 1993.

[Smy13] Steven Smyth. Code generation for sequential constructiveness. Diploma thesis, Christian-Albrechts-Universität zu Kiel, Department of Computer Science, July 2013. `http://rtsys.informatik.uni-kiel.de/~biblio/downloads/theses/ssm-dt.pdf`.

[Tar04] Olivier Tardieu. *Loops in Esterel: From Operational Semantics to Formally Specified Compilers*. PhD thesis, INRIA Sophia-Antipolis Research Unit, September 2004.

[Tar05] Olivier Tardieu. A deterministic logical semantics for Esterel. *Electronic Notes in Theoretical Computer Science*, 128(1):103–122, 2005.

[TdS03] Olivier Tardieu and Robert de Simone. Instantaneous termination in pure Esterel. In *Static Analysis Symposium*, San Diego, California, June 2003.

[vH09] Reinhard von Hanxleden. Synccharts in c—a proposal for light-weight, deterministic concurrency. In Albert Benveniste, Stephen A. Edwards, Edward Lee, Klaus Schneider, and Reinhard von Hanxleden, editors, *SYNCHRON'09—Proceedings of Dagstuhl Seminar 09481*, number 09481 in Dagstuhl Seminar Proceedings. Internationales Begegnungs- und Forschungszentrum (IBFI), Schloss Dagstuhl, Germany, 22–27 November 2009.

[vHDM+14] Reinhard von Hanxleden, Björn Duderstadt, Christian Motika, Steven Smyth, Michael Mendler, Joaquín Aguado, Stephen Mercer, and Owen O'Brien. SCCharts: Sequentially Constructive Statecharts for safety-critical applications. In *Proc. ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'14)*, Edinburgh, UK, June 2014. ACM.

[vHMA+13] Reinhard von Hanxleden, Michael Mendler, Joaquín Aguado, Björn Duderstadt, Insa Fuhrmann, Christian Motika, Stephen Mercer, Owen O'Brien, and Partha Roop. Sequentially Constructive Concurrency—A conservative extension of the synchronous model of computation. Technical Report 1308, Christian-Albrechts-Universität zu Kiel, Department of Computer Science, August 2013. ISSN 2192-6247.