Strict Sequential Constructiveness

Alexander Schulz-Rosengarten

Master's Thesis
2016

Real-Time and Embedded Systems Group
Prof. Dr. Reinhard von Hanxleden
Department of Computer Science
Kiel University

Advised by
Dipl.-Inf. Steven Smyth

Eidesstattliche Erklärung

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

Kiel,

Abstract

Synchronous languages offer deterministic concurrency but often come with heavy restrictions on the accepted programs. Classical synchronous languages such as *Esterel* use the concept of *signals* which require a globally consistent value.
The Sequentially Constructive (SC) Model of Computation (MoC) overcomes this restriction and enables the use of sequential programming paradigms with multivalued shared variables while preserving determinism. It is designed as a conservative extension of the synchronous MoC, including Esterel. However, in terms of speculation, the concept of constructiveness in Esterel differs from constructiveness in the SC MoC. While constructive Esterel is based on the idea of propagating facts about signals and controlflow, the SC MoC considers all possible program traces under a restricted scheduling regime. This results in situations where the SC MoC accepts programs whose execution traces are considered speculative in the sense of Esterel's constructiveness. This raises the problem that these programs do not translate into delay-insensitive circuits, which is a strong property of constructive Esterel.
The topic of this thesis is about restricting the class of SC programs to those Strictly Sequentially Constructive (SSC) programs which are considered constructive in the strictly non-speculative sense of Esterel, while preserving the benefits of sequentiality in the SC MoC.
This thesis presents a practical approach to determine whether an SC program fulfills this requirement by translating it into a semantically equivalent Esterel program and checking its constructiveness. This requires an SC-specific Static Single Assignment (SSA) form to allow the correct transformation of SC concurrency and shared variables into Esterel.

Contents

Chapter 1

Introduction

Many computer systems are *embedded* into their environment. These embedded systems range from digital watches over cars to aircrafts and most of them are *reactive* [HP85]. Such systems maintain a continuous exchange of information with their environment. As a result, these systems often use concurrency to interact with the simultaneously running real-world processes. In a *safty-critical* context where malfunctions can easily endanger human lives, this becomes a crucial aspect. Classical approaches for concurrency, such as *threads*, are prune to non-deterministic behavior, especially *race conditions* [Lee06]. In safty-critical systems, such behavior is dangerous threat and thus unacceptable. Synchronous languages represent a solution for this problem. The *synchronous* Model of Computation (MoC) divides time into discrete *ticks*, executing one finite reaction of the system in an instant. This facilitates the definition of deterministic concurrency by reasoning about ticks [BCE+03].

1.1 Esterel

One prominent synchronous language is *Esterel*, developed by Gérard Berry [Ber02]. Esterel is an imperative synchronous programming language especially designed for reactive systems, providing built-in parallelism with deterministic semantics. The main data type in Esterel are *signals*. Signals can either be *present* or *absent* and their state is globally consistent during an entire tick. A signal is present in a tick iff it is emitted in this tick, otherwise it is absent.
Furthermore, the constructive semantics of Esterel, implements *[...] the idea of propagating facts about control flow and signal statuses* [Ber02]. Constructive programs must determine the state of a signal based on propagated facts before the state is further evaluated. If a statement reads a signal, the state is determined based on preceding emissions and cannot not change due to further execution of the program. This results in a *write before read* protocol for signal accesses, and prevents *speculation* about the presence of a signal.
Listing 1.1 shows the program `P12`, which violates the concept of constructiveness and consequently is rejected. The program has one output signal `O` which is first tested for its presence state and then emitted in each of the branches. The reason for the rejection is the fact that, when the output signal is read, the signal was not yet emitted and can be emitted in the same tick by the subsequent statements. Thus, the controlflow does not allow to propagate enough facts about the signal to determine its state. In this case `O` will always be emitted, independent from branching. However, taking this fact into account when determining the signal state would be speculation about subsequent effects.

```
module P12:
output O;
present O then
  emit O
else
  emit O
end
end module
```

Listing 1.1. The Esterel program `P12` which is not constructive in the sense of Esterel [Ber02]

```
module P12
output bool O;
{
  O = false;
  present O then
    O = true
  else
    O = true
  end
}
```

Listing 1.2. The `P12` program in SCL

1.2 Sequential Constructiveness

The Sequentially Constructive (SC) MoC developed by von Hanxlenden et al. [HMA+14] conservatively extends the synchronous MoC. It allows multiple sequential read and write accesses to variables during the same tick, resulting in multiple values which can be read from the same variable. Nevertheless, the SC MoC provides deterministic concurrency.

Regarding the conservative extension, programs that are considered constructive in the synchronous MoC are also considered SC. Thus, introducing a sequential concept for variable values allows to accept the program `P12` from Listing 1.1. An SCL version of the program is presented in Listing 1.2. Naturally, the semantics of the program changes under the SC MoC. Since signals are implicitly reset to absent in every tick, the variable `O` is initialized to false, representing this behavior. When reading `O`, the current value is false and the else branch is taken. After writing `O` to true all sequentially following statements will read this value from `O`. Hence, the SC MoC does not require a globally consistent value for variables.

As mentioned before, unrestricted concurrent execution of program code can be nondeterministic. To assure the determinism of accepted programs, the SC MoC introduces the initialize-update-read (iur) protocol for scheduling concurrent variable access. The SC MoC considers a *free scheduling* and states a schedule as *SC-admissible* run if all statements comply with their sequential ordering and the iur protocol. If at least one SC-admissible run exists and all SC-admissible runs generate the same deterministic trace of finite macro ticks, the program is accepted. Listing 1.3 shows the program `P10` written in SCL, a minimal imperative language created in the process of defining the semantics of SC.

The program first initializes `y` and then forks into two threads. One initializing the variable `x` to 1 and initializing `y` with the value of `x` and the other initializing the variable `x` to 0 if `y` is 0. Consequently, statement `S2` has to read `x` before writing to `y` and `S4` may write `x` if the read value of `y` in `S3` is 0. Figure 1.1 illustrates the SCG representation of the `P10` program including iur dependencies between the nodes. The green arrows are read-before-write dependencies and the dependency in red points out a possible write-write conflict.

```
1  module P10
2  int x, y;
3  {
4    y = 0;            //S1
5    fork
6      x = 1;          //S2
7      y = x           //S3
8    par
9      if y == 0 then //S4
10       x = 0         //S5
11     end
12   join
13 }
```

Listing 1.3. The P10 program in SCL[1]

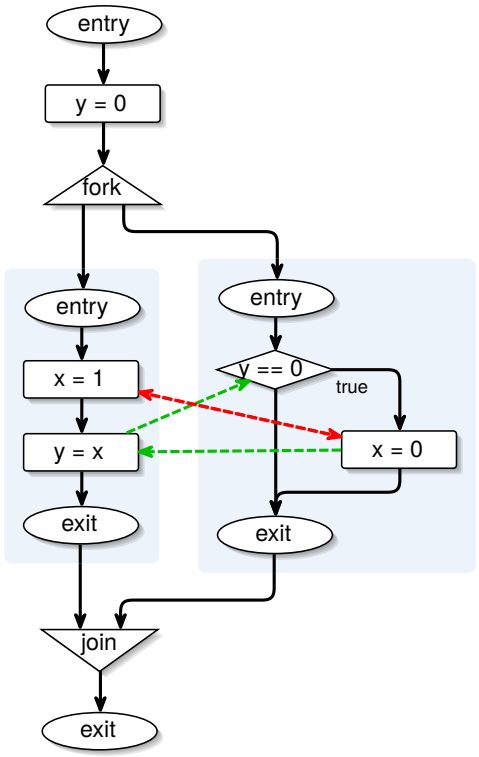


Figure 1.1. SCG representation of program P10 with dependencies

The program P10 is considered SC because only one SC-admissible run exists, generated by the following schedule of statements:

$$S1 \rightarrow S2 \rightarrow S3 \rightarrow S4$$

Any other trace is considered invalid because it violates the initialize-before-read rule of the IUR protocol.

1.3 Problem Statement

Comparing the two concepts of sequential constructiveness and the constructive semantics of Esterel, the SC MoC does not follow the same idea of no speculation and propagating facts when accepting programs. All execution traces of a program may result in SC-admissible runs, regardless of whether they can be constructed by propagating facts about control flow and variable values or not. The SC-admissible run of P10 illustrates this concept. Statement S3 is scheduled before S4 because it reads y and S5 is explicitly ordered sequentially after S4. If in

[1] https://www.rtsys.informatik.uni-kiel.de/en/synchron-2015/Reinhard-von-Hanxleden.pdf

S4 y is read without executing statement S5 beforehand which is not possible, then S5 will not be executed because y is equals zero. The SC-admissibility of the schedule is based on the fact that S5 is not executed in this schedule which justifies the ordering. However, this fact cannot be constructively determined when scheduling S3 because statement S5 could influence the value of x. Furthermore, programs which do not comply with the constructive semantics of Esterel are not guaranteed to form delay-insensitive circuits.
Thus, establishing the non-speculative concept of Berry's constructive semantics requires a restriction of Sequential Constructiveness. This thesis presents a practical approach to detect the Strictly Sequentially Constructive (SSC) programs which are considered constructive in the sense of Esterel. The approach is based on a translation of SC programs into semantically equivalent Esterel programs using an SC-specific Static Single Assignment (SSA) form to transform scheduling constraints and variable accesses. Checking the constructiveness of the resulting program allows to conclude the constructiveness of the source program.

1.4 Outline

Subsequently to this chapter, Chapter 2 presents the theoretical foundations of the constructive semantics of Esterel and the SC MoC in more detail. Furthermore, Section 2.3 introduces the technologies used in the implementation, especially the KIELER project which provides the implementation for the SC MoC. Chapter 3 presents the related work. First, Section 3.1 describes related approaches to translate synchronous languages into Esterel or the other way round. Secondly, Section 3.2 presents related work which use SSA in the context of synchronous or explicitly parallel programming languages. The concept of Strict Sequential Constructiveness is presented in Chapter 4. At first, Section 4.1 illustrates the concept of restricting Sequential Constructiveness and the approach of translating SC programs into Esterel to check their constructiveness. Afterwards, Section 4.2 presents the SSA form which transforms the more complex sequential and concurrent aspects of the SC MoC such that the semantics and language definition of Esterel can handle it. The final translation from SC programs in SSA form into Esterel is described in Section 4.3. Chapter 5 presents the implementation of the previously described concept and its integration into the KIELER project. The concept of Strict Sequential Constructiveness is evaluated in Chapter 6 by investigating Esterel constructiveness of characteristic SC programs. In the end, Chapter 7 summarizes the ideas and results of this thesis and presents an outlook on potential future work.

Chapter 2

Foundations

The *synchrony hypothesis* emerged form the challenge of designing correct and efficient programs for embedded reactive systems [PST05]. In principle, embedded reactive systems are constantly interacting with their environment. They read inputs, compute their reaction and convey outputs. The *synchronous hypothesis* presents a discretization of the physical time based on this behavior. The computation is separated into single execution instants, called ticks. Figure 2.1 illustrates this discretized lifecycle of an embedded reactive system. Furthermore, this concept of instants requires the conceptual abstraction that such a reaction is instantaneous, thus takes zero-time to compute. Consequently, outputs are generated at the same time the inputs are read. Such a system is considered in *perfect synchrony*.
When signals are used to propagate information, the concept of perfect synchrony requires that the signal state must be consistent for all read operations during an instant, especially for concurrent components. Hence, it is a crucial task in synchronous program validation to decide whether a signal is present or absent.
Nevertheless, the concept of zero-time computation is only a theoretical abstraction for the MoC. In a tick, the system performs a *macro step* of the designed logic. The logic itself performs a finite sequence of *micro steps* to compute the result of the reaction, for example the sequential execution of statements in an active code segment. Figure 2.2 illustrates the abstraction of micro steps and macro steps. The sequence of transitions may be as complex as the logic requires, but must always be finite to coincide with the concept of zero duration.

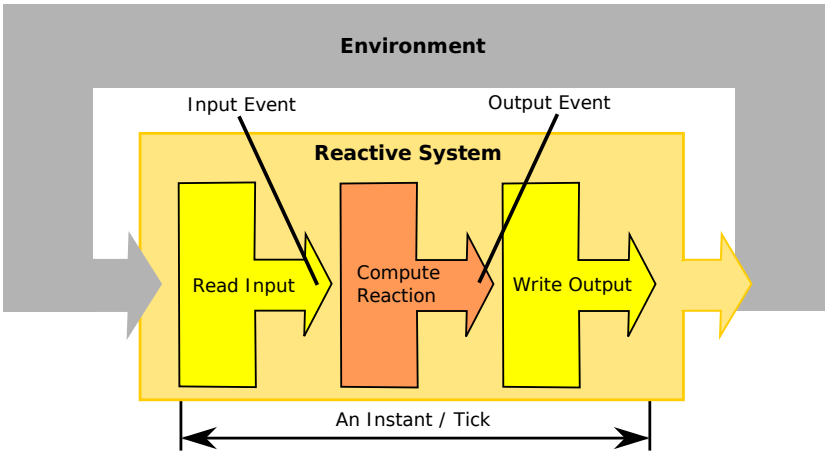


Figure 2.1. Embedded Reactive System, based on [MHH13]

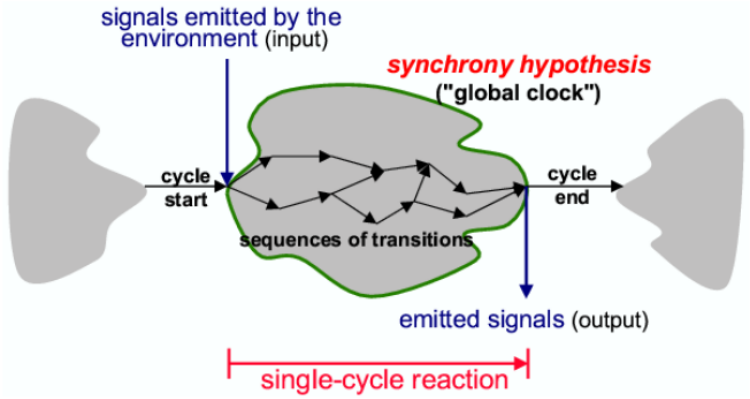


Figure 2.2. Synchrony Hypothesis (G. Luettgen, 2001)

From a practical point of view, it is sufficient that the computation is *fast enough* to prevent overlapping of execution instants. The ticks are driven by a global clock according to timing requirements of the physical environment. Consequently, this requires the use of Worst-Case Execution Time (WCET) analyses to assure the correct timing behavior.
The synchronous MoC is especially suited for safety-critical systems since the discretization of time allows reasoning about ticks to assure the correct and deterministic behavior of a system.

2.1 Esterel

Esterel implements the synchronous MoC in an imperative programming language [BC84; BS91; BG92; Ber02]. It is tailored for designing controllers for reactive systems driven by input events. The controller programming benefits from the synchronous hypothesis by reconciling concurrency and determinism. Furthermore, Esterel facilitates hardware software co-design [MG97]. Esterel allows to compile programs into executables or circuits, both providing the exact same behavior.
The Esterel language provides a wide set of programming constructs [Ber00], but most can be seen as syntactic sugar since they can be derived from other statements. Hence, Esterel can be reduced to a *kernel language* which provides the full semantical capability while requiring a minimal set of statements for a semantical definition or a compiler implementation. Table 2.1 lists the statements available in the kernel language.
Esterel uses signals as primary communication mechanism. In addition to normal signals denoted as *pure signals*, Esterel also provides *valued signals*. Valued signals have a state just like pure signals, which is reset every tick. Additionally, they carry a value which is persisted across ticks. Valued signals support multiple emissions with different values in the same tick, if a combine function is assigned to handle the deterministic merge of all emitted values. Moreover, Esterel provides variables. Variables can be assigned in the common manner,

Statement	Description
`nothing`	no-op statement
`emit S`	signal broadcasting
`present S then` p `else` q	signal test
`pause`	unit delay
`suspend` p `when S`	suspension
p`;` q	sequential composition
`loop` p `end`	loop
p `||` q	parallel composition
`trap T in` p `end`	exception catch clause
`exit T`	exception trigger
`signal S in` p `end`	local signal declaration

Table 2.1. Esterel kernel language [Ber02]

	Esterel			SCL	C
	Pure Signals	Valued Signals	Variables		
Syntax	`emit` x `present` x	`emit` $x(v)$ `if` $(?x)$	x `:=` y `if` (x)	$x = y$ `if` (x)	$x = y$ `if` (x)
Type	present/absent	arbitrary	arbitrary	arbitrary	arbitrary
Initialized each tick	yes (absent)	no	no	no	no
Persistence across ticks	no	yes	yes	yes	yes
Multiple values per tick	no	no	yes	yes	yes
Sequential scheduling constraints	first emit → reads	emits → reads	none	none	none
Concurrent scheduling constraints	first emit → reads	emits → reads	read only	inits → updates → reads	none
Determinacy guaranteed	yes	yes	yes	yes	no

Table 2.2. Comparison of data handling, based on [RSM+15]

including sequential overriding of values in the same tick. However, to provide deterministic concurrency, variables are considered local to the writing thread. That means, either all concurrent threads have read-only access to the variable, or only one thread can read and write the variable and all other must neither read nor write. Table 2.2 presents the different forms of data handling in Esterel and compares their behavior and properties with other languages. The table lists the reset behavior and constraints of the different methods of data handling. Especially the sequential and concurrent scheduling constrains affect the applicability of the different types. Variables in C are the least restricted form, but do not guarantee determinacy. Esterel variables allow similar data handling, independent from the

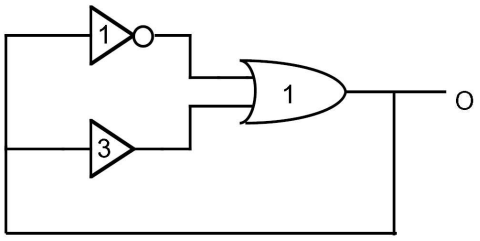


Figure 2.3. P12 Circuit [Ber02]

globally consistent value required for signals. However, to achieve determinacy the concurrent usage allows only read only access. A compromise is presented by SC variable handling, providing multiple values per tick with a deterministic concurrent scheduling regime.
In Esterel, signals are the primary form of data handling. The main challenge in defining a sound semantics for Esterel is *coherence*, the reasoning about presence or absence of a signal in a tick.
A first intuitive attempt to define the semantics is *logical correctness*. A programs is considered logically correct if for all input events, there exists only one global status for all signals that satisfies the *logical coherence law*:

> A signal `S` is present in an instant if and only if an `emit S` statement is executed in this instant. [Ber02]

Performing an exhaustive case analysis can be used to analyze an Esterel program for its logical correctness. The program `P12` from Listing 1.1 on page 2 is considered logical correct, since only $\mathtt{O} \leftarrow \mathit{present}$ satisfies the logical coherence law. However, this semantics is contradictory to the common programming intuition and sequential controlflow because the value of `O` is accessible before it is written. Another important aspect is that the correct behavior of the program cannot be guaranteed under all circumstances. The program `P12` represents the following logic:

$$\mathtt{O} = \mathtt{O} \vee \neg\mathtt{O}$$

Figure 2.3 illustrates the corresponding logic circuit of `P12`. This circuit is not guaranteed to stabilize to the expected output value w.r.t. the wire initialization. For example the component delays indicated by the numbers in the figure cause the wire representing `O` to oscillate.
These two problems inspired a more constructive approach which follows the sequential intuition and produces sound circuits. The constructive semantics of Esterel is defined for pure signals based on the kernel language [Ber02]. There are three equivalent definitions for the constructiveness semantics.

Constructive Behavioral Semantics: The *structural operational semantics* [Plo81] of Esterel provides formal rules for analyzing the behavior of an Esterel program. These rules allow to constructively reason about whether a statement *must* or *cannot* be executed based on a given environment. The constructive behavioral semantics extends the signal states by an *unknown* value and defines a *constructive coherence law*:

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. [Ber02]

The analysis starts with all non-input signals unknown and tries to constructively derive a known value. The program `P12` from Listing 1.1 on page 2 is rejected because `O` must not be emitted since the must analysis is not allowed to speculatively execute branches and `O` can be emitted in both branches.

Constructive Operational Semantics: The constructive operational semantics also uses the concept of tree valued signals. However, instead of reasoning about whether a signal must and cannot be emitted, the programs are simply simulated. The simulation executes the micro steps of the program and requires the signals to monotonically reach an unique known state.

Constructive Circuit Semantics: Esterel defines rules to translate programs into boolean digital circuits. The constructiveness of such circuits [SBT96] is tested with a monotonic ternary analysis [Mal94] for all program states in the stable domain. A complicated problem arising when translating programs into circuits is *schizophrenia*. There are constructive programs that produce non-constructive cyclic circuits because some loops allow to execute the same statement twice in the same tick, which is then considered schizophrenic.

2.2 Sequential Constructiveness

The data handling and constructiveness of Esterel provides deterministic concurrency but come along with restrictions on the accepted programs. Especially the use of signals and the restrictions on shared variables stand contrary to common programming paradigms, for example known from C. Table 2.2 illustrates this fact by comparing the data handling of Esterel with C.
The SC MoC overcomes this restriction and allows sequential and concurrent variable access during tricks [HMA+14]. Additionally, the SC MoC conservatively extends the classical synchronous MoC and preserves deterministic concurrency. This means that variables can be read and written in any order and multiple times as long as the program can be scheduled such that it provides a deterministic behavior. The SC MoC renounces the concept of signals and only uses variables. Variables provide a more general form of data handling which allows in combination with the SC scheduling regime to represent signals by variables. This representation include explicit modeling of the emission and reset behavior. Table 2.2 also illustrates SC variables and their properties. SCL represents a minimal definition for an SC programming language and provides the same structural components as the SCG. The SCG is a controlflow graph (CFG) extension for the SC MoC and provides the basic structure for the semantics definition and compilation. Table 2.3 presents the structural components of SCL and SCG.
The SC MoC provides deterministic concurrent variable access by introducing a restricted scheduling regime. In general, the SC MoC uses a concept of *free scheduling*, considering the

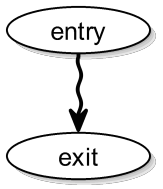
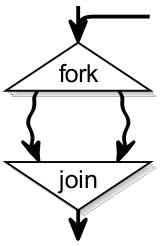
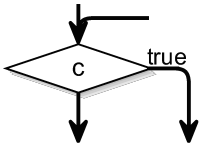
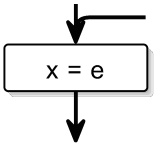
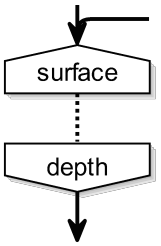
	Thread	Parallel	Sequence	Conditional	Assignment	Delay
SCL	t	`fork` t_1 `par` t_2 `join`	s_1 `;` s_2 `goto` l	`if` (c) s_1 `else` s_2	$x = e$	`pause`
SCG						

Table 2.3. Overview of SCL and SCG elements, based on [RSM+15]

sequential ordering of the statements and allowing arbitrary interleaving of threads. However, for the concurrent access on variables the SC MoC dictates the use of the iur protocol. The iur protocol adds dependencies between concurrent nodes in the SCG, accessing the same variable. These dependencies state that all initializations must be scheduled before updates and all writes before reads. Initializations are absolute writes in the form $x = e$, were e is an expression independent from x. Since the iur protocol only dictates the order types of writes, two concurrent initializations can still be the source of non-determinism and consequently are considered a write-write conflict. Updates are relative writes in the form $x = f(x, e)$, were e is a sideeffect-free expression independent from x and f is a combine function. The definition of a combine function requires that the result of multiple updates with the same combine function must be independent from the order of application. Consequently, concurrent updates with different combine functions, for example addition and multiplication, are also considered a write-write conflict. However, write-write conflicts are only a potential source for non-determinism. Two concurrent nodes are considered *conflicting* if they are active in the same tick and the order of execution influences the result of the program. In the absence of such a conflict the nodes are considered *confluent*. Thus, two initializations with a write-write dependency can be non-conflicting, if they are never executed in the same tick or both expressions are evaluated to the same value.

Listing 2.1 illustrates the iur protocol by presenting an SCL program `IUR` which performs an initialization, two updates and a read in three concurrent threads. Figure 2.4 shows the corresponding SCG including iur dependencies between the concurrent nodes.

A non-conflicting schedule for a single macro tick which complies with the iur protocol and the sequential ordering is considered SC-admissible. If all macro ticks in a run for an SCG are SC-admissible, it is denoted an SC-admissible run. A program is considered SC if at least one SC-admissible run exists and all SC-admissible runs generate the same deterministic trace of finite macro ticks.

To facilitate practical analysis, the class of SC programs can be restricted to Acyclic Sequentially Constructive (ASC) programs. This approach abstracts from the dynamic nature of conflicts

```
1  module IUR
2  int x, y;
3  {
4    fork
5      x = 0
6    par
7      x = x + 1;
8      x = x + 1
9    par
10     y = x
11   join
12 }
```

Listing 2.1. The IUR program in SCL

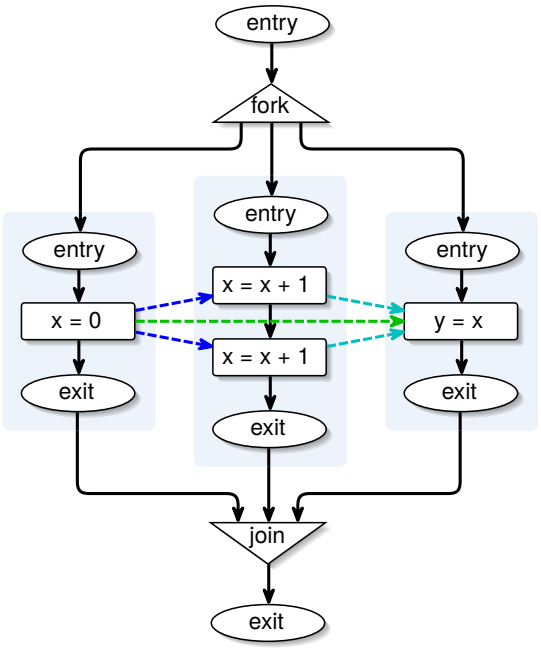


Figure 2.4. SCG representation of the IUR program with dependencies

and statically analyzes the controlflow and iur relations between nodes in the SCG. If the sequential order and the iur dependencies form an acyclic SC-schedule the SCG is considered ASC.

An SCG which is ASC can be statically scheduled, resulting in a Sequentialized SCG using guarded blocks. This dataflow approach uses the idea of creating a netlist for the program, which facilitates hardware synthesis [HDM+14]. Another more dynamic compilation approach is the priority-based scheduling. A priority is assigned to each node in the SCG representing the scheduling order and a runtime environment pasted into the translated program handling the dispatching between threads.

Alongside SCL and SCG, von Hanxleden et al. developed Sequentially Constructive Statecharts (SCCharts), a graphical SC language with statechart notation, similar to SyncCharts [HDM+14]. The graphical syntax is incrementally defined to provide extended features based on a core set of syntax elements, using the same concept as the Esterel kernel statements. All SCCharts can be transformed into core SCCharts and further into SCGs. Figure 2.5 shows the ABO SCChart, which only uses core language features. In the first tick O1 and O2 are initialized with false. Afterwards, the two concurrent regions wait for the input variable to become true. HandleA reacts immediately on the input event, sets O1 to true and communicates to the other region that B is true. The reaction of HandleB is always delayed by one tick, thus the reaction to B can occur at the earliest in the second tick. In the tick were both regions have reached their final state, the macro state terminates and sets both O1 and O2 to true. Listing 2.2 presents the equivalent SCL code for ABO.

```
1  module ABO
2  input output bool A, B;
3  output bool O1, O2;
4  {
5    O1 = false;
6    O2 = false;
7    fork
8      HandleA:
9      if !A then
10       pause;
11       goto HandleA;
12     end;
13     B = true;
14     O1 = true;
15   par
16     HandleB:
17     pause;
18     if !B then
19       goto HandleB;
20     end;
21     O1 = true;
22   join;
23   O1 = false;
24   O2 = true;
25 }
```

Listing 2.2. The ABO program in SCL [HDM+14]

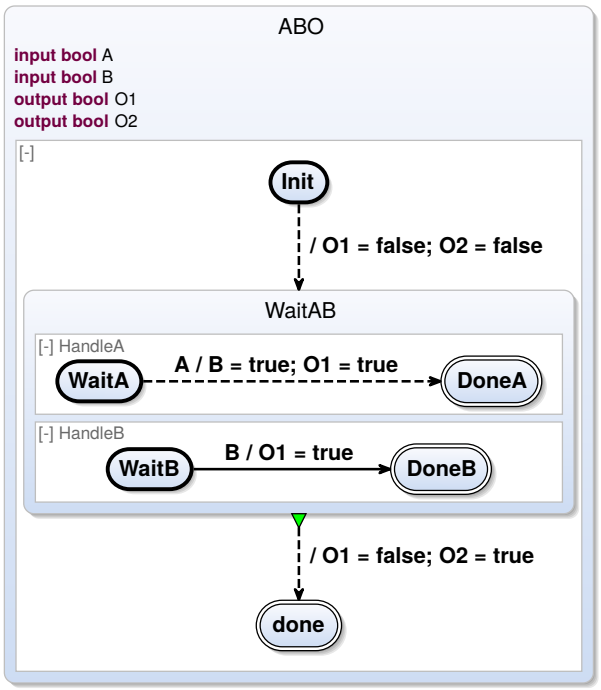


Figure 2.5. The ABO SCChart [HDM+14]

With the SC MoC conservatively extending the synchronous MoC, it allows programs considered constructive in the semantics of Esterel to also be considered SC and to retain the same semantics. Figure 2.6 illustrates the relations of program classes produced by the different synchronous semantics. The class of constructive Esterel programs forms a subset of the SC program class. Furthermore, the class of ASC programs restricts the class of SC program and covers the acyclic constructive Esterel programs. The class of SC program only intersects with the class of logically correct programs, since variables with multiple values per tick cannot have one global logical coherent status. The same holds for the semantics defined by Pnueli and Shalev [PS91], which checks the globally consistent signal state by speculation of the absence of signals.

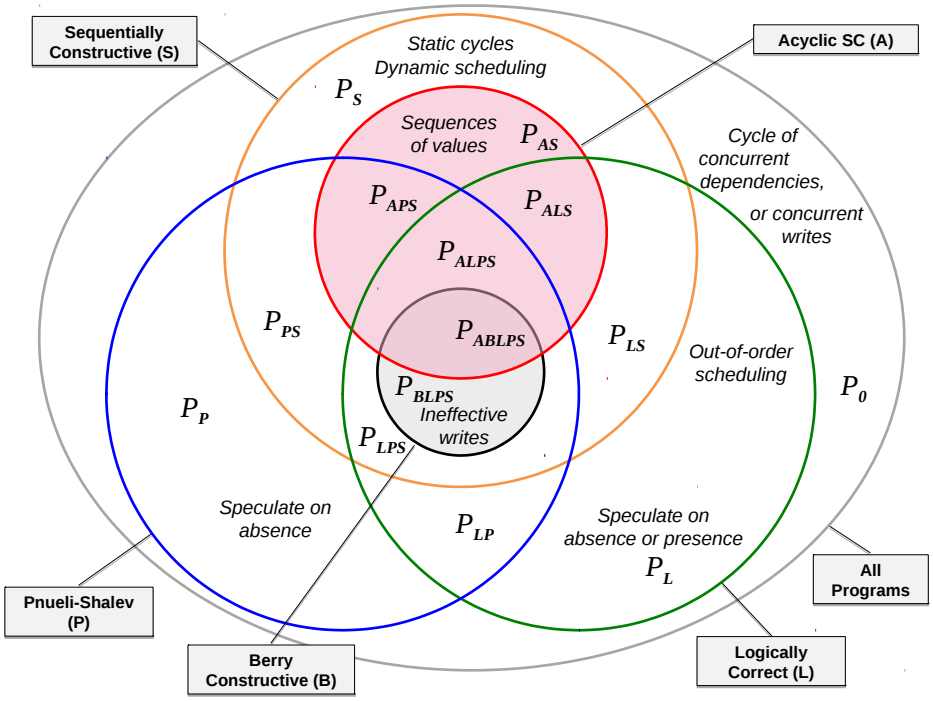


Figure 2.6. Relationships of synchronous program classes [HMA+13]

2.3 Used Technologies

The reference implementation of the SC MoC, all related languages, and compilation approaches are part of the Kiel Integrated Environment for Layout Eclipse Rich Client (KIELER) project. The project uses the Eclipse platform and various Eclipse-based technologies to enhance graphical model-based design of complex systems.
The implementation presented in Chapter 5 is also part of the KIELER project. This section gives a brief overview of technologies used in the context of this thesis.

2.3.1 Eclipse

Eclipse[1] is an open source Integrated Development Environment (IDE) managed by the Eclipse Foundation. It was initially created for developing Java applications, but evolved to a highly adaptable and expendable framework suitable for many domains of software development and different programming languages. The architecture of Eclipse is based on *plug-ins*, implementing the Open Services Gateway initiative (OSGi) specification and providing a very flexible modular design. Plug-ins are designed to carry a modular fragment of functionality. They can provide and use *extension points* which allow other plug-ins to contribute their functionality, for example to integrate new menu entries into the user interface.

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

The Eclipse Rich Client Platform (RCP) provides the basic platform and infrastructure for the IDE and allows to compose further plug-ins from the Eclipse Project and self-developed ones into a domain-specific development environment. The KIELER project is such an RCP tailored to allow graphical model-based system design.

EMF

The Eclipse Modeling Framework (EMF)[2] is an Eclipse project contributing fundamental functionality for model-driven development. EMF provides tools for meta-modeling, allowing the user to design and specify the elements and structure of a model. Moreover, EMF has the ability to generate Java code from meta-models. The generated interfaces and classes implement the different model elements and provide methods to correctly structure a model based on the model element relation in the meta-model. In addition to that, the framework creates factories and utilities for model instantiation, manipulation and persistence. Furthermore, EMF allows to generate editors for end-users to create and edit model instances.
The structured data models used in the implementation of languages such as SCL, SCG, SCCharts, and Esterel in the KIELER project, are based and generated from EMF meta-models.

Xtext

Xtext[3] is a framework for developing programming languages and Domain Specific Languages (DSLs). It provides a grammar definition language similar to the extended Backus-Naur form to design new DSLs. Based on the DSL grammar, Xtext is able to generate a parser, serializer, and a full-featured editor including code-completion, auto-formatting, and syntax-highlighting. Furthermore, Xtext uses model-based data structures based on EMF for the implementation of the DSLs.
The KIELER project uses Xtext for implementing languages such as SCL, Esterel, or a textual representations for SCCharts.

2.3.2 KIELER

The KIELER[4] project is an open source research project of the Real-Time and Embedded Systems Group at Kiel University. KIELER itself is an Eclipse RCP providing the implemented research results in the area of graphical model-based design of complex systems. The KIELER project covers different aspects of this wide research area. Figure 2.7 presents an overview of the sub-projects in KIELER.
One aspect of graphical model-based system design is layout. A graphical model benefits from its intuitive reception in contrast to plain text. However, this requires that the model is well structured and readable, thus having a useful layout. Since manual layout of graphical models

[2] http://www.eclipse.org/modeling/emf
[3] https://eclipse.org/Xtext
[4] http://rtsys.informatik.uni-kiel.de/kieler

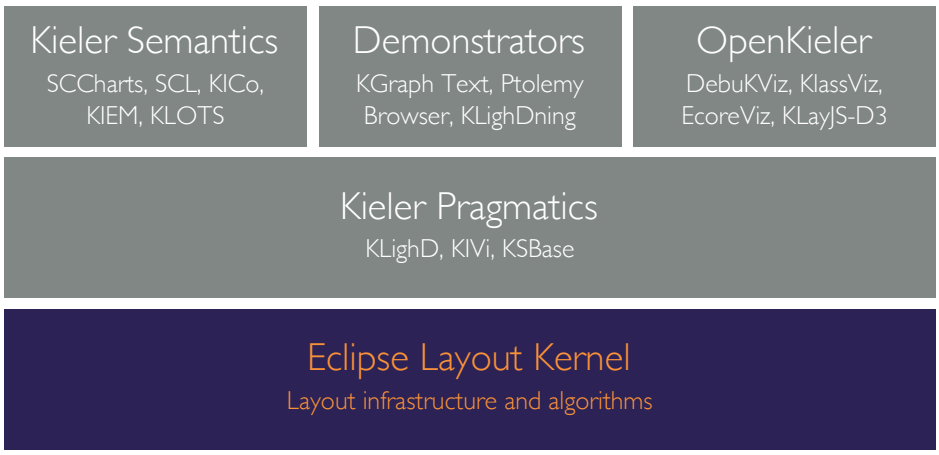


Figure 2.7. KIELER project overview[6]

takes notable amounts of time and manpower, the research of the KIELER project in the area of modeling *pragmatics* [Fuh11] focuses on automatic layout. The Eclipse Layout Kernel (ELK)[5] provides the infrastructure and algorithms to automatically layout graphs including hierarchy and ports. ELK has recently become an official Eclipse project to feed back the results in the area of automatic layout back into the Eclipse community.

Another part of KIELER pragmatics is the KIELER Lightweight Diagrams (KLighD) framework [SSH13]. Based on the concept of transient views, KLighD features the synthesis of diagrams from models using automatic layout. Furthermore, KLighD provides many diagram exploration techniques such as collapsing or expanding areas or filtering diagram elements based on the context.

The technology of lightweight diagrams and automatic layout are applied to different demonstrators, for example the OpenKIELER project which uses automatically layouted diagrams for visualizing class diagrams or debugging data relation in Eclipse.

The KIELER semantics project is focused on the semantics of systems, especially synchronous semantics. KIELER provides implementations for different synchronous languages, such as Esterel and most importantly the SC languages. Furthermore, KIELER offers a flexible compiler, providing multiple compile chains. SCCharts are a convenient example for how the different areas in the KIELER project come together. SCCharts are modeled side-by-side with a textual language and a transient view of the model, automatically generated and updated by KLighD and layouted by ELK. A resulting SCCharts diagram is presented in Figure 2.5 on page 12.

[5] http://www.eclipse.org/elk

[6] http://rtsys.informatik.uni-kiel.de/confluence/display/KIELER/Overview

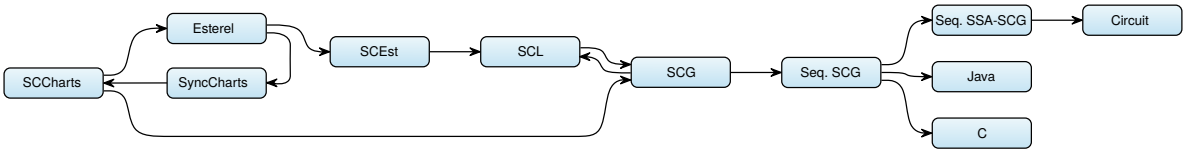


Figure 2.8. KIELER compilation overview

KIELER Compiler

The KIELER Compiler (KiCo) [MSH14] is a compilation framework for model-based programs, implementing the Single-Pass Language-Driven Incremental Compilation (SLIC) approach. KiCo is designed to flexibly compose transformations which handle a single feature or language translation into a one-pass compilation chain to compile an input model. Additionally, KiCo dynamically decides the necessity and ordering of transformations for a specific compilation target based on the input requirements of each transformation. If a transformation cannot handle a language feature contained in the input model, another transformation converting this feature into a compatible from will be executed first.
Based on KiCo the KIELER semantics project provides different compilation approaches and language translations. Figure 2.8 illustrates an abstract overview about the different source and target languages the current compiler is capable of. The nodes represent the models or languages, either in general or in special form, such as SSA or Sequentialized (Seq.). The edges indicate which translations are available, but the actual transformations for compilation between the models are not shown in the Figure. Note that the `Seq. SCG` represents the result of the dataflow compilation approach, mentioned in Section 2.2.

Chapter 3

Related Work

The synchronous MoC inspired different synchronous languages following different programming paradigms. Imperative languages such as Esterel or Quartz [Sch10], dataflow languages, for example Lustre [HCR+91] and SIGNAL [GGB+91], and also graphical modeling languages such as Synchronous Charts (SyncCharts) [And96].
This thesis mainly focuses on the constructiveness of Esterel and the SC MoC, but in the context of synchronous languages there are further approaches to define constructiveness. Pnueli and Shalev define a statecharts semantics with a globally consistent execution and signal state [PS91]. The concept allows to speculate on the absence of a signal using enabling functions. Another approach by Boussinot [Bou98] and as well Berry [Ber02] presents a concept of *logical coherence* which allows speculation on both absence and presence of signals.
Some synchronous languages share similar semantic constructs or structures, which allows to translate the languages into one another. Section 3.1 presents approaches which are related to this thesis because they concern the translation of a synchronous languages from or into Esterel.
Furthermore, some work referenced in Section 3.1 also state the SSA form as potentially suitable for transforming the concept of multivalued variables into signals with a globally consistent state per tick. The basic SSA form was developed only for sequential programs without parallelism. Hence, Section 3.2 presents related work which uses or extends the concept of SSA to programming languages with explicit parallelism or a synchronous MoC.

3.1 Translations Regarding Esterel

Translating other languages into Esterel requires rules to transform the syntax elements. However, if the resulting Esterel program should be semantically equivalent, also the semantics of the source program must be adapted to Esterel.

3.1.1 SyncCharts

Charles André developed the SyncCharts language to create a graphical modeling language especially tailored to the synchronous paradigm [And96]. SyncCharts is inspired by Harel's StateCharts formalism [Har87] and provides hierarchical state machines with parallelism and preemption mechanisms. Just like Esterel, SyncCharts facilitates the representation of reactive systems by a well-defined process algebra with synchronous signals. The most important aspect is that the formal semantics are fully compatible with the semantics of Esterel. André

```
1 module ABRO:
2 input A, B, R;
3 output O;
4 loop
5    [ await A || await B ];
6    emit O
7 each R
8 end module
```

Listing 3.1. The ABRO program in Esterel [Ber99]

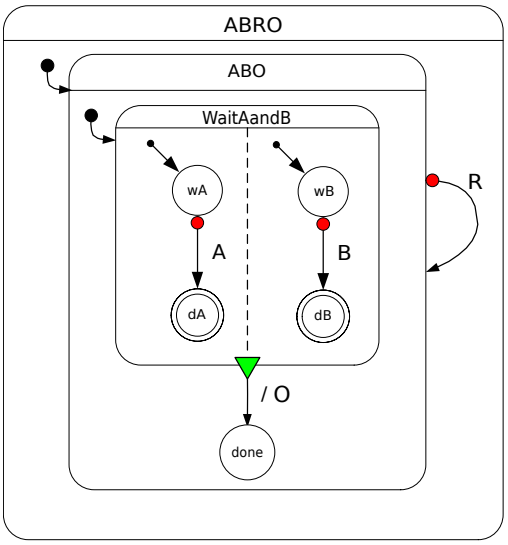


Figure 3.1. The ABRO program in SyncCharts [And03]

presents rules which allow to translate every SyncCharts into a semantically equivalent Esterel program. Listing 3.1 shows the ABRO program, the "Hello World!" of synchronous languages, written in Esterel and Figure 3.1 illustrates the equivalent SyncChart. ABRO emits the output O as soon as the two inputs A and B have occurred and resets this behavior each time the input R occurs.
The other direction, translating Esterel to SyncCharts, was done by Kühl [Küh06] and implemented by Rüegg [Rüe11], in the context of the KIELER project.
Since SyncCharts are designed to share the same semantics as Esterel, the translation presented by André has more structural than semantical challenges. In contrast to that, this thesis focuses on the adaption of SC semantics into Esterel.

3.1.2 SCCharts

A transformation from SCCharts to Esterel is developed by Nasin [Nas15]. Since the graphical syntax of SCCharts is inspired by SyncCharts, they share similar syntactic and semantic constructs. Due to the equivalence to Esterel, SCCharts also shared these with Esterel. Nasin presents translation rules for the different syntactic elements and proves of their behavioral correctness based on the approach of André.
Due to the incremental definition of SCCharts based on a minimal set of *core* elements, any SCChart can be translated into SCL. Consequently, a translation from SCL into Esterel indirectly covers the translation of SCCharts. However, this approach is contrary to utilizing the more advanced language features Esterel and SCCharts share. The most important aspect, for this thesis, is that Nasin did not translate the concepts of SC. He restricts his approach and focuses on SCCharts that comply with the synchronous MoC of Esterel.

3.1.3 SCL

Another form of translating Esterel is done by Rathlev presenting a transformation from Esterel into SCL [Rat15]. With the successful translation of the Esterel into SCL, he shows that SCL is suitable represent any Esterel program. Furthermore, with SCL built with less statements than the Esterel *kernel* language, SCL is capable of achieving at least the same expressiveness as Esterel. Additionally, the translation to SCL enables the use of the SC compiler toolchain implemented in KIELER. In his section about future work, Rathlev presents first thoughts about the translation from SCL into Esterel.

Based on the successful translation, Rathlev et al. introduce Sequentially Constructive Esterel (SCEst), enhancing Esterel with features from the SC MoC [RSM+15]. The Esterel language is extended by a minimally invasive set of statements, alongside semantics definitions for SC signals. This way SCEst allows to write SC programs in Esterel, which can be compiled into executable code by first translating them into SCL.

Rathlev et al. also face the discrepancy between signals and variables in Esterel, when representing multiple different values within a single tick. They present different approaches for handling this problem, in particular the concept of SSA.

3.2 Static Single Assignment

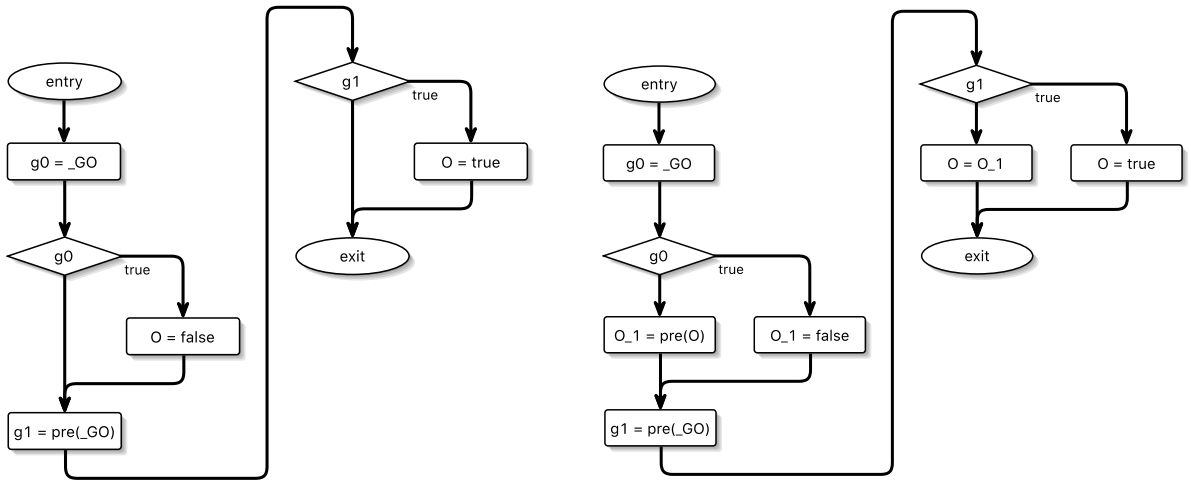
The SSA form was developed by Wegman, Zadeck, Alpern, and Rosen for code optimization such as global value numbering, dead code elimination, and constant propagation with conditional branches [AWZ88; RWZ88; WZ91]. Cytron et al. describe the efficient computation of SSA form using dominance frontiers [CFR+91].

In the SSA form each variable is split up into *versions*, such that each version is is assigned a value only once in the program code. When reading a variable which is split up into versions, a *ϕ-function* resolves the correct value from the eligible versions.

Since SSA performs by definition only one assignment per variable version, it seems predestined for translating variables into signals.

C/C++ to SIGNAL using SSA

Kalla et al. present an automated translation process for C/C++ model into SIGNAL to enable formal verification methods [KTB+06]. SIGNAL is a multi-clocked synchronous dataflow language based on equations over signals. The automated translation process first creates the SSA intermediate representation of the C/C++ components using the GNU Compiler Collection (GCC). Subsequently, the SSA CFG is transformed into SIGNAL using a pattern-based translation scheme.



(a) Sequentialized SCG **(b)** SSA SCG

Figure 3.2. Transformation of Sequentialized SCG into SSA form without ϕ-functions [Ryb16]

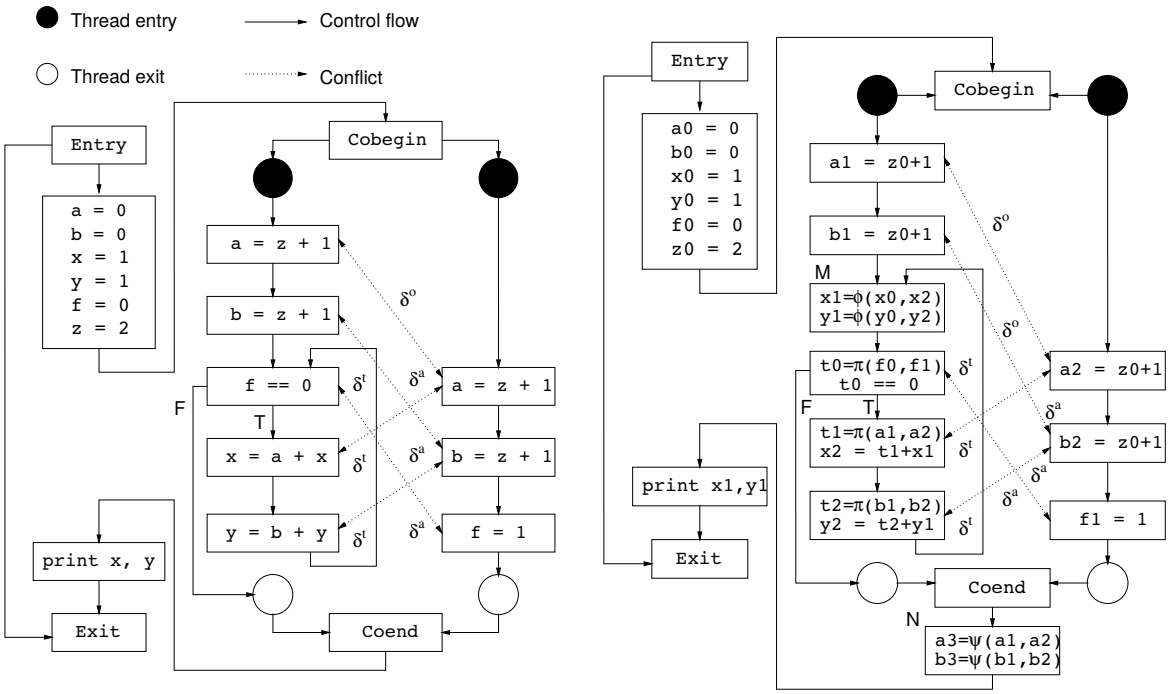
Besnard et al. extend this concept to parallel C/C++ programs, supporting a small subset of the SystemC library [BGM+09]. They encode a non-preemptive scheduler using the clocks of the signals to handle parallel code. As a limiting factor they present that modeling a more complex scheduler, allowing preemption or real parallelism is hard task.

3.2.1 SSA in Hardware Synthesis

SSA has an important role in hardware synthesis because the wires of a circuit can only represent one value at a time. A general consideration of the challenges in hardware synthesis from C-like languages was done by Edwards [Edw05].
Furthermore, SSA is also used and implemented in the context of hardware synthesis for SCCharts [RSM+16]. Both Johannsen [Joh13] and Rybicki [Ryb16] used SSA to prepare a Sequentialized SCG for their circuit representation. Figure 3.2 shows the transformation of a Sequentialized SCG into SSA SCG. Note that the ϕ-function is already resolved into two assignments, one in each branch. In the circuit, such assignments are merged with a multiplexor.
The basic algorithms and the SSA form are well suited for this use case because the SSA transformation is performed on the Sequentialized SCG. Consequently the SCG does not contain any concurrency or feedback.

3.2.2 SSA for Explicitly Parallel Programs

The basic SSA concept is defined for sequential programs, Srinivasan et al. extend this concept to programs with explicit parallelism [SHW93]. They use the parallel sections semantics to define code blocks which are executed concurrently. These sections can have wait synchro-



(a) CCFG **(b)** CCFG in CSSA form

Figure 3.3. Transformation of a CCFG into CSSA form [Lee99]

nization clauses, but the transfer of control in or out of a parallel section is not supported. When translating into SSA form, multiple CFGs are created for each parallel section and for the surrounding program. Afterwards, a parallel dominance frontier analysis is performed to place the ϕ-functions. Additionally, a ψ-function is introduced to merge write accesses from different parallel sections.

However, the restriction to copy-in/copy-out semantics prevents the result of a parallel execution from depending on a particular interleaving of statements. A more general approach is presented by Lee et al. [LMP98; Lee99]. They introduce a Concurrent Static Single Assignment (CSSA) form which handles a more general form of parallel constructs with interleaving semantics and post-wait synchronization. An additional π-function is defined to handle the read access of concurrently written variables inside the threads. The definition of the π-function is nondeterministic due to the interleaving semantics of the concurrent threads. The algorithm for the CSSA also extends the concept of dominance frontiers defined on a special concurrent controlflow graph (CCFG). Figure 3.3 shows the CCFG of an example program in its normal and CSSA form. The CSSA form introduces ϕ-functions in block `M` to merge the values of the loop iterations. The results of the shared variables are merged by ψ-functions in block `N` at the end of concurrent sections. Additionally, π-functions are introduced before each read access on a variable which is concurrently written. The dotted lines marked with δ indicate concurrent write-read and write-write dependencies.

Lee et al. also present an adapted algorithm for sparse conditional constant propagation based on the CSSA.
In comparison to this thesis, Lee et al. present an SSA definition for relatively unrestricted scheduling. However, regarding a synchronous context, deterministic scheduling is more important than optimization based on SSA. Furthermore, when SSA is used to introduce specifically controlled sequentiality into another language, such as Esterel, the definitions for merge functions, such as ϕ-, π- or ψ-functions need to be executable in this language.

Chapter 4

Strict Sequential Constructiveness

Section 1.3 presents the problem of SC programs scheduled and accepted such that they are deterministic but considered speculative in the sense of constructiveness in Esterel. This chapter presents a solution for this problem by restricting SC.
Section 4.1 presents the problem of unrestricted SC programs. Additionally, the sections describes the restriction of SC programs and the approach to detect them. The approach presented in this thesis includes the translation of SC programs into Esterel to check their constructiveness based on the constructive semantics of pure Esterel. The transformation of shared variables and the SC scheduling regime into an Esterel-compatible form requires an SC-specific SSA form, presented in Section 4.2. The section first analyzes and adapts the regular SSA form to the SC domain. However, this approach cannot handle all kinds of SC programs, thus secondly, an SC-specific SSA form is presented. The definition of this form is separated into several subsections which handle the different aspects of SC programs and gradually extend the capabilities of this SSA form. Subsequently, Section 4.3 defines the actual translation of SC programs in SSA form into Esterel. This includes a structural transformation and the encoding of variables into signals.

4.1 Restricting Sequential Constructiveness

In Chapter 1 the program `P10` is presented in Listing 1.3 on page 3. It is a characteristic example for a non-constructive program in the sense of Esterel accepted by the SC MoC. The constructiveness in the sense of Esterel would guarantee that the program forms a constructive circuit. However, this property does not hold in this case. Constructive boolean logic circuits provide two properties for all input sequences in the care set [SBT96].

- They yield a unique boolean solution for each output.
- They guarantee electrical stabilization of all outputs, independent from any wire or gate delays.

In other words, constructiveness ensures a predictable deterministic behavior on circuit level, independent from the actual implementation in hardware. This is also a desired property for the SC MoC.
In the example of `P10` the variable `x` can be 0 or 1. More precisely, it must be both, if both concurrent assignments are executed because no explicit override relation exists. The 1 is always assigned, but the assignment to 0 depends on the value of `y`, and `y` has either value of

x or is 0. In a circuit the wire of y can have the value 0 depending on wire initializations and gate delays. If y is 0, the wire representing x is supposed to have two different values at the same time, which is not possible in a circuit. This is a sufficient reason to reject the program P10.

The target of this thesis is to restrict the class of SC programs to those which can be considered constructive in the sense of Esterel. These programs are denoted as *Strictly Sequentially Constructive* (SSC).

4.1.1 Detecting Strict Sequential Constructiveness

The restriction to SSC requires the detection of constructive SC programs in the sense of constructive circuits. The constructive semantics of pure Esterel provides one definition by transforming programs into circuits and checking their constructiveness using Malik's procedure [SBT96; Ber02].

Therefore, one solution to detect SSC programs is to provide a similar translation and check the constructiveness of the resulting circuit, using ternary values function evaluation. Another is to extend the definition of the SC MoC by additional restrictions to comply with the constructive concept.

However, this thesis presents a more practical approach. Instead of changing the MoC or simulating circuits directly generated from SC programs, this approach uses the capabilities already provided by Esterel. Translating SC programs into Esterel to check their constructiveness according to the constructive semantics of Esterel provides a well-defined foundation for SSC programs. The Esterel compiler[1] allows to perform a full-featured constructiveness analysis for pure Esterel programs based on a circuit analysis. Hence, the Strict Sequential Constructiveness provides a physical foundation for the SC semantics in the constructive circuit semantics of Esterel.

Since the SC MoC extends the synchronous MoC including Esterel, it provides features not present in Esterel. Especially deterministic shared variables with multiple values per tick raise challenges when translating them into a more restricted language, such as Esterel. Hence, the translation needs to transform variables with possibly multiple values per tick into signals with one globally consistent value. To achieve this property the concept of SSA is used. It ensures that variables which are assigned with multiple values are split up into copies of that variable to perform only a single assignments per variable. This way the concept of variables can comply with signals. Additionally, the resulting Esterel programs must be semantically equivalent to their source programs in order to imply the constructiveness of the SC source program based on the constructiveness of the Esterel program. Consequently, the translation must consider and translate the SC semantics into Esterel, especially the iur protocol.

To summarize, this approach detects SSC programs by transforming SC programs into semantically equivalent Esterel programs and test their constructiveness.

[1] http://www-sop.inria.fr/esterel-org/filesv5_92/home.htm

4.2 SSA Form for Sequentially Constructive Programs

This section presents the SSA transformation for SC programs. To separate the definition of the transformation into modular steps, SC programs are divided by four orthogonal structural features.

- Concurrency
- Delay
- Cycles
- Updates

The class of concurrent programs separates programs with both sequentially and concurrently composed statements from strictly sequential programs. The class of delayed programs contains programs with at least one *pause*, whereas the absence of this feature classifies instantaneous programs. The cycles feature separates cyclic programs form acyclic. Note that is program is both cyclic and delayed, it does not imply that all cycles are delayed loops, because a pause must not necessarily occur inside the cycle. The class of programs with updates, indicates whether the programs contains any relative writes considered updates or not.

The following sections use these classifications to gradually define the SSA transformation. At first, Section 4.2.1 presents the regular SSA form. Even if the transformation cannot handle concurrent SC programs, the section illustrates the handling of delay, cycles and updates in sequential programs. The features are used to incrementally extend the number of handled program classes. Consequently, if the handling of cycles is described after delays, then the considered input programs for handling this feature are both delayed and non-delayed programs. The same strategy is used in the subsequent sections to present the SC-specific SSA form. Section 4.2.2 presents the general concept of handling concurrent SC programs. In Section 4.2.3 the rules for constructing merge expressions are defined. These rules handle programs which can be concurrent but must be instantaneous, acyclic and do not contain updates. Subsequently, the concept is extended to delayed (Section 4.2.4), cyclic (Section 4.2.5), and programs using updates (Section 4.2.6). Disregarding some restrictions mentioned in the corresponding sections, the concept handles all four SC program classes. Section 4.2.7 additionally presents a transformation ensuring the compliance of the SSA form with the defined interface of the program.

All sections illustrate their concepts with example programs. Note that the source code is mostly given in SCL, but sometimes a corresponding or processed SCG is presented for easier perception of the relations between statements. However, both representations, SCL and SCG, are fully equivalent, illustrated in Table 2.3 on page 10.

4.2.1 Regular SSA

The SSA form is developed to provide an intermediate representation for sequential programs which exposes the direct links between the definitions and uses of variables. These *def-use* chains facilitate many compiler optimizations, such as *constant propagation* or *dead code elimination*. Since the SC MoC is designed in the sense of sequential programming paradigms and based on an extended CFG formalism, the SCG, the SSA concept and algorithms are applicable to the domain of SC programs. Section 3.2.1 presents that SSA is successfully used with Sequentialized SCGs, which do not contain any concurrency, delay or cycles. Hence, it needs to be evaluated how the regular form of SSA is applicable to general SCGs.
First of all, the SSA form ensures that each use of a variable, declared by the programmer, is reached by exactly one assignment defining the value for that use. This property of *single reaching definition* is achieved by two core aspects of SSA:

1. Separation of variables into versions
2. Value merge, using ϕ-functions

Each variable is split up into multiple versioned variables such that each version is defined by exactly one assignment. This separates the different values assigned to a variable and allows the distinction of the incoming definitions when using a variable. The next step is to select the correct variable version for the use. In the face of branching controlflow, the ϕ-function is introduced. This pseudo assignment function has the purpose of merging values from different incoming controlflow paths. The ϕ-function has the form $U \leftarrow \phi(V_0, V_1, \ldots, V_n)$ where U and V_i are versions of the same variable. U is the version to carry the merged value and V_i are the incoming variable versions based on the controlflow predecessors of the point where the specific ϕ-function is placed. The operands are in an arbitrary fixed order, where the j-th operand corresponds to the j-th predecessor. If the controlflow reaches a ϕ-function from its j-th predecessor, it assigns the value of j-th operand to U.
Another important aspect of SSA is the placement of ϕ-functions. In general it is sufficient to place a ϕ-functions for every variable at every point where the controlflow joins, but this easily produces many superfluous ϕ-functions. Cytron et al. [CFR+91] present an algorithm for placing a *minimal* number of ϕ-functions based on *dominance frontier* analysis. According to Tarjan [Tar74], a node n in a CFG *dominates* another node m, if n appears in every path from the program's entry node to m. For $n \neq m$, n *strictly* dominates m. If n is the closest strict dominator of m on any controlflow path to m, then n is denoted the *immediate dominator* of m. A *dominator tree* contains the nodes of a CFG and edges based on their immediate dominator relation. The dominance frontier DF(n) of a node n is the set of all nodes where n dominates any predecessor, but does not strictly dominate the node itself. The procedure of Cytron et al. is based on the fact that whenever node n assigns a variable, then any node in DF(n) requires a ϕ-function for that variable.
When using SSA directly for the generated code and not only as an intermediate representation for optimizations, the problem occurs that the ϕ-function itself is not directly executable. The definition requires to detect the controlflow path used to reach ϕ-function to select the

corresponding value. However, this is not possible for actual controlflow without explicitly adding executed statements. Hence, one possible transformation into an executable form is to create assignments to the variable, assigned by the ϕ-function, in each of the incoming controlflow paths. This is the same procedure used for circuit translation, presented in Section 3.2.1. Subsequent optimizations such as *copy propagation* can further reduce these additional variables. Regarding the translation into Esterel code, the translation into multiple assignments on the same variable version is acceptable, as long as only one of these is executed in a tick. For example, this is the case for mutually exclusive conditional branches.
To illustrate the regular SSA transformation, Listing 4.1 presents the source code of the `AbsoluteValue` program written in SCL. The program calculates for an input `i` its absolute value and conveys the result in the output variable `o`. The sign of the input number is determined in `x` and the input is multiplied such that the sign cancels out. Figure 4.2 shows the corresponding SCG in 4.2a untransformed form, 4.2b SSA form and 4.2c SSA form with transformed ϕ-functions. The ϕ-function is introduced to merge the values of the different conditional branches. None of the statements in the branches dominate the reading statement but all precede it. Thus, the reading statement is in their dominance frontier. The dominator tree providing the underlying data structure for this analysis is depicted in Figure 4.1.

Concurrency

The major limitation of the regular SSA form is concurrency. As described in Section 3.2.2, the SSA form and especially the ϕ-function is not capable of merging values from concurrent threads or considering concurrent assignments based on thread interleaving.
Listing 4.2 presents the program `ConcurrentWrites` which concurrently assigns the variable `x` to different values based on the inputs `i` and `j`. The corresponding SCG is shown in Figure 4.3a, where the write-write dependency indicates a potential conflict between the concurrent absolute writes. Figure 4.3b illustrates the SCG in regular SSA form. The critical and especially incorrect part is the ϕ-function node after the join node. Even if the ϕ-function node has only one incoming controlflow edge, it is introduced to merge the incoming controlflow of the join node. The notation of placing the ϕ-function after the join corresponds to the concepts of SSA for explicitly parallel programs, presented in Section 3.2.2. However, placing a ϕ-function in this case is incorrect, because its definition is restricted to situations where the ϕ-function is reached from exactly one of its predecessors. The CSSA form presented in Section 3.2.2 would use the ψ-function for merging the result of the two threads. Additionally, if the variable `x` would be read in any of the threads, a π-functions would be inserted before the read access to resolve possible interleaving of concurrent statements. Figure 4.3c shows the SCG in CSSA form.
However, the CSSA form may be sufficient as an intermediate representation for detecting def-use chains, but translating a CSSA program directly into executable code arises the problem that the ψ- and π-functions are not executable. They are defined to select the value based on the dynamic and non-deterministic interleaving of the writing statements in concurrent threads, which cannot simply detected at runtime. Moreover, the SC MoC defines specific

```
1  module AbsoluteValue
2  input int i;
3  output int o;
4  int x;
5  {
6  x = 0;
7  if i > x then
8    x = 1
9  else
10   if i < x then
11     x = -1
12   end
13 end;
14 o = i * x
15 }
```

Listing 4.1. The `AbsoluteValue` program

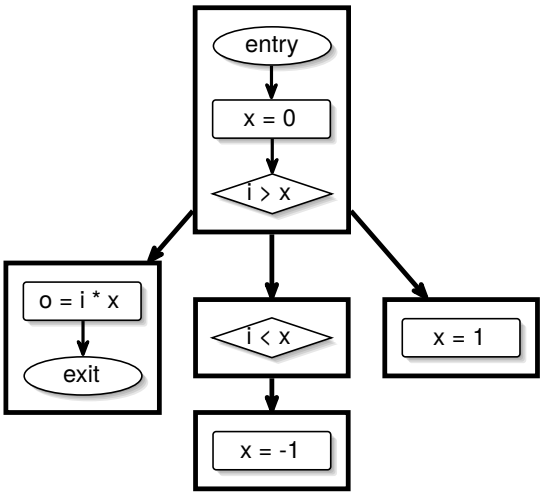
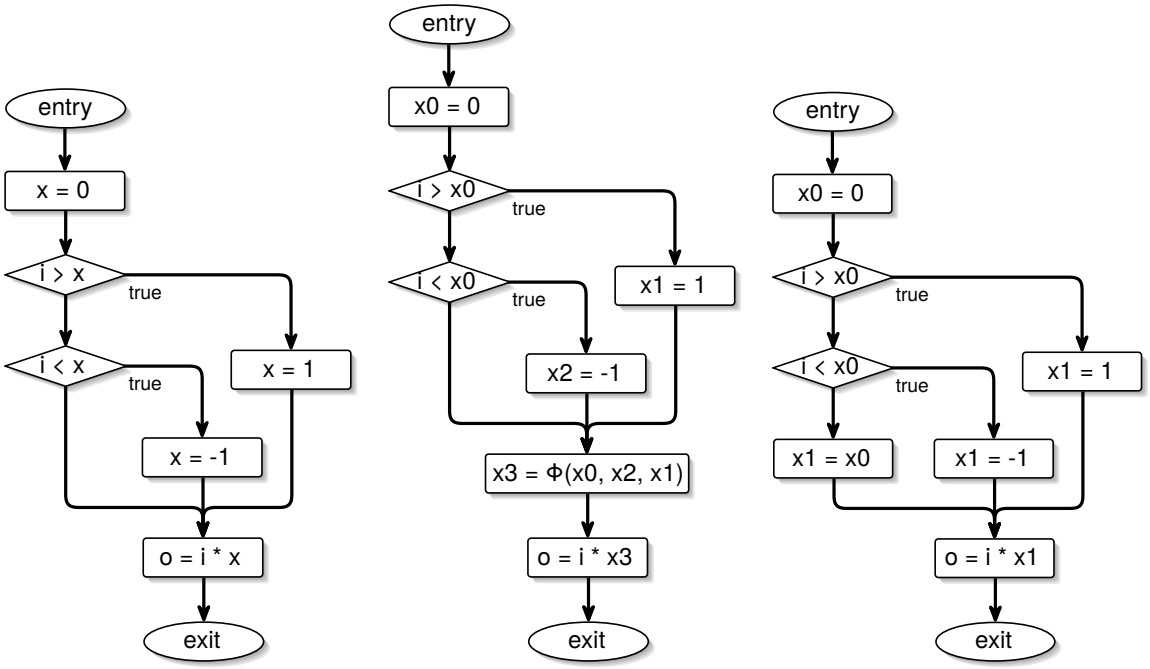


Figure 4.1. Dominator tree of the `AbsoluteValue` program



(a) Original

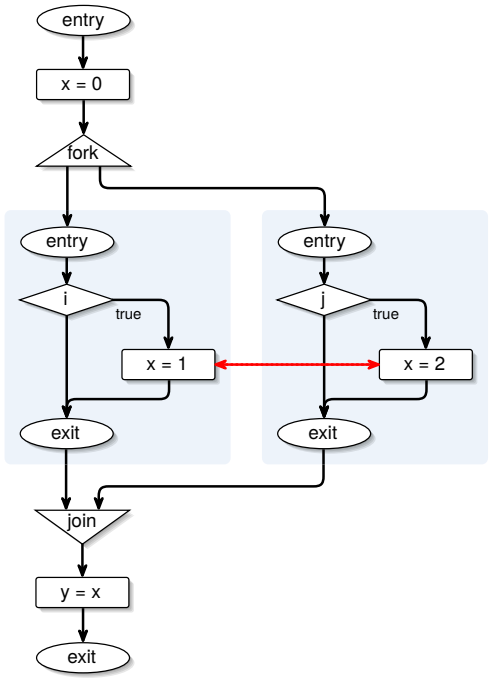
(b) SSA

(c) SSA with transformed ϕ-function

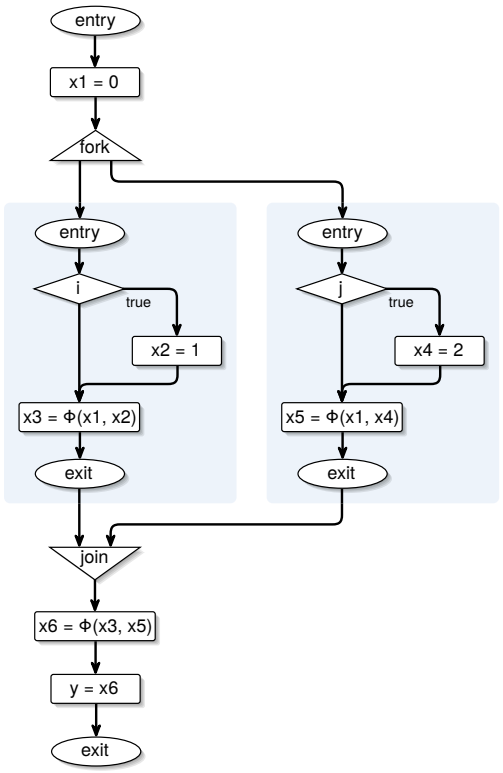
Figure 4.2. SCG representation of the `AbsoluteValue` program

```
1  module ConcurrentWrites
2  input bool i, j;
3  int x, y;
4  {
5    x = 0;
6    fork
7      if i then
8        x = 1
9      end
10   par
11     if j then
12       x = 2
13     end
14   join;
15   y = x
16 }
```

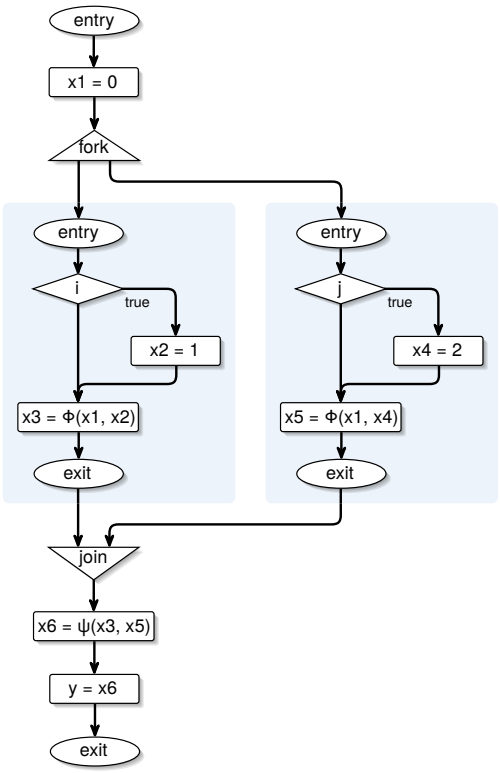
Listing 4.2. The `ConcurrentWrites` program



(a) Original



(b) SSA form with incorrect behavior



(c) CSSA form

Figure 4.3. SCG representation of the `ConcurrentWrites` program

i	j	y
false	false	0
false	true	2
true	false	1
true	true	**reject**

Table 4.1. Expected results for `y` in the `ConcurrentWrites` program depending on the input values

scheduling restrictions for concurrent reads and writes on the same variable. Alongside these scheduling constraints, a program can also be rejected as non-SC, if nodes are conflicting. Thus, not only the current value of a variable matters, but also how and where in the program it is assigned. Table 4.1 presents the values of the output variable `y` for different input values with an SC-scheduling. Note that if both `i` and `j` are true, the program is rejected because of the conflicting concurrent initializations. Only if this situation can never occur, the program is accepted.
This example illustrates that the ψ- and π-functions cannot be used for SC programs which use concurrency because they lack in expressiveness for these scheduling constraints. Furthermore, the regular SSA form is only applicable for strictly sequential SC programs. Section 4.2.2 presents the SC-specific SSA form for programs using SC concurrency.

Delay

Variables in the SC MoC have a persistent value across ticks. Using pause statements in concurrent programs might influence the time when assignments are executed, including conflicting writes. In a strictly sequential program a pause does not affect the value of a variable. However, there is one exception, input variables. Input variables are read from the environment at the beginning of every tick. Consequently, each pause statement is an implicit assignment to all input variables. This implicit behavior is specific to the synchronous domain and not handled by the regular SSA definition.
Listing 4.3a presents the program `SequentialIO` that reads and writes an input output variable `x`. If the input `i` is true in the first tick, the variable `x` is doubled in line 6, then the program is paused and in the next tick `x` is divided by 2. Since `x` is an input output variable the value divided in the second tick might not the same conveyed in the first tick. If the input `i` is false in the first tick, `x` is incremented by 1 twice and the programs terminates in this instant.
If the input variable is only read by the program and never assigned, such as `i`, then input behavior is uncritical because the SSA form does not merge the definitions of the variable. However, if input variables are written by the program, such as `x`, then the local definition overrides the value of the environment. Consequently, both definitions must be taken into account.
A solution to this problem is to post-process the SSA form and insert assignments for each input variable after each pause statement. These assignments move the input value into the

```
1  module SequentialIO
2  input bool i;
3  input output int x;
4  {
5    if i then
6      x = x * 2;
7      pause;
8      x = x / 2
9    else
10     x = x + 1
11     x = x + 1
12   end;
13 }
```

(a) Original

```
1  module SequentialIO-SSA
2  input bool i;
3  input output int x;
4  int x0, x1, x2, x3, x4, x5;
5  {
6    x0 = x;
7    if i then
8      x1 = x0 * 2;
9      x = x1;
10     pause;
11     x1 = x;
12     x2 = x1 / 2
13   else
14     x3 = x0 + 1;
15     x4 = x3 + 1
16   end;
17   x5 = φ(x2, x4);
18   x = x5
19 }
```

(b) SSA form

Listing 4.3. The `SequentialIO` program

reaching variable version. As long as there are no pause statements concurrent to each other, this solution is sufficient.
A similar problem occurs when using output variables, such as `x`. The environment expects the program to convey the final value of an output variable at the end of every tick. To meet this requirement, at the end of the program and before each pause statement, the value of the local reaching definition of the output variable version is written into the output variable.
Both solutions require the separation of internally assigned variable versions from the interface variables of the module. This is necessary for the environment to detect interface variables by their name. Hence, the renaming of the variables must not affect the input or output variables but create new definitions for local versions. Section 4.2.7 discusses this aspect in more detail.
Listing 4.3b illustrates the program `SequentialIO` in SSA form with correct handling of the input output behavior, producing a semantically equivalent program. The output value for `x` is conveyed in line 9 before the pause and when the programs terminates in line 18. The input value of `x` is written into the currently reaching definition after the pause in line 11. Moreover, the naming of the interface is retained by introducing local versions for the variable.

Jumps and Cycles

The regular SSA form is capable of handling all sequential CFG structures correctly, including cycles. The entry of a loop has more than one incoming controlflow edge causing the placement of a ϕ-function, which handles the value merge of the variable in each iteration.

```
1  module Factorial
2  input int i;
3  int n, f;
4  {
5    n = i;
6    f = 0;
7    if n >= 0 then
8      Loop:
9      f = f * n;
10     n = n - 1;
11     if n > 1 then
12       goto Loop
13     end
14   end
15 }
```

(a) Original

```
1  module Factorial-SSA
2  input int i;
3  int n0, n1, n2, f0, f1, f2;
4  {
5    n0 = i;
6    f0 = 0;
7    if n0 >= 0 then
8      Loop:
9      f1 = φ(f0, f2);
10     n1 = φ(n0, n2);
11     f2 = f1 * n1;
12     n2 = n1 - 1;
13     if n2 > 1 then
14       goto Loop
15     end
16   end
17 }
```

(b) SSA form

Listing 4.4. The `Factorial` program

Listing 4.4a shows a cyclic program computing the factorial of a positive input number. The program performs `i` iterations to multiply the descending value of `n` into `f`. Listing 4.4b presents the program in SSA form, introducing ϕ-functions in lines 9 and 10 to merge the definitions of `f` and `n` in each iteration.
As long as the variables in the loop body are not accessed concurrently, this SSA form is sufficient for merging the variable values. Regarding the translation into Esterel, there are both structural and behavioral problems with cyclic CFGs, further described in Section 4.3.1 and Section 4.3.2.

Updates

Since the SC MoC does not restrict relative writes in a non-concurrent context, the regular SSA form can handle them normally. Updates simply read the value of the referenced variable, provided by the single reaching definition, apply their operation, and write back the result. Hence, they are not handled differently form absolute writes, for example the assignment to `o` depicted in Figure 4.2b.

4.2.2 SC-specific SSA Form

The previous section points out that the regular SSA form is not capable of handling the deterministic concurrency of SC. Furthermore, the CSSA form presented in Section 3.2.2 lacks in expressiveness considering the scheduling constrains of the SC MoC and more importantly cannot simply translated into an executable form.

An executable SC-specific SSA has to represent the sequential and concurrent relations between assignments, including the distinction between initializations and updates to consider the additional constraints for concurrent scheduling. Another important aspect when using the SSA form to create executable code with SC semantics is that the definition of the function performing a value merge has to detect conflicting writes. This means that the function merging concurrent values has to detect if and how the value was written by the concurrent threads to correctly merge the value or reject the whole program. Conflicts occur for example when two concurrent nodes with writing different absolute values are active in the same tick. Listing 4.2 on page 29 depicts a program with a potential conflict. If both `i` and `j` are true in the same tick than both assignments, in the lines 8 and 12, are executed and assign conflicting values to `x`.

To detect whether an assignment node is active in a tick, the variable value is augmented with a signal. The signal is emitted when a value is assigned to that variable and according to the classic signal semantics it is reset to absent at the beginning of every tick. Thus, each value comes alongside an additional signal which indicates the presence or validity of the actual value. The notation for a variable x_i, renamed by SSA and carrying an augmented value, is $\langle x_i^p, x_i \rangle$, where x_i^p is the presence signal and x_i the actual variable value. If the signal is absent, the value must not be read because the value is invalid with respect to current global value of the variable.

Based on these augmented values the SC-specific SSA form has three merge functions: *seq*, *conc*, and *combine*. These functions are composed to merge expressions, handling the merge of all incoming variable versions.

Listing 4.5 presents the definition of the merge functions in a pseudocode notation similar to Esterel. The `present` test checks a signal for its presence and `if` test evaluates a boolean expressions. The `reject` statement indicates that the complete program should be rejected immediately.

seq (Listing 4.5a) The function $seq(x_i, x_j)$ handles the sequential overriding of two variable values. The first parameter x_i is considered ordered sequentially before the second x_j. Hence, if the signal of the x_j indicates that the value is present in the current tick, the function returns x_j, otherwise x_i if it is active. In the case that none of the values are valid, the *seq*-function returns a neutral inactive value. This allows other merge function to ignore the result of this function and facilitates nested merge expressions.

conc (Listing 4.5b) The function $conc(x_i, x_j)$ handles the merge of concurrent absolute writes. If only one of the concurrent threads executes its assignment, the corresponding value is selected, determined by its signal. A conflict can occur, if both x_i and x_j are active in the same tick. In this case their value is tested and if both writes produce the same value, they are considered confluent. Consequently, any of them can be selected, in this definition x_i. If the values differ, then a conflict occurred and the programs must be rejected. In case neither x_i nor x_j are active, the function returns a neutral inactive value.

```
1 seq(⟨x_i^p, x_i⟩, ⟨x_j^p, x_j⟩) :=
2   present x_j^p then
3     return ⟨x_j^p, x_j⟩
4   else
5     present x_i^p then
6       return ⟨x_i^p, x_i⟩
7     else
8       return ⟨absent, nil⟩
```

(a) *seq*

```
1 conc(⟨x_i^p, x_i⟩, ⟨x_j^p, x_j⟩) :=
2   present x_i^p then
3     present x_j^p then
4       if x_i == x_j then
5         return ⟨x_i^p, x_i⟩
6       else
7         reject
8     else
9       return ⟨x_i^p, x_i⟩
10  else
11    present x_j^p then
12      return ⟨x_j^p, x_j⟩
13    else
14      return ⟨absent, nil⟩
```

(b) *conc*

```
1 combine(f, ⟨x^p, x⟩, ⟨x_up^p, x_up⟩) :=
2   present x^p then
3     present x_up^p then
4       return ⟨x^p, f(x, x_up)⟩
5     else
6       return ⟨x^p, x⟩
7   else
8     present x_up^p then
9       reject
10    else
11      return ⟨absent, nil⟩
```

(c) *combine*

Listing 4.5. The definition of merge functions

combine (Listing 4.5c) The function $combine(f, x, x_{up})$ applies an update on a value x, where x_{up} is the absolute update value derived from the expression in the relative write and f the combination function. If x_{up} indicates that the relative write is active, then the value is combined with x via f and returned. Otherwise the function returns x unmodified. In case the update is performed without a valid value x to read, the program is rejected. This case occurs when an update is performed on an uninitialized variable.

The merge functions are designed to be composed to expressions which represent the scheduling relations and allow to resolve the effective value from the incoming definitions. The presented definitions use two arguments for *seq*- and *conc*-functions and one update argument for *combine*. This is a minimal definition with respect to the considered variable versions. It allows to express the same relation between more than two variable versions by nesting the functions into each other and thus iteratively merge the definitions. A definition with more arguments to express such relations could also be possible, but it would increase the size of the definition. For example a *conc*-function definition would have to check all possible combinations of present variable version to compare their values. Thus, the two arguments definition with nesting is used, also with a simpler implementation in Esterel in mind.

Due to the greater number of reaching definitions and interleaving constraints in a concurrent context, it is more complicated to maintain a single variable with the currently valid value, as the regular SSA form does. Hence, the merge expressions are inserted into the program in every read access. Future work may further optimize this procedure.

The program `ConcurrentWrites` presented in Listing 4.2 on page 29 can be translated into a correct SSA form using a merge expression, depicted in Listing 4.6. In this form the value of `x` is resolved by a merge expression in line 15. Based on the definition of the *conc*-function,

```
module ConcurrentWrites-SSA
input bool i, j;
int x0, x1, x2, y;
{
  x0 = 0;
  fork
    if i then
      x1 = 1
    end
  par
    if j then
      x2 = 2
    end
  join;
  y = seq(x0, conc(x1, x2))
}
```

Listing 4.6. The `ConcurrentWrites` program in SSA form using a merge expression

either x_1 or x_2 sequentially overrides the value of x_0. If both writes are active the program is rejected, and if none of them is executed the value of x_0 is assigned to y by the *seq*-function. This behavior complies with the expected output, presented in Table 4.1 on page 30.

4.2.3 Constructing Seq-Conc-Expressions

In the absence of updates and jumps, the merge expressions represent exactly the sequential and concurrent composition of the related assignments. Thus, these expressions can be generated by analyzing the program structure. Table 4.2 shows the recursive patterns for composing merge expressions for a specific variable, based on the structural language constructs in the program. Assignments to other variables are ignored and skipped in the analysis process. Note that for the conditional structure the sequential order is arbitrary, since both branches are mutually exclusive in the absence of loops. In this definition, the then branch is ordered sequentially before the else branch. Furthermore, the pattern do not contain any rule for unconditional jumps, such as `gotos`. The SC MoC prohibits jumps which have targets outside their thread, but this still allows jumps to form structures which cannot easily translated into Esterel, further discussed in Section 4.3.1. Hence, the composition ignores jumps and supports only the given pattern. However, the special case of jumps forming loops is supported and presented in Section 4.2.5.
The translation into SC-specific SSA form can be achieved by performing the following steps:

1. Split and rename variables
2. Insert a merge expression for each read access on a variable.
3. Reduce merge expressions to the locally incoming definitions.
4. Normalize merge expressions.

SCL Structure	Expression

`if (condition) then`
`  // then-branch`
`else`
`  // else-branch`
`end;`
`// next`
	$seq(seq(\texttt{then-branch}, \texttt{else-branch}), \texttt{next})$

`fork`
`  // thread0`
`par`
`  // thread1`
`par`
`  // ...`
`join;`
`// next`
	$seq(conc(\texttt{thread}_0, \texttt{thread}_1, ...), \texttt{next})$

$\texttt{x}_i$ `= e;` `// next`	$seq(\texttt{x}_i, \texttt{next})$

Table 4.2. Structural construction pattern for merge expressions

Listing 4.7 illustrates the structural composition of merge functions into expressions for a more complex program. The program presented in Listing 4.7a performs various absolute writes on the variable x, read by y at the end of the program. The first transformation step is already performed on the program, splitting and renaming the variable. Listing 4.7b illustrates creation of the merge expression for x in the assignment to y in line 21. The expression is separated into corresponding lines, to illustrate the results of the matching pattern.
The result for the read access of x in line 21 is:

$$seq(conc(seq(\texttt{x}_0), seq(\texttt{x}_1, seq(conc(seq(seq(seq(\texttt{x}_2), seq(\texttt{x}_3)), seq(\texttt{x}_4)), seq(\texttt{x}_5)), seq(\texttt{x}_6)))))$$

Reduction

The construction patterns in Table 4.2 are designed to recursively create an expression for a complete program and consequently take all assignments to a variable into account. Thus, depending on the position of the read access where the merge expression is inserted, it must be reduced to contain only the reaching definitions affecting this read. Especially assignments which are sequentially after the read access must be removed because they do not affect the value.
Regarding the SCG representation of a program, this means that the expressions are reduced to preceding variable versions. These are the variables assigned in nodes which are a sequential predecessors of the reading node or have a concurrent write-read dependency to the reading node.

```
1  module ExpressionExample-SSA
2  input bool i;
3  int x0, x1, x2, x3, x4, x5, x6, y;
4  {
5    fork
6      x0 = 0;
7    par
8      x1 = 1;
9      fork
10       if i then
11         x2 = 2
12       else
13         x3 = 3
14       end;
15       x4 = 4
16     par
17       x5 = 5
18     join;
19     x6 = 6
20   join;
21   y = x?
22 }
```

(a) The `ExpressionExample` program

```
1
2
3
4
5  seq(conc(
6    seq(x0
7  ),
8    seq(x1,
9    seq(conc(
10     seq(seq(
11       seq(x2
12     ),
13       seq(x3
14     )),
15   seq(x4
16 )),
17   seq(x5
18 )),
19 seq(x6
20 ))))
21
22 )
```

(b) Construction of the merge expression

Listing 4.7. Example for creating a merge expression

Moreover, not all sequentially preceding definitions actually affect the read value. Considering dominator relations between nodes, there may be sequentially preceding definitions which strictly dominate the reading node. Of all nodes which dominate the reading node, only the closest dominating definition affects the read value. This node is the immediate dominating writer, all preceding definitions are always overridden by its value. Additionally, all definitions succeeding immediate dominating writer and preceding the read node can also affect the read value.

Another aspect when using dominator relations is the concurrent execution. The definition of domination does not use a concept of concurrency and thus a `fork` is handled like an `if` assuming that only one outgoing controlflow edge is used to leave the node. Hence, domination is always a strictly sequential controlflow domination disregarding interleaving. Consequently, the reduction must consider the iur dependencies in the SCG. Additionally, the iur protocol allows, in some constellations, to remove the sequentially immediately dominating writer from merge expression in a concurrent thread. Listing 4.8 shows a motivating example program for such a constellation. x_0 is the sequentially immediately dominating write, but due to the iur protocol and the structure of the program x_1 will always override the value of x_0. Consequently, the assignment in line 6 can be reduced to `y = x1`. The corresponding SCG is depicted in Figure 4.4. In general, all definitions whether they are dominant or not, which are

```
1  module ConcurrentDominantWrite-SSA
2  int x0, x1, y;
3  {
4     x0 = 0;
5     fork
6        y = seq(x0,seq(conc(seq(x1))))
7     par
8        x1 = 1
9     join
10 }
```

Listing 4.8. The `ConcurrentDominantWrite` program

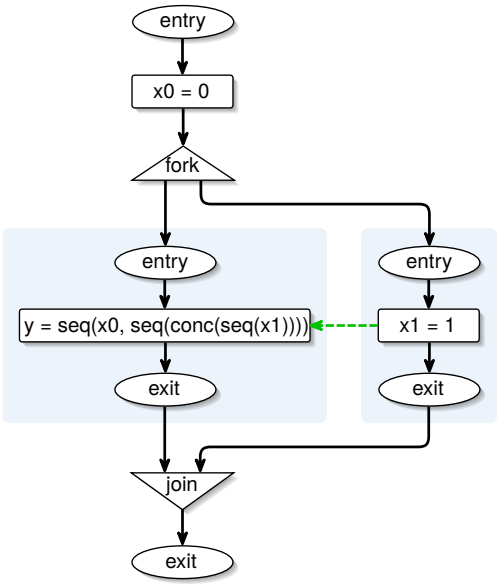


Figure 4.4. SCG representation of the `ConcurrentDominantWrite`

predecessors of a fork node can be removed if any of its threads contains an instantaneous concurrent immediate dominant write. Such a writing node must dominate the exit node of its thread, to ensure that the assignment will always be executed. Furthermore, the node must only be instantaneously reachable from the thread's entry node, to ensure that the it will always be executed in the same tick the thread is started. If there is any node matching these requirements, such as the assignment to x_1 in line 8 of Listing 4.8, then it will always override the value of the definitions preceding the fork node. Additionally, due to the iur protocol this writing node will always be scheduled before any node reading this variable and thus is the more dominant write and the preceding definitions can be removed.
To summarize the process, the merge expressions are reduced, based on the position in the program, to contain the following references.

- Concurrent definitions
- Immediate dominant definition either sequential or instantaneously concurrent
- Non-dominant sequentially preceding definitions, succeeding the sequentially immediate dominant definition

Listing 4.9 shows an extended version of the program from Listing 4.7. It introduces additional read accesses of x in lines 10, 16, 18, and 23. The program is in SSA form to illustrate the context specific reduction of the expressions. Note that the expressions are the direct results of the construction scheme, but simplified to a more compact form, eliminating ineffective merge functions to improve the readability. The general procedure for this compaction is described

in the following section about normalization. Figure 4.5 presents the SCG corresponding to the original program without SSA. It shows write-read dependencies in green and additional manually added cyan arrows to indicate the effectively sequential reaching definitions. The SCG illustrates that for example the read access on x in line 10 is only reached by the definitions of x_0 and x_1. Note that for the expression in line 18 the dominant write to x_1 preceding to the fork can be ignored because x_5 is an instantaneously concurrent immediate dominant write.

Normalization

The patterns in Table 4.2 used for building the expressions create functions based on structural program constructs, regardless of the actual usage of the variable in the specific structure pattern. In addition to that, the reduction of the expressions only removes the variable references, leaving the surrounding functions in the expression. Thus, merge expressions may contain many superfluous merge functions. To eliminate superfluous functions and ensure that each function is correctly parameterized according to their definition, expressions need to be *normalized*.
The normalization includes the following transformation steps:

1. Remove functions without any arguments.
2. Replace functions with only one argument by their argument.
3. Convert functions with more than two arguments into nested functions with only two arguments.
4. Fix nesting of functions such that first arguments are nested first.

Steps 1 and 2 remove the superfluous merge functions and step 3 ensures that the functions comply with their definitions signature. For example the fork-par-join pattern can produce *conc*-function with more than two arguments, if more than two threads are defined. Merge functions with more than two arguments can be inductively nested using the definition $r(x_i, x_j, x_k) = r(r(x_i, x_j), x_k)$ where r is either *seq*, *conc* or *combine* for a fixed f. The structural composition and the previous steps can produce nested functions which do not comply with this definition, for example if a *seq*-function is nested in the second and not the first argument of another *seq*-function. Hence, step 4 reorders the the nesting of functions, such that first arguments are nested first.

$$r(x_i, r(x_j, x_k)) \rightarrow r(r(x_i, x_j), x_k)$$

The normalization of the unreduced expression in the program in Listing 4.7 on page 37 has the following effect:

$$seq(conc(seq(x_0),\quad seq(\quad x_1, seq(conc(seq(seq(seq(x_2), seq(x_3)), seq(x_4)), seq(x_5)\ ), seq(x_6)))))$$

$$\downarrow \textit{normalization}$$

$$conc(\quad x_0\ , seq(seq(x_1,\quad conc(seq(seq(\quad x_2\ ,\quad x_3\ ),\quad x_4\ ),\quad x_5\ )),\quad x_6\ ))$$

```
1  module ReducedExpressions-SSA
2  input bool i;
3  int x0, x1, x2, x3, x4, x5, x6;
4  int y0, y1, y2, y3, y4;
5  {
6   fork
7     x0 = 0;
8   par
9     x1 = 1;
10    y0 = conc(x0, x1)
11    fork
12      if i then
13        x2 = 2
14      else
15        x3 = 3;
16        y1 = conc(x0, conc(x3, x5))
17      end;
18      y2 = conc(x0, conc(seq(x2, x3), x5));
19      x4 = 4
20    par
21      x5 = 5
22    join;
23    y3 = conc(x0, conc(x4, x5));
24    x6 = 6
25  join;
26  y4 = conc(x0, x6)
27 }
```

Listing 4.9. The `ReducedExpressions` program in SSA form

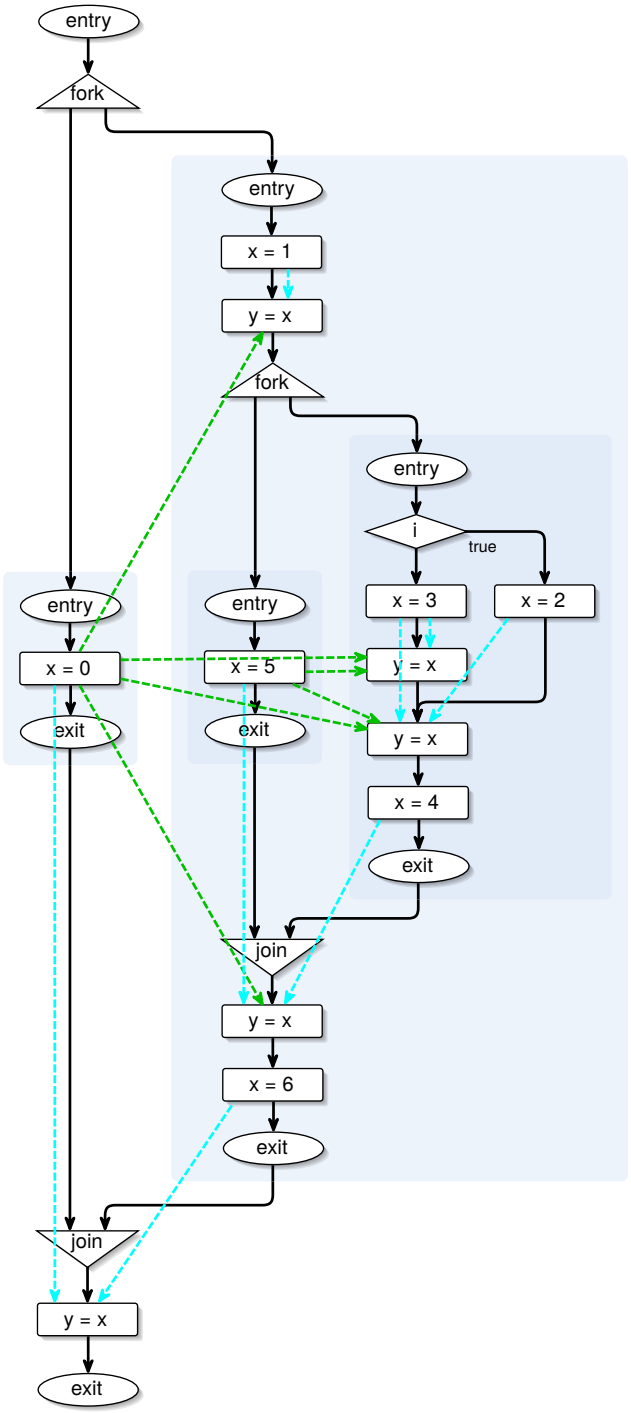


Figure 4.5. SCG representation of the `ReducedExpressions` without SSA

```
1  module NonConflictingWrites-SSA
2  int x0, x1, y, z
3  {
4    fork
5      x0 = 1;
6      y = conc(x0, x1)
7    par
8      pause;
9      x1 = 0;
10     z = conc(x0, x1)
11   join
12 }
```

Listing 4.10. The `NonConflictingWrites` program in SSA form

```
1  module PauseProblem-SSA
2  input int i
3  int x0, x1, y;
4  {
5    x0 = 0
6    if i then
7      x1 = 1;
8      pause
9    end;
10   y = seq(x0, x1)
11 }
```

Listing 4.11. The `PauseProblem` program in incomplete SSA form

The normalization eliminates superfluous functions, but it also introduces new functions to comply with the two argument definition. To to create a more reader-friendly form, steps 3 and 4 of the normalization can be omitted and instead all nested functions of the same type are joined into functions with more than two arguments. The resulting *compact* form uses a minimal number of merge functions but requires implicit nesting semantics. The unreduced expression from Listing 4.7 has the following compact form:

$$conc(x_0, seq(x_1, conc(seq(x_2, x_3, x_4), x_5), x_6))$$

The reduced compact expression is illustrated in line 26 of Listing 4.9.

4.2.4 Pauses

Extending the capabilities of the SC-specific SSA form to delayed programs is more complex than in the regular SSA from. Variables are persisted across ticks, but the augmented value notation used for the merge functions uses signals with reset semantics.
Resetting the signal value to absent at the start of every tick ensures that conflicts are correctly detected by the *conc*-functions. Listing 4.10 shows the `NonConflictingWrites` program, where x_0 and x_1 form a potential write-write conflict, but are not actually conflicting, because they are never executed in the same tick due to the pause statement in line 8.
However, using a signal with reset behavior raises the problem that the augmented value can only be correctly evaluated in same tick the assignment is executed. Listing 4.11 illustrates this problem with a small delayed program. If `i` is true, then x_1 should override x_0, but the assignment of `y` in line 10 where `x` is read is executed in the next tick. Hence, the signals were reset to absent and neither x_0 nor x_1 indicate that their value should be assigned to `y`. If `i` is false, the *seq*-function can resolve the correct value, because write and read occur in the same tick.
To solve this problem the augmented values must always be merged in the same tick their assignments are executed, regardless of any read access. Moreover, the value must correctly

influence the merge expression of later ticks. To provide a variable which represents the merged overall value of a variable in the previous tick, a variable v_{reg} is introduced for any variable v. Then, all merge expressions e for a variable v are transformed into $seq(pre(v_{reg}), e)$, where the *pre*-function indicates that the value from the previous tick is read and not any value assigned in this tick. Thus, the *pre*-function is equivalent to the pre in Esterel. The transformation ensures that the previous value is read with least sequential priority and thus is only used if no assignment is active in the current tick. Furthermore, the register variables also use an augmented value. On the one hand, this ensures that the register value is not read in the first tick, because the signal will be absent. On the other hand, this again raises the problem that the value is only valid in the same tick it is written. However, simply writing the register variable in every tick solves this problem and supersedes any analysis to detect whether the register must be written or not. It might be sufficient to write the register variables before each pause. However, for the translation into Esterel, multiple writes are problematic, and multiple pause statements can be reached the fact that in concurrent threads. Therefore, it is better to write the register variables in a single assignment. These assignments are placed in a delayed loop located in a concurrent thread surrounding the complete program. Consequently, merge expressions are used which are not reduced and include all variable versions to resolve an overall value. The loop is continued as long as the actual program does not terminate. Due to the write-before-read regime and the use of *pre*-functions, the write is performed at the end of every tick, storing the final value.
Listing 4.12 illustrates the transformation pattern applied to the `PauseProblem` program from Listing 4.11. The original program is moved into a concurrent thread, starting at line 7. Its termination is detected by the `term` variable to stop the pause handling loop. Note that the `term` flag is intended to be directly translated into a single Esterel signal and thus excluded from the SSA form. The body of the pause handling loop, starting at line 16, merges the value of `x` and writes it into the register variable. The read accesses on $\texttt{x}_{\texttt{reg}}$, in lines 12 and 17, use the *pre*-function to refer to the value of the previous tick.

Reduction

Inserting *pre*-functions into merge expressions to read the value from the previous tick allows further reductions of the definitions affecting the read value. All assignments which cannot be executed in the same tick the read access is performed can be removed form the expressions. This also affects the register variable itself. If a reading node has no preceding pause node i. e., no surface or depth nodes in the SCG, then it is always executed in the first instant. Consequently, the register variable can be removed form the expression. Furthermore, since the *pre*-functions represent the value of the previous tick, the pause itself can be considered a write to that register variable. Therefore, a pause node can be considered the immediate dominating writer to a reading node and consequently override all preceding definitions because they can only be executed in the prior to the pause. The corresponding variable versions of those preceding definitions can be removed form the merge expressions.
This analysis only detects sequential preceding pauses and not concurrent ones, but a more

```
1  module PauseProblem-SSA
2  int x0, x1, xreg, y;
3  bool term = false;
4  {
5    fork
6      // Original program
7      x0 = 0;
8      if i then
9        x1 = 1;
10       pause
11     end;
12     y = seq(pre(xreg), x0, x1);
13     // Termination
14     term = true
15   par
16     PauseLoop:
17     xreg = seq(pre(xreg), x0, x1);
18     if !term then
19       pause;
20       goto PauseLoop
21     end
22   join
23 }
```

Listing 4.12. The `PauseProblem` program in correct SSA form

```
1  module PauseReducedExpressions-SSA
2  input bool i;
3  int x0, x1, xreg, y, z;
4  bool term = false;
5  {
6    fork
7      // Original Program
8      fork
9        x0 = 0;
10       pause;
11       y = seq(pre(xreg), x1)
12     par
13       x1 = 1;
14       if i then
15         pause;
16       end;
17       z = seq(pre(xreg), conc(x0, x1))
18     join;
19     // Termination
20     term = true
21   par
22     PauseLoop:
23     xreg = seq(pre(xreg), conc(x0, x1));
24     if !term then
25       pause;
26       goto PauseLoop
27     end
28   join
29 }
```

Listing 4.13. The `PauseReducedExpressions` program in SSA form

advanced analysis might be able to eliminate more definitions, for example by analyzing surface and depth relation between threads. However, further analysis approaches are not part of this thesis.

Listing 4.13 shows the `PauseReducedExpressions` program to illustrates the reduction of merge expressions by finding dominant pauses. The program performs two concurrent assignment to `x`, one in line 9 and the other in line 13, both with different values. After the assignment to x_0, there is a pause statement in line 10. Hence, the subsequent read does not need to read x_0. In contrast to that, x_1 is followed by a conditional pause in line 15 depending on `i`. Consequently, the subsequent read in line 17 needs to check both x_1 and x_0, and will detect the conflict if `i` is true. The concurrent loop starting at line 22 handles the preservation of the variable value across ticks borders.

Inputs

Another aspect when handling pause statements is the input output interface of the program. Variables marked as input are set by the environment every tick and thus the pause acts a definition to each of the input variables, which must be correctly considered in the SSA form. The regular SSA form, presented in Section 4.2.1, has the same problem and it was solved by introducing explicit assignments after each pause. The SC-specific SSA form does not need these additional assignments because each merge expression in a delayed program already considers the pause statement as an assignment due to the *pre*-functions. Since input variables are maintained by the environment they do not need a register variable. For an input variable i the $seq(pre(i), e)$ expressions is replaced by $seq(i, e)$. This solution also correctly handles the case that an input variable is locally assigned by the program, because the inner merge expressions e consisting of local input variable versions is ordered sequentially after the environment input value and thus overrides it.

Another important aspect regarding the interface is that splitting up and renaming variables changes the interface of the program. It is necessary to transform input output variables to comply with the SSA form. However, the resulting program is no longer equivalent to the source program because it cannot communicate with the environment via the originally defined interface. Section 4.2.7 discusses this problem in more detail and presents a solution to retain semantical equivalence.

4.2.5 Loops

Allowing jumps in programs gives the programmer great freedom to create complex programs structures, but at the same increases the demands on analyzing these structures. Jumps can form cycles, but not every cycle is a loop structure compatible with Esterel, further discussed in Section 4.3.1. To analyze loops for this SSA form, the SCGs representing the programs are restricted to those which are *reducible flowgraphs* [HU72]. In a reducible flowgraph the nodes can be collapsed into their ancestors, resulting in a graph with a single node. This especially includes cyclic node structures where the entire loop body can be reduced to a single node. As a result, reducible flowgraphs provide the property that each loop has a single loop entry node, which can be determined using dominator relations [HU74]. This concept allows to detect loops which are compatible to the Esterel loops structure.

The construction scheme for merge expression presented so far does not take jumps into account. Performing the procedure ignoring any backward jumps results in expressions which consider only a single iterations of a loops body. This is sufficient for expressing the structural relation of the reaching definitions. However, the main problem is that executing the loops will cause the actual controlflow to jump back to a position which it already passed. In combination with the fixed sequential order encoded in the merge expressions this leads to incorrectly determined values, if multiple assignment to the same variables are located in a loop.

```
1  module InstantaneousLoop
2  input int i, j;
3  int x, y;
4  {
5   x = 0;
6   Loop:
7    if i then
8     x = 1
9    end;
10   y = x;
11   if j then
12    x = 0
13   end;
14  goto Loop
15 }
```

(a) Original

```
1  module InstantaneousLoop-SSA
2  input int i, j;
3  int x0, x1, x2, y;
4  {
5   x0 = 0;
6   Loop:
7    if i then
8     x1 = 1
9    end;
10   y = seq(x0, x1, x2);
11   if j then
12    x2 = 0
13   end;
14  goto Loop
15 }
```

(b) Incorrect SSA form

Listing 4.14. The `InstantaneousLoop` program

Listing 4.14a presents the `InstantaneousLoop` program which performs such multiple writes in a loop starting at line 6. Inside the loop body, `x` is assigned to 1 and afterwards to 0 depending on `i` and `j`. In each iteration `y` reads the value of `x` in line 10. The current state of the SSA transformation cannot create an expression resolving the correct value. Listing 4.14b shows the result of the SSA transformation. The merge expression in line 10 considers all versions of `x` and sequentially orders them based on the textual ordering in the source code. This solution is incorrect for the case where `i` and `j` are true and both assignment are executed in each iteration. On the one hand, the textual order is inverted due to the backward jump. First x_2 overrides x_1, then the backward jump is performed and x_1 overrides x_2. On the other hand, due to the loop the definition of x_2 reaches `y` and is consequently included in the merge function. However, reading the value of a variable which is assigned in the same tick but sequentially after the read, violates the constructiveness in the sense of Esterel.
The regular SSA transformation handles cycles by introducing ϕ-functions at the entry node of the loop, to merge definitions prior to the loop with the definitions of the iterations. However, in a concurrent context this solution is insufficient because a variable in the loop might be concurrently read and thus requires a merge expression including the sequential relation of all assignments in the loop body.

Handling Restricted Loop Structures

The restriction to reducible flowgraph facilitates the detection of loops, but to solve the problem of sequentiality inversion the supported loop structures must be further restricted. In the SC MoC instantaneous loops are allowed, whereas in Esterel a loop cannot perform multiple instantaneous iterations. The controlflow must not instantaneously reach the end

of the loop body when the body is entered at its head. Hence, requiring non-instantaneous loops is a reasonable restriction. However, requiring delayed loop body is not sufficient for the following approach to solve the problem of the sequential ordering. Loops structures are further restricted to those which have at least one non-concurrent pause statement in the loop body, which is executed in every iteration. Note that this restriction is only necessary to solve the ordering problem of multiple assignments to the same variables located in a loop. If no such statements exist the normally generated merge expressions are sufficient.

Based on this restriction, the sequential ordering of multiple writes can be fixed by reordering the definitions in the merge expression according to the separation into different ticks. At first, the first pause statement which dominates the end of the loops body must be detected. The are delayed loops that do not contain such a pause statement which consequently limits this approach, as presented at the end of this section. Moreover, the dominant pause must be non-concurrent. A concurrent dominant pause could allow a reordering but may cause the loop to contain schizophrenic statements. These are statements which are executed multiple times in the same tick. However, handling schizophrenia is a complex task [TS04] and not targeted by this thesis. Based on the fist non-concurrent pause statement which is executed in each iteration, all nodes in the loop body are separated into two distinct sets. The *surface* contains all nodes preceding the dominant pause and ignoring the loopback. All other nodes are assigned to the *depth*. This separation allows to reorder definition of nodes in the surface sequentially after the depth, in all merge expressions regarding the loop body. This results in a new pattern for the expression construction. The rule applies only to labels indicating the entry of a loop.

$$\texttt{LoopEntryLabel:} \rightarrow \mathit{seq}(\mathit{loop\text{-}body\text{-}depth}, \mathit{loop\text{-}body\text{-}surface}, \mathit{loop\text{-}end})$$

The reordering can be made because the surface is always separated from the depth by the domination pause. When the loop is entered, the surface is ordered sequentially after the depth, but no assignment of the depth can be, by definition, executed in this tick. Hence, no value of the depth is spuriously overridden in this situation. Due to the register introduced for each variable to handle the delay, the value is correctly stored and can be read or overridden in future ticks. The interesting part is, when assignments from both surface and depth are active in the same tick. This situation occurs when the backward jump is performed. The reordered merge expressions handles this situation correctly, because the surface can override the depth. Additionally, the definition of the surface assures that no assignment of the depth will be executed after the surface, if surface and depth are active in the same tick. The reason is the dominant pause which is always executed before the depth is entered.

Listing 4.15a presents an extended delayed variant of the `InstantaneousLoop` program in Listing 4.14a. A pause statement is inserted after the assignment to `y` in line 12 and an additional assignment to `z` is added in line 17, reading `x`. Listing 4.15b illustrates the transformed program in SSA form with correct handling of the loop. Note that the presented SSA form could use a loop pattern for the register assignments, without termination detection because the original program cannot terminate.

```
1  module DelayedLoop
2  input int i, j;
3  int x, y, z;
4  {
5   x = 0;
6   Loop:
7    if i then
8     x = 1
9    end;
10   y = x;
11   pause;
12   if j then
13    x = 0
14   end;
15   z = x;
16  goto Loop
17 }
```

(a) Original

```
1  module DelayedLoop-SSA
2  input int i, j;
3  int x0, x1, x2, xreg, y, z;
4  bool term = false;
5  {
6   fork
7    x0 = 0;
8    Loop:
9     if i then
10     x1 = 1
11    end;
12    y = seq(pre(xreg), x0, x2, x1);
13    pause;
14    if j then
15     x2 = 0
16    end;
17    z = seq(pre(xreg), x2);
18   goto Loop;
19   term = true
20  par
21   PauseLoop:
22   xreg = seq(pre(xreg), x0, x2, x1);
23   if !term then
24    pause;
25    goto PauseLoop
26   end
27  join
28 }
```

(b) SSA form

Listing 4.15. The `DelayedLoop` program

Compared to the original ordering in Listing 4.14b, the merge expression for the variable y changes in the following way.

$$\mathsf{y} = seq(pre(\mathsf{x}_{reg}), \mathsf{x}_0, \mathsf{x}_1, \mathsf{x}_2)$$
$$\downarrow$$
$$\mathsf{y} = seq(pre(\mathsf{x}_{reg}), \mathsf{x}_0, \mathsf{x}_2, \mathsf{x}_1)$$

Note that the reordering of the referenced variable versions is performed before the pause transformation. This assures that the loop introduced for the register variable is not affected by the loop handling, and the *pre*-function is not considered in the reordering. Additionally, the expressions inside a loop body must be reduced with respect to pause statements. Otherwise merge expression located in the depth before the backward jump may check the signal of augmented values assigned in the sequentially preceding surface. This is illustrated by the merge expression for the assignment of z in line 17 of Listing 4.15b. The reduction

```
module NestedLoops-SSA
int x0, x1, x2, x3, x4;
bool term = false;
{
  fork
    x0 = 0
    Loop1:
      if seq(pre(xreg), x0, x4) then
        x1 = 1;
        Loop2:
          x2 = 0;
          pause;
          x3 = 1;
        goto Loop2
      end;
      pause;
      x4 = 0;
    goto Loop1
    term = true
  par
    PauseLoop:
    xreg = seq(pre(xreg), x0, x4, x1, x3, x2);
    if !term then
      pause;
      goto PauseLoop
    end
  join
}
```

Listing 4.16. The `NestedLoops` program in SSA form

```
module RejectedDelayedLoop
intput int i;
int x, y;
{
  Loop:
    if i then
      pause
    end;
    x = 0;
    if !i then
      pause
    end;
    x = 1;
  goto Loop
}
```

Listing 4.17. The `RejectedDelayedLoop` program

is necessary, since reading x_1 would be considered non-constructive in Esterel, because even if x_1 is sequentially preceding the reading statement, it is executed after it. Hence, whether x_1 is executed or not cannot be constructively determined in the assignments to `z`. To prevent programs from being unnecessarily rejected, the dominant pause statement must be considered to remove ineffective preceding references. Since the depth is always separated by the dominant pause, the reduction of the expressions is always possible.

In programs where multiple loops are nested in each other, the reordering is performed in an inside out manner. Consequently, the loops may have different dominant pause statements. However, the surface will always be ordered sequentially after its depths. The `NestedLoops` program presented in Listing 4.16 illustrates the handling of two nested loops. The `Loop2` structure in nested inside `Loop1` and both loops have different dominant pauses.

At first, only the merge expressions for `Loop2` is reordered.

$$seq(x_2, x_3) \downarrow seq(x_3, x_2)$$

Afterwards, the surface depth analysis is performed for the `Loop1` structure and the corresponding part in the expression is reordered.

$$seq(x_1, x_3, x_2, x_4) \downarrow seq(x_4, x_1, x_3, x_2)$$

Limitation

The presented solution uses a more restricted requirement for delayed loop structures than Esterel. It requires a dominant pause statement which is executed in every iteration. This is necessary to ensure clear separation of surface and depth. However, programs with valid delayed loops, in the sense of Esterel, are rejected if they do not contain such a pause statement. Listing 4.17 illustrates a rejected program, containing a valid delayed loop. The Esterel constructiveness analysis detects that the loop body is delayed, because one of the pauses is always executed in every tick. Yet there is no pause statement which is executed in every iteration. In this example, both assignments are considered instantaneously reachable and thus assigned to the surface. This again raises the sequentiality inversion problem because depending on `i`, the sequential overriding of the two assignments changes.

4.2.6 Updates

In a strictly sequential program, updates are easy to handle. They just read the value of the preceding variable, apply their update, and write the new value into their variable version. This is how the regular SSA form transforms relative writes, presented in Section 4.2.1. However, if updates are executed in a concurrent context, the iur protocol defines additional scheduling constrains. Hence, the merge expressions have to consider these additional constraints which influence the variable value. The *combine*-function is defined to handle the application updates in merge expressions. Creating a merge expressions with *combine*-functions uses a different procedure than the solutions for absolute writes.

1. Transform all updates into *combine*-form
2. Create the merge expressions based on a scheduling analysis
3. Transform loops and pauses

```
1  module UpdateTransformation
2  input bool i, j, k;
3  int x, y;
4  {
5    fork
6      x = 0;
7      if i them
8        x = 1
9      end
10   par
11     if j then
12       x = x + 1
13     end
14   par
15     if k then
16       x = x + 1
17     end
18   par
19     y = x
20   join
21 }
```

(a) Original

```
1  module UpdateTransformation-SSA
2  input bool i, j, k;
3  int x0, x1, x2, x3, y;
4  {
5    fork
6      x0 = 0;
7      if i them
8        x1 = 1
9      end
10   par
11     if j then
12       x2 = conc(seq(x0, x1), x3) + 1
13     end
14   par
15     if k then
16       x3 = conc(seq(x0, x1), x2) + 1
17     end
18   par
19     y = conc(seq(x0, x1), x2, x3)
20   join
21 }
```

(b) Incorrect SSA form using only *seq* and *conc*-functions

Listing 4.18. The `UpdateTransformation` program

Update Transformation

The SC MoC considers a relative write in the form $x = f(x, e)$ an update, where x is a variable, e an expressions independent from x, and f a valid combination function.
Without a *combine*-function, relative writes can be translated such that they are considered a normal read on the variable and a subsequent write. Listing 4.18a shows a program which performs two initialization, two updates and a read in four concurrent threads. Except the first initialization, all write accesses depend on an input signal. Thus, the possible results for `y` can be in the range from 0 to 3, depending on these inputs. Listing 4.18b illustrates the result of a notional SSA transformation using only *seq*- and *conc*-functions to express updates. There are two problems illustrated by this example. At first, reading the variable value in the update statements requires a merge expressions in lines 12 and 16. These expressions consider all reaching definitions including other updates. However, this leads to a program considered not constructive in Esterel because the updates form a causality cycle. Moreover, handling updates like normal reads introduces additional merge expressions. The second problems is that the *conc*-function is not capable of correctly considering the scheduling order between initializations and updates. The result of the update stands in conflict with the initializations, but the iur protocol introduces clear ordering rules for the different types of concurrent writes. Consequently, these rules should be encoded into the merge expressions.

```
module UpdateTransformation-SSA
input bool i, j, k;
int x0, x1, x2up, x3up, y;
{
 fork
  x0 = 0;
  if i them
   x1 = 1
  end
 par
  if j then
   x2up = 1
  end
 par
  if k then
   x3up = 1
  end
 par
  y = combine(+, seq(x0, x1), x2up, x3up)
 join
}
```

Listing 4.19. The `UpdateTransformation` program in SSA form using *combine*-functions

To reduce the number of merge expressions, to prevent cycles due to reading variables in the update statements, and to encode the update ordering in the merge expression, the updates are transformed into *combine*-function form. Each update statement $x = f(x, e)$ is transformed into $x_{up} = e$ using an augmented value. Hence, each of the update modification values e is stored in a variable and applied to the actual value when it is read. In the merge expressions reading the variable x, including the update x_{up}, the *combine*-functions is added to apply the modification value, if the signal indicates that the update was executed. The expression $combine(f, e_{read}, x_{up})$ represents the merging of the update x_{up}, which is applied to the incoming value resolved by the merge expression e_{read}. Thus, the update result is not stored in the variable written by the actual update statement but composed when the overall variable value is read. Listing 4.19 illustrates the *combine* transformation on the previous program. x_2 and x_3 become x_{2up} and x_{3up} and no longer require additional merge functions. The merge function in the assignment to y handles the application of the actual update modification values. Note that the transformation into *combine*-functions does not solve the problem of the correct ordering of writes. The merge expressions in the presented example simply orders both updates after the initializations in the textual definition order. An equivalent expression is $combine(seq(\mathsf{x}_0, \mathsf{x}_1), \mathsf{x}_{3up}, \mathsf{x}_{2up})$. For the given program both expressions are correct. However, in general it is not sufficient to generate merge expressions only based on the textual order but also requires an analysis of the scheduling constrains.

Algorithm 1 Construction of merge expressions based on static schedules

```
1: procedure EXPRESSION([S:s])
2:     if S is empty then
3:         return s
4:     else if s is update of type f then
5:         return combine(f, EXPRESSION(S), s)
6:     else
7:         return seq(EXPRESSION(S), s)
8:     end if
9: end procedure
```

Scheduling

In general the SC MoC is based on an iur restricted free scheduling. Thus, the ordering of statements is dynamically decided by an SC conformant scheduler. However, the merge expressions have to encode the scheduling constraints and ordering rules, and they are statically generated at compile time. The reason for the encoding into static expression is the simple fact that the SC-specific SSA form is designed to generate programs which are independent from an SC compiler and can be compiled with Esterel, which only requires a constructive write-before-read regime. Consequently, in the presence of concurrent updates only those programs can be handled by the SC-specific SSA transformation which provide some static schedule for the variable definitions.

Static Scheduling

For SC programs which are structurally ASC, it is possible to determine a static SC schedule. This schedule defines a fixed scheduling order for each node in the SCG. Based on such a static SC schedule the merge expression are constructed using the recursive construction procedure presented in Algorithm 1. The procedure receives a list of scheduled variable versions influencing the read access of that value. In the algorithm, S is the list of statements ordered by the static schedule, except its last element s. Note that due to the fixed scheduling the resulting merge expressions only use *combine*- and *seq*-functions. Moreover, since the variable read by the *combine*-functions is nested inside the function, the pattern for the consideration of input and register variables, described in Section 4.2.4 does not work correctly. For merge expressions without *combine*-functions the expression is surrounded by a *seq*-function considering the input or register with the lower sequential order. In case of merge expressions for variables written by updates, this would prevent the *combine*-functions from reading and updating the input or register value. Hence, if a merge expressions must consider an input or register variables, it must be added to the end of the schedule. This way, the algorithm will place the variable in a position with the least sequential priority but available to be read by the *combine*-functions.

```
1  module UpdateOrder
2  input bool i, j, k;
3  int x, y;
4  {
5   x = 0;
6   fork
7     if i then
8       x = x + 1
9     end;
10    if j then
11      x = 5
12    end;
13    y = x
14  par
15    if k then
16      x = x + 1
17    end
18  join
19 }
```

(a) Original

```
1  module UpdateOrder-SSA
2  input bool i, j, k;
3  int x0, x1up, x2, x3up, y;
4  {
5   x0 = 0;
6   fork
7     if i then
8       x1up = 1
9     end;
10    if j then
11      x2 = 5
12    end;
13    y = combine(+, seq(combine(+, x0, x1up), x2), x3up)
14  par
15    if k then
16      x3up = 1
17    end
18  join
19 }
```

(b) SSA form with scheduled updates

Listing 4.20. The `UpdateOrder` program

For the program in Listing 4.19 the order of x_0 and x_1 is explicitly defined by the sequential ordering in the program. According to the initialize-before-update dependencies x_{3up} and x_{2up} are ordered after the initializations. Based on the definition of combination functions, the order of x_{3up} and x_{2up} does not affect the final value of the overall variable, thus it is up to the scheduler do decide any ordering between the two updates. The `UpdateTransformation` program in Listing 4.19 is a very simple scheduling problem, where all updates are scheduled after the initializations. In general, the ordering of updates after initializations does not always hold, as the following example illustrates.

Listing 4.20a presents the more challenging program `UpdateOrder`, illustrating the requirement for a full scheduling analysis rather than simple structural patterns such as those used for programs without updates. The program initializes `x` with 0 and forks into two threads. The first thread performs an increment by 1 depending on `i` and then initializes `x` with 5 depending on `j`. The second thread performs another increment of `x` by 1 depending on `k`. Table 4.3 shows the expected output behavior of the program for all possible inputs. The iur regime only extends the sequential ordering by additional dependencies, and thus the initialization of x_2 in line 11 is expected to sequentially override the update result of x_{1up}. Since no cycles including concurrent dependencies exist, the program is considered ASC.

i	j	k	y
false	false	false	0
false	false	true	1
false	true	false	5
false	true	true	6
true	false	false	1
true	false	true	2
true	true	false	5
true	true	true	6

Table 4.3. Expected results for y in the `UpdateOrder` program depending on the input values

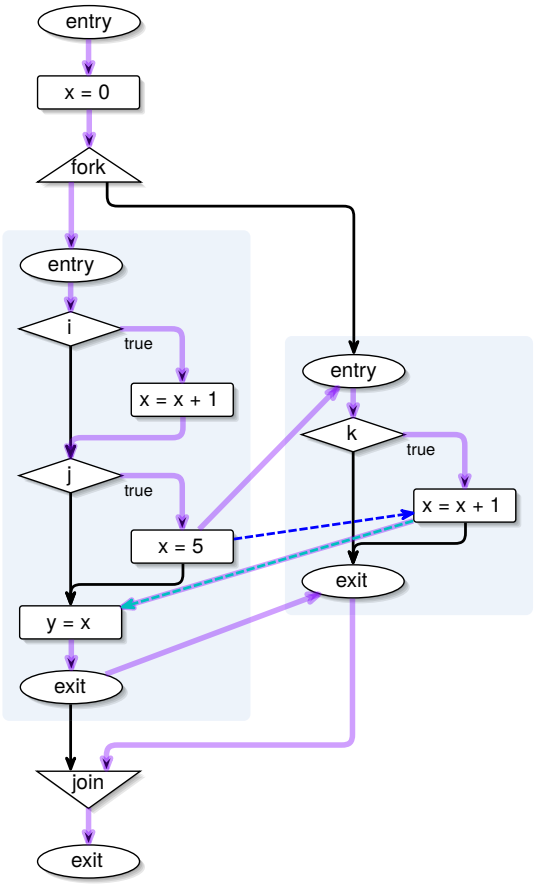


Figure 4.6. SCG representation of the `UpdateOrder` program with dependencies and static scheduling path

Regarding the two threads the following sequential and concurrent dependency relations are determined.

$$x_2 \rightarrow_{iu} x_{3up}$$
$$x_{3up} \rightarrow_{ur} y$$
$$x_0 \rightarrow_{seq} x_{1up} \rightarrow_{seq} x_2 \rightarrow_{seq} y$$
$$x_0 \rightarrow_{seq} x_{3up}$$
$$\Downarrow$$
$$x_0 \rightarrow x_{1up} \rightarrow x_2 \rightarrow x_{3up} \rightarrow y$$

```
1  module NotASC
2  input bool i;
3  int x, y;
4  {
5    fork
6      if i then
7        y = 0
8      end;
9      x = 0
10   par
11     if !i then
12       y = 1
13     end;
14     x = x + 1
15   join
16 }
```

Listing 4.21. The NotASC program

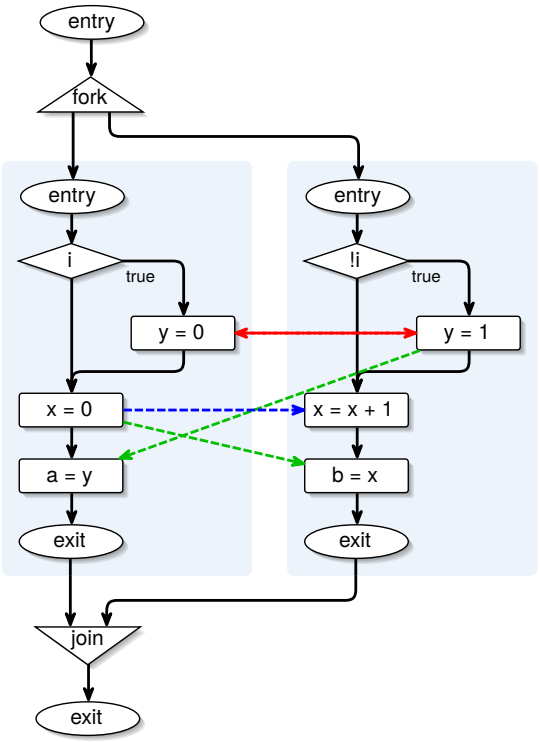


Figure 4.7. SCG representation of the NotASC program with dependencies

As illustrated it is possible to deduce an acyclic partial order for all statements which complies with the dependencies. This ordering is also depicted by the scheduling path in the corresponding SCG in Figure 4.6. For the given schedule the algorithm produces the following merge expression for reading x in the assignment to y:

$$y = combine(+, seq(combine(+, x_0, x_{1up}), x_2), x_{3up})$$

Listing 4.20b shows the UpdateOrder translated into SSA form using this merge expression.

However, requiring a static schedule to handle updates would unnecessarily restrict the number of programs which can be translated, disregarding their constructiveness in Esterel. All programs which contain a static cyclic dependency and updates would be rejected. A classic example of a cyclic, yet constructive program is the Token Ring Arbiter [Pan02], described in more detail in Section 6.2.3. Moreover, every write-write dependency prevents a static schedule because such dependencies are symmetrical and form a cycle themselves. Hence, even if this situation can be handled dynamically by the *conc*-function, updates in such programs could not be transformed if a global static schedule would be required.

Listing 4.21 illustrates the program NotASC with a write-write dependency between the initializations of y. Independent from y, there is an initialization and an update on x. The

```
1  module NotSC
2  int x, y, a, b;
3  {
4     fork
5        y = y + 1;
6        x = 0
7     par
8        x = x + 1;
9        y = 0
10    par
11       a = x;
12       b = y
13    join
14 }
```

(a) Original

```
1  module NotSC-SSA
2  int x0, x1up, y0up, y1, a, b;
3  {
4     fork
5        y0up = 1;
6        x0 = 0
7     par
8        x1up = 1;
9        y1 = 0
10    par
11       a = combine(+, x0, x1up);
12       b = combine(+, y1, y0up)
13    join
14 }
```

(b) SSA form with scheduled updates based on partial scheduling

Listing 4.22. The `NotSC` program

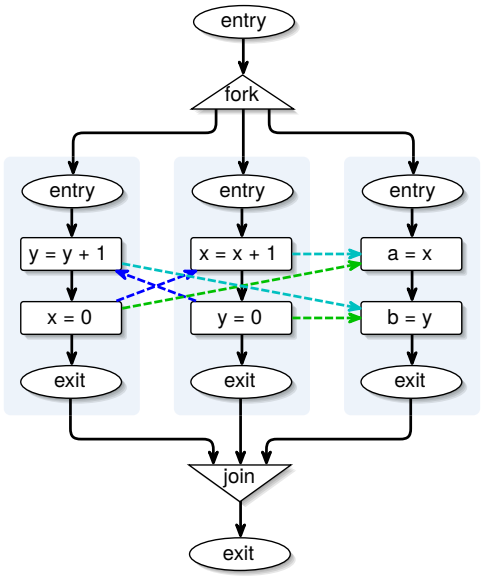


Figure 4.8. SCG representation of the `NotSC` program with dependencies

two initializations are not conflicting due to the surrounding mutual exclusive conditions. Figure 4.7 depicts the corresponding SCG with dependencies. For the two reading assignments to `a` and `b`, there are obvious merge expressions, $\mathtt{a} = conc(\mathtt{y}_0, \mathtt{y}_1)$ and $\mathtt{b} = combine(+, \mathtt{x}_0, \mathtt{x}_{1up})$. However, this is not possible to generate using the approach presented so far.

To solve situations where the scheduling of updates is hindered by cycles without any data relation to the actual variable, the scheduling is simplified. To produce a static schedule for a specific variable, only those parts of the program are included in the scheduling process which access the variable. In this example, the schedule is created only for the two concurrent assignments of `x` and the correct merge expression can be generated. For the creation of the merge expression for `y` the usual patterns can be used because no update on `y` is present. This will result in the expected usage of the *conc*-function.

One important aspect when reducing the program to create a partial schedule is that dependencies of other statements in the program can no longer influence the ordering of the statements of the analyzed variable. Nevertheless, such dependencies cannot affect the order of initializations, because it is either explicitly defined by the sequential order or form a concurrent write-write dependency. Also, the order of updates is not influenced because by definition of the allowed combination functions in the SC MoC updates are always confluent. However, surrounding statements may require to schedule an update before the initialization. Such a program would be reject by the SC MoC because the iur protocol would be violated. Listing 4.22a shows the program `NotSC` which is not SC. The updates and initializations on `x` and `y` form a cycle including the initialize-before-update dependencies. None of the updates can be executed, because both require the initialization to be scheduled beforehand, but

the initializations are ordered sequentially after the other update. Figure 4.7 depicts the corresponding SCG with dependencies. However, the partial scheduling yields a result for both variables, resulting in merge expressions for `a` and `b`. Hence, even if the program is not SC, it can be translated into SSA form, presented in Listing 4.22b. Moreover, the program is considered constructive in the sense of Esterel because each variable in the SSA form is written before read. Thus, the program is accepted as SSC without being SC. Nevertheless, this is reasonable regarding the concept of SC+. SC+ is an extension of SC, allowing out of order execution of statements without data dependencies. It weakens the role of sequential ordering and only considers sequential data dependencies and iur dependencies when it comes to scheduling. This is akin to out-of-order execution on modern processors and focuses more on data-dependencies than sequential controlflow [Wei15]. The program `NotSC` is an example for a valid SC+ program because `x` and `y` do not share any relation and thus the sequential ordering between the initialization and updates can be ignored. Reducing the program when creating a schedule to only those statements accessing a variable results in the same dissolution of unrelated sequential orderings. Especially the motivation of constructive circuit representations justifies this solution, because independent calculations in a circuit are always parallel within one instance.
The process of creating merge expressions including *combine*-function replaces the procedure described in Section 4.2.3 for all variables which are written using updates. Hence, pause statements and loops are also handled subsequently.

Limitations of Static Scheduling

Using static schedules for constructing merge expressions restricts the transformable programs. Even the expansion to partial schedules still requires that the remaining statements are iur-acyclic. Thus, all programs containing assignments with write-write dependencies and updates of that variable cannot be transformed. The same holds for non-confluent updates, using incompatible combinations functions. Even if the updates are dynamically not conflicting, the ordering still requires a static schedule.
Such programs are directly rejected by the SSA transformation.

Valued Signals Approach

Another approach for handling updates is inspired by valued signals of Esterel. Valued signals support multiple emission with different values, if a combine function with commutative and associative property is defined to merge the values. The constructive reading of the value is only allowed if all reachable emits were executed.
At first glance, this regime seems similar to the handling of updates of the same type in the SC MoC. A possible translation could directly translate all updates of a variable into multiple emission of a valued signal. When the variable is read the combined update value is merged with the initialization. Listing 4.23a shows the program `ValuedSignalUpdates` with multiple

```
module ValuedSignalUpdates
int x, y;
{
  x = 0
  fork
    x = x + 1
  par
    x = x + 1
  par
    y = x
  join
}
```

(a) Original in SCL

```
module ValuedSignalUpdates:
signal x0 : interger in
signal xup : combine interger with + in
signal y : interger in

emit x0(0)
[
  emit xup(1)
||
  emit xup(1)
||
  emit y(?x0 + ?xup)
]
end end end signal
end module
```

(b) Esterel program using a single valued signal for updates

Listing 4.23. The `ValuedSignalUpdates` program

concurrent updates on the variable x. Listing 4.23b illustrates a possible SSA form directly translated into Esterel.
Apart from the fact that this solution requires valued signals in SCL to create a similar form as Esterel, this approach has one major problem. The updates are not separated from each other. Consequently, programs which require a schedule such that initializations interleave between updates are handled incorrectly by this approach. The program in Listing 4.20a is a clear example for a program that requires separation between update values. Furthermore, the accumulation of all update values in a single valued signal does not allow to read intermediate results of the update. In an SC a reading statement can be placed between two sequential updates to read the intermediate result. However, with a single valued signal such a read is considered non-constructive in Esterel because the value of the signal is accessed before the last reachable emit is executed. Hence, this approach is not capable of handling updates sufficiently.

4.2.7 Interface Compliance

The interface of an SC programs explicitly specifies the communications with the environment. The conveyed output variables at the end of a tick in relation to the input variables define the reaction of a program. Two programs are considered equivalent, if both produce the same output streams for all possible input streams.
The SC-specific SSA transformation is designed to produce equivalent programs in SSA form, which are executable. However, due to the renaming of variable into SSA versions, which also affects input and output variables, the original interface is altered. Consequently, the resulting

program is no longer equivalent to the source model because the environments cannot match the interface variables.
The effect of input variables on the merge expressions is already handled for delayed programs, presented in Section 4.2.4, because the definitions of input values by the environment influence the semantics of SSA itself. However, handling the implicit environment definition in a pause differently from local definitions when assigning another value to input variables requires the separation of the input variable and local versions.
Output variables have a similar problem, multiple assignments in a program may assign the output variable, resulting in multiple variable versions, but the environment expects one value for the output variable every tick.
To achieve this separation, variables marked as input or output are not renamed itself. All assignments resulting in new versions of the variable will be declared locally. Hence, the handling of inputs can refer to the original input value and merge the value with local definitions. Assignments to the output variable also assign local variable versions. Since inputs must be considered when creating correct merge expressions they are handled together with pauses. However, output variables require an additional transformation to provide the environment with the correct value in each tick. First, each program must convey the outputs in the tick it terminates, consequently an assignment to the output variable assigning the value based on the different variable versions is added. This is only required for instantaneous programs, because in delayed programs the pause handling introduced a concurrent loop and register variables, which can be used for that purpose. The register variable can also be used to convey the final output value in every tick. Variables defined as input and output require both transformations, including the merge expression for the register variable
Listing 4.24a shows the program `ConcurrentIO` which uses input, output, and input output variables. The first thread sets the input output variable `x` to 1, if it is equal 0. This prevents a division by zero error in the subsequent assignment to `o`, where `i` is divided by `x`. Based on the condition `x` contains either the actual input value or 1. Additionally, the value is conveyed to the environment. In the second thread the input `i` is set to 0, if it is negative. Consequently, the expression in line 10 also refers either to this new value or the original input value. In the second tick, the second threads performs an assignment to `x` multiplying `o` and `i`. In this case `o` has the value of the previous instant and `i` is has the value provided by the environment, ignoring possible assignments of the previous tick. Listing 4.24b illustrates that the result of the SSA transformations correctly considers the input and output behavior and produces an equivalent program. The value of `o` is persisted every tick in $\texttt{o}_{\texttt{reg}}$ and the final value of both `o` and `x` is merged and conveyed at end of every tick. The expressions reading the inputs in line 15 consider both the value form the environment and the local assignments. The assignment in line 21 only considers the input value of `i` because the local assignment is located before a dominant pause.

```
module ConcurrentIO
input int i;
output int o;
input output int x;
{
  fork
    if x == 0 then
      x = 1
    end;
    o = i / x
  par
    if i < 0 then
      i = 0
    end;
    pause;
    x = o * i
  join
}
```

(a) Original

```
module ConcurrentIO-SSA
input int i;
int i0;
output int o;
int o0, oreg;
input output int x;
int x0, x1;
bool term = false;
{
  fork
    fork
      if seq(x, x1) == 0 then
        x0 = 1
      end;
      o0 = seq(i, i0) / seq(x, conc(x0, x1));
    par
      if i < 0 then
        i0 = 0
      end;
      pause;
      x1 = pre(oreg) * i;
    join;
    term = true
  par
    term = false;
    PauseLoop:
    oreg = seq(pre(oreg), o0);
    o = oreg;
    x = seq(x, conc(x0, x1));
    if !term then
      pause;
      goto PauseLoop
    end
  join
}
```

(b) SSA form

Listing 4.24. The `ConcurrentIO` program

4.3 Translation into Esterel

This section describes the translation of SC programs into Esterel. At first, Section 4.3.1 discusses the challenges of translating the structural syntax form SCL or the SCG into Esterel, which are mainly caused by the `goto` statement. The section presents general structural translation patterns. Another challenging aspect is the transformation of SC variables into Esterel. Regarding the definition of the constructive semantics, Esterel uses signals as primary data type. This collides with the usage of shared variables in SC programs, even in the absence of an iur protocol compilation due to the SSA form. Section 2.1 on page 6 illustrates data types of Esterel and compares it with variables in the SC MoC. Section 4.3.2 first discusses behavioral challenges due to the different semantics in Esterel. Considering these aspects the following sections present two solutions for encoding SC variables in Esterel. The first, presented in Section 4.3.3, uses pure signals for boolean variables. The second, in Section 4.3.4, presents an encoding using valued signals which are not fully supported by the implemented constructiveness analysis of Esterel but handles non-boolean data types more easily.
Furthermore, in the absence of concurrent programs, Esterel variables provide a suitable translation for SC variables. However, since Esterel variables cannot be sufficiently used as shared variables between threads or analyzed when determining constructiveness, this section does not consider them in the presented solutions. Moreover, in the following sections, SC variables are denoted as variables disregarding Esterel variables.

4.3.1 Structure

When comparing the language definition of SCL, presented in Table 2.3 on page 10, with the Esterel kernel language, shown in Table 2.1 on page 7, the languages share similar structures. The sequential and concurrent composition of statements can be directly adapted into Esterel. The present test can be used to translate if-then-else structures, but the condition depends on the variable encoding. The translation of assignments is also based on the chosen encoding strategy. The delay statement pause can be directly adopted to Esterel. The most challenging construct is the `goto` statement.
There is no explicit jump statement in regular Esterel. However, there is a thread-safe jump statement for extending Esterel proposed by Tardieu [Tar04]. Nevertheless, jumps can be separated into forward and backward jumps. The only statement which allows similar forward jump behavior is the trap construct. However, the structure of Esterel is always hierarchical. That means, all structural statement are paired and the included statements form a nested program. When translating forward jumps with traps, the `goto` can be replaced by an `exit` t and the target label by `end trap`. Due to the hierarchical structuring, the `trap` t must be placed on the same level as the `end trap` and this is not always the case with jumps. Hence, jumps into nested structures of other statements such as conditional branches or loops cannot directly translated into Esterel. Listing 4.25 shows the `JumpIntoBranch` program which performs a jump into the then branch of conditional structure. The target label of the `goto` in line 7 is inside the then branch of the conditional in line 11. Figure 4.9 depicts the corresponding SCG.

```
1  module JumpIntoBranch
2  input bool i, j;
3  int x;
4  {
5   if i then
6     x = 1;
7     goto Branch
8   else
9     x = 2
10  end;
11  if j then
12    x = 3;
13    Branch:
14    x = 4
15  else
16    x = 5
17  end
18 }
```

Listing 4.25. The `JumpIntoBranch` program

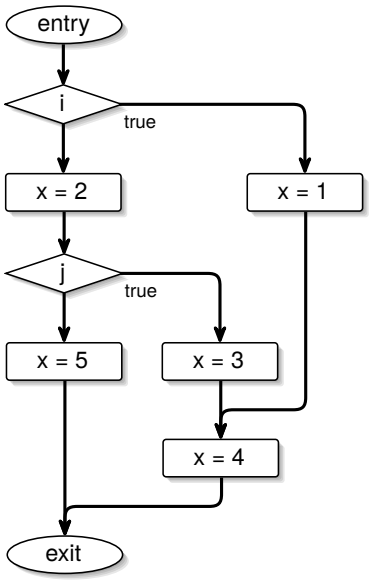


Figure 4.9. SCG representation of the `JumpIntoBranch` program

The future work section of Rathlev [Rat15] also points out this problem when discussing the translation from SCL into Esterel. He presents a solution which includes code duplication to recreate a translatable structure. However, this thesis focuses on the semantical translation in combination with SSA and thus excludes all jump structures which cannot translated using if-then-else structures. This restriction is also made in Section 4.2.3, when analyzing the program structure to create merge expressions.

Another case are backward jumps forming loops, which are more common to SC programs than complex forward jump structures. Backward jumps may lead to irreducible flowgraphs whose structure cannot be translated into Esterel and thus are permitted. This is the same restriction made in loop handling of the SSA form, presented in Section 4.2.5. The reducibility property assures that the cycle has only one entry point and does not overlap with other cycles. This allows to encapsulate the loop body in the Esterel loop structure. All jumps to the entry label of the loop are transformed into traps resulting in the continuation of the loop. The jumps targeting a point outside the loop result in traps leaving the loop body.

Table 4.4 illustrates the translation of loop structures. The essential part is the jump to the `LoopEntry` forming the actual loop. The condition around the backward jump is optional, to form infinite loops. Moreover, the jump exiting loop is also optional and the subprograms `P1`, `P2`, and `P3` can contain further jumps to the `LoopExit`.

SCL	Esterel
1 LoopEntry: 2 // Subprogram P1 3 **if** *condition* **then** 4 **goto** LoopExit 5 **end** 6 // Subprogram P2 7 **if** *condition* **then** 8 **goto** LoopEntry 9 **end** 10 // Subprogram P3 11 LoopExit:	1 **trap** LoopExit 2 **loop** 3 **trap** LoopEntry 4 % Subprogram P1 5 **present** *condition* **then** 6 **exit** LoopExit 7 **end**; 8 % Subprogram P2 9 **present** *condition* **then** 10 **exit** LoopEntry 11 **end**; 12 % Subprogram P3 13 **exit** LoopExit 14 **end trap** 15 **end loop** 16 **end trap**

Table 4.4. Translation pattern for loops

4.3.2 Behavior

Besides the structural aspects in the translation into Esterel, there are also semantical aspects which must be considered. The main difference is the use of signals instead of variables. The SC MoC extends the semantics of Esterel and consequently provides a scheme for handling signals with variables, but translating the other way around raises some challenges.

Interface

The semantics of signals allow only one globally consistent state per tick. This is the initial motivation for using the SSA form to assure this semantics while supporting multiple values per tick. However, there is one situation where the SSA form is not able to assure this property and that is the case of input output variables. A variable marked as input and output is not renamed by the SSA transformation, to retain the interface. Since the program can assign this variable, the conveyed value can differ from the input value given by the environment. Assuming boolean variables encoded as single signals, an SC programs can read a true value, respectively present and convey false, respectively absent. This behavior is considered an unemit and is not supported by an Esterel environment communicating with the program via its interface. Due to the different semantics of signals in comparison to variables in Esterel, programs may be not semantically equivalent or non-constructive.
A solution is to split the input output variable into two different variables, one input and one output variable, with appropriate names. Consequently, this alters the interface and rules out an equivalent behavior based on the same interface. However, this is tolerable since in the translation into Esterel, the variables are changed to signals anyway, resulting in an interface

with different types. Hence, when comparing the behavior of the source program with the generated Esterel program, the environment always has to adapt to the new interface.
Another aspect are augmented input variables. The definition of augmented variables includes the introduction of an additional signal to indicate the presence of a written value. This signal is also checked by the merge functions. However, since a input variable is set by the environment in each tick, there is no assignment emitting the corresponding presence signal. One solution is the explicit emission of the presence signals of each input in every tick. However, to reduce the number of signals and presence test, this thesis prefers the approach of having no additional presence signal for inputs.

Initialization

Regarding the definition of signals, they are always considered absent, if not emitted. Consequently, they always have a defined value and can be considered initialized to absent. In contrast, the SC MoC does not define such an implicit initialization for variables. If a variable is read before any write has defined its value, the value is undefined. The SC MoC expects the programmer to explicitly initialize all variables correctly.
When a program with uninitialized reads is translated into Esterel and executed with initialized signals, the program may behave different to other environments. Hence, the situation of uninitialized reads must be detected and the programs needs to be rejected, to assure that accepted programs behave correctly independent from initializations in the environment.
Every assignment reading a variable, which is not dominated by an initialization of that variable, must be additionally checked for a possible uninitialized read. Using augmented values, the presence flag indicates an active write. Consequently, in the translation rule for merge functions, the case where none of the read definitions are executed must be detected and the program must be rejected. This also works for the usage of $pre(s)$ for a signal s in the initial tick, because all signals are considered absent prior to the first tick [Ber99].

Delayed Loops

Another considerable aspect are loops. In Esterel a loop must not instantaneously reach the end of the loop body from its head. Thus, Esterel cannot perform multiple iterations in the same tick. Nevertheless, the loop may be left and reentered in the same tick, but since the loop body must be delayed this loop instance cannot terminate instantaneously. Hence, there are instantaneous loops which can be structurally translated into Esterel but are rejected by the Esterel compiler. However, the handling of loops in the SSA transformation, presented in Section 4.2.5, is restricted to an even stronger from of delayed loops if multiple assignments to the same variable are present in the loop body.
Furthermore, loops are known to cause the problem of schizophrenia in some situations. That means statements are executed more than once in the same tick. In combination with the globally consistent state of signals, this may cause unexpected behavior. The SSA form only assures the single assignment statically and consequently does not prevent multiple

x_i \ **not_x**$_i$	**present**	**absent**
present	illegal	true
absent	false	undef

Table 4.5. Pure signal encoding inspired by unemit

x_i^p \ x_i	**present**	**absent**
present	true	false
absent	undef	undef

Table 4.6. Pure signal encoding separating presence flag and value

execution of statements due to loops in the actual controlflow. Curing schizophrenia is a complex task [TS04] and not targeted by this thesis.
Alternatively there is Dynamic Single Assignment (DSA) which is an SSA variant ensuring that variables are are written at most once [VJB+07]. DSA is primarily designed for scalars and arrays, manipulated in loops.

4.3.3 Pure Signal Encoding

The available Esterel compiler supports a complete constructiveness analysis only for programs using pure signals. The values of valued signals are conservatively not evaluated even if this would be possible [PEB07]. Hence, a pure signal encoding for variables is required to check the constructiveness of SC programs with Esterel.
Encoding variables with signals includes the translation of variables, their values, and expressions using variables. This includes merge expressions. Additionally, conditions in if statements need to be translated into present tests, comparing the corresponding signals.
Signals represent the value set $\{absent, present\}$. This is sufficient to encode a boolean variable in a single signal. Furthermore, C-like boolean variables, using an integer with the effective value set $\{0,1\}$, can also represented by a signal. Larger value ranges and arbitrary data types can be represented by multiple signals using binary or one-hot encoding. The solution for pure signal encoding in this section only presents rules and patterns for the handling of boolean variables and integers restricted accordingly.

Value Encoding

A single signal can encode one boolean variable, but the merge functions of the SSA form use augmented values, requiring an additional signal which is emitted when the value is assigned to a variable. There are two different variants to represent an augmented boolean value.
Table 4.5 shows an encoding inspired by emit and unemit states. Each augmented boolean variable x_i is encoded by two signals x_i and not_x_i. The presence of the signal x_i indicates that the variable is written with a true value in this tick, and the not_x_i that it is set to false. The two signals represent an emit and unemit operation on the variable. If none of the signals is present, the variable was not written at all, indicating the absence of any value. The presence of both signals is considered an illegal state, which must not occur because the value cannot be true and false at the same time. The advantage of this encoding is that in case of assignments using constants, the assigned variable can be represented by only one of

the signals indicating the value. Moreover, a one-hot encoding can be used to encode more non-boolean data types. However, checking the absence of any write action always requires a test on both signals, because each signal indicates both the value and a partial presence of the write. Since the constructiveness analysis of Esterel uses a three valued logic, it is desired to provide information about a write independent from the actual value to facilitate speculation on absence.
The other encoding variant presented in Table 4.6, is directly derived from the notation for augmented values. For a variable x_i two signals x_i^p and x_i are created, where x_i^p indicates the presence of the value, respectively the execution of a write, and x_i the actual boolean value. Consequently, both signal must always be appropriately set when a variable is written. However, the separation of presence and value allows binary encoding for more complex data types. More importantly it enables the encoding of the merge function such that the constructiveness analysis can speculate on the absence of write independent from the computed value. Hence, the latter variant is used to defined the following encoding rules.

Assignments and Conditionals

Table 4.7 shows the translation rules for assignments and conditionals. Constant assignments are translated by simply emitting the corresponding signals. If the value of an expressions is assigned, for example the result of a merge expression, then the expressions is first dismantled and assigned to a new unique intermediate variable v. $v \leftarrow e$ indicates the evaluation of an expression using the rules defined in the following section. Afterwards, the intermediate variable v is used to assign the correct value to x_i. If the expression only consists of a single variable, for example $x_i = x_j$, then no additional intermediate variable v is needed. The same goes for the translations of conditionals. Note that the `error` signal activates a non-constructive program section and indicates that a variable was read without having a valid value. The initialization check can be omitted if the reading statement is dominated by an initialization. Moreover, if a read variable is an input the presence test of its presence signal is omitted.

Expressions

Expressions in the SCL source code can consist of constants, variables, operations including compare operations and merge expressions. They can be hierarchically nested using precedence and parentheses. The merge expressions are nested as well but only contain merge functions and variables. Using augmented values with such expressions requires a dismantling to translate each component. Table 4.8 illustrates the pattern for dismantling expressions with introducing new variables for intermediate results. All boolean operators in SCL have equivalent counterparts in Esterel evaluating signals. Consequently, all operations can be translated using a general pattern, shown in Table 4.9. The equals and not-equals comparison operation requires a more complex representation in boolean logic. Note that the operations are only performed when both input variables are present. Hence, if any referenced variable is not correctly initialized, the operation result is absent. This propagates through the entire

SCL	Esterel
`1 x_i = true`	`1 emit x_i^p` `2 emit x_i`
`1 x_i = false`	`1 emit x_i^p`

`1 x_i = e`	`1 v ← e;`
	`2 present v^p then`
	`3  emit x_i^p`
	`4  present v then`
	`5  emit x_i`
	`6  end`
	`7 else`
	`8  emit error`
	`9 end`

`1 if (e) then`
`2  //then-block`
`3 else`
`4  //else-block`
`5 end`
	`1 v ← e;`
	`2 present v^p then`
	`3  present v then`
	`4  % then-block`
	`5  else`
	`6  % else-block`
	`7  end`
	`8 else`
	`9  emit error`
	`10 end`

with: e expression	

Table 4.7. Translation patterns for assignments and conditionals using pure signals

expression and the assignment or conditional reading the result of the expression will fail with the not-initialized error. If an operand is a constant, the general pattern can be statically evaluated removing the corresponding signals. Hence, an equality check with zero will result in a negation of the signal state. Monadic operations can be directly included into the signal expression of the present test determining the value. The same holds for the *pre*-function when a register variable is read. This expression neither requires dismantling. Furthermore, if a read variable is an input the presence test of its presence signal is omitted.
The implementation of the merge functions can be directly derived form their definitions, since they are designed with Esterel in mind. Table 4.10 presents the Esterel implementation of the three merge functions. Note that again the `error` signal activates a non-constructive program section to reject the program.

Composed Expression	Decomposed Expression
e `op` $\mathtt{x}_i$	$\mathtt{v} \leftarrow e;$ $\mathtt{v}\ op\ \mathtt{x}_i$
r `op` $\mathtt{x}_i$	$\mathtt{v} \leftarrow r;$ $\mathtt{v}\ op\ \mathtt{x}_i$
$f(\mathtt{x}_i, r)$	$\mathtt{v} \leftarrow r;$ $f(\mathtt{x}_i, \mathtt{v})$
$f(r, \mathtt{x}_i)$	$\mathtt{v} \leftarrow r;$ $f(\mathtt{v}, \mathtt{x}_i)$
$f(r_0, r_1)$	$\mathtt{v}_0 \leftarrow r_0;$ $\mathtt{v}_1 \leftarrow r_1;$ $f(\mathtt{v}_0, \mathtt{v}_1)$
$combine(op, r, \mathtt{x}_{up})$	$\mathtt{v} \leftarrow r;$ $combine(op, \mathtt{v}, \mathtt{x}_{up})$
with: e expression, op operator, $f \in \{seq, conc\}$, r resolve expression	

Table 4.8. Translation patterns for dismantling expressions using pure signals

Expression	Esterel

$\mathtt{v} \leftarrow \mathtt{x}_i\ op_{bin}\ \mathtt{x}_j$	`present` $\mathtt{x}_i^p$ `and` $\mathtt{x}_j^p$ `then`
	  `emit` $\mathtt{v}^p$;
	  `present` $\mathtt{x}_i\ op_{bin}\ \mathtt{x}_j$ `then`
	  `emit` v
	  `end`
	`end`

$\mathtt{v} \leftarrow \mathtt{x}_i$ `==` $\mathtt{x}_j$	`present` $\mathtt{x}_i^p$ `and` $\mathtt{x}_j^p$ `then`
	  `emit` $\mathtt{v}^p$;
	  `present` ($\mathtt{x}_i$ `and` $\mathtt{x}_j$) `or` (`not` $\mathtt{x}_i$ `and not` $\mathtt{x}_j$) `then`
	  `emit` v
	  `end`
	`end`

$\mathtt{v} \leftarrow \mathtt{x}_i$ `!=` $\mathtt{x}_j$	`present` $\mathtt{x}_i^p$ `and` $\mathtt{x}_j^p$ `then`
	  `emit` $\mathtt{v}^p$;
	  `present` ($\mathtt{x}_j$ `and not` $\mathtt{x}_j$) `or` (`not` $\mathtt{x}_j$ `and` $\mathtt{x}_j$) `then`
	  `emit` v
	  `end`
	`end`

with: op_{bin} binary operator	

Table 4.9. Translation patterns for single operator expressions using pure signals

Resolve Function	Esterel
1 v ← $seq(x_i, x_j)$	1 present x_j^p then 2 emit v^p; 3 present x_j then 4 emit v 5 end 6 else 7 present x_i^p then 8 emit v^p; 9 present x_i then 10 emit v 11 end 12 end
1 v ← $conc(x_i, x_j)$	1 present x_i^p and x_j^p then 2 present (x_i and not x_j) or (x_i and not x_j) then 3 emit error 4 else 5 emit v^p; 6 present x_i then 7 emit v 8 end 9 else 10 present x_i^p or x_j^p then 11 emit v^p; 12 present x_i or x_j then 13 emit v; 14 end 15 end 16 end
1 v ← $combine(op_{bin}, x, x_{up})$	1 present x^p then 2 emit v^p; 3 present x_{up}^p then 4 present x op_{bin} x_{up} then 5 emit v 6 end 7 else 8 present x then 9 emit v 10 end 11 end 12 else 13 present x_{up}^p then 14 emit error 15 end 16 end
with: op_{bin} binary operator	

Table 4.10. Translation patterns for merge functions using pure signals

```
1 if x == (seq(y0, y1) | z) then
2    // then-branch
3 end
```

(a) Truncated source program in SCL

```
1 temp0 ← seq(y0, y1)
2 temp1 ← temp0 | z
3 temp2 ← x == temp1
4 present temp2p then
5   present temp2 then
6     % then-block
7   end
8 else
9   emit error
10 end
```

(b) Intermediate result after dismantling the expressions

```
1  present y1p then
2    emit temp0p;
3    present y1 then
4      emit temp0
5    end
6  else
7    present y0p then
8      emit temp0p;
9      present y0 then
10       emit temp0
11     end
12 end;
13 present temp0p and zp then
14   emit temp1p;
15   present temp0 or z then
16     emit temp1
17   end
18 end;
19 present xp and temp1p then
20   emit temp2p;
21   present (x and temp1) or (not x and not temp1)
         then
22     emit temp2
23   end
24 end;
25 present temp2p then
26   present temp2 then
27     % then-block
28   end
29 else
30   emit error
31 end
```

(c) Resulting Esterel program

Listing 4.26. Truncated example program for expressions translation

Listing 4.26 illustrates the translation of a conditional expression into Esterel. Listing 4.26a shows the source statement which is translated. The surrounding program is ignored in this example to focus on the expression itself. Listing 4.26b presents an intermediate step in the transformation. The conditional expression is dismantled using the same intermediate notation as presented in the corresponding patterns. Each operation is performed separately and the result is subsequently evaluated by the surrounding operation. The introduced intermediate variables are named `temp`. Each variable is represented by two signals, the one with appended *p* is considered the signal indicating the presence of the value, the other is the value. The conditional is translated using the corresponding pattern, starting at line 4. The first present test checks the signal indicating if the conditional expression yields a valid result.

If this signal is not set, some variable is not correctly initialized and the program is rejected by emitting the error signal in line 9. The second present test performs the actual branching based on the boolean result of the condition. Listing 4.26c shows the final resulting Esterel program, separately evaluating each operator and merge function. Lines 1 to 12 represent the *seq*-function, lines 13 to 18 the logical or, and lines 19 to 24 the equals comparison. The result is evaluated by the translated if pattern.

Rejecting Programs

The `error` signal is used to dynamically reject programs in the process of the Esterel constructiveness analysis. The signal itself is declared global to the programs and can be used by all *conc*-functions and initialization checks. If the signal is emitted, it activates a non-constructive program section which runs concurrent to the source program. For delayed programs the section is encapsulated in a loop or placed directly into the loop handling the registers of the SSA form. Listing 4.27 illustrates the general pattern for inserting the error section. The original program will be inserted at line 4 and the subsequently emitted term signal terminates the loop handling the error signal. The non-constructive section starts at line 12, it emits the `errorhelper` based on its own state. The `error` signal activates this section, which will cause the Esterel constructiveness analysis to reject the program iff the `error` signal can be present.

4.3.4 Valued Signal Encoding

Another approach for encoding variables is to use valued signals. Valued signals carry in addition to their signal state a value which is persistent across ticks. Hence, this encoding is well suited to represent augmented values, especially with more complex data types than boolean. However, valued signals are not part of the kernel language and evaluating signal values is not part of the implemented Esterel constructiveness analysis. Consequently, the valued signal encoding can only be used for creating equivalent programs and not for checking constructiveness with the Esterel compiler.
The translation of assignments and conditionals requires the same dismantling as the pure signal encoding, presented in Section 4.3.3. Table 4.11 shows the pattern for transforming assignments and conditionals including the general pattern for operations and Table 4.12 presents the implementation of the merge functions.

```
module ErrorPattern:
signal error, term in
[
 % Original program here
 emit term
||
 trap termError
  loop
   % Error handling
   signal errorhelper in
    present error then
     present errorhelper else
      emit errorhelper
     end
    end
   end signal;
   % Exiting loop when original program terminates
   present term then
    exit termError
   end;
   pause
  end loop
 end trap
]
end signal
end module
```

Listing 4.27. Pattern for rejecting programs

SCL	Esterel
1 x_i = c	1 **emit** $x_i(c)$
1 x_i = e	1 v ← e; 2 **present** v **then** 3 **emit** x_i(?v) 4 **else** 5 **emit** error 6 **end**
1 **if** (e) **then** 2 //then-block 3 **else** 4 //else-block 5 **end**	1 v ← e; 2 **present** v **then** 3 **if** ?v **then** 4 % then-block 5 **else** 6 % else-block 7 **end** 8 **else** 9 **emit** error 10 **end**
1 v ← x_i op x_j	1 **present** x_i **and** x_j **then** 2 **emit** v(?x_i op ?x_j) 3 **end**
with: c constant value, e expression, op operator	

Table 4.11. Translation patterns for assignments, conditionals, and operations using valued signals

Resolve Function	Esterel
1 v ← $seq(x_i, x_j)$	1 present x_j then 2 emit v(?x_j) 3 else 4 present x_i then 5 emit v(?x_i) 6 end 7 end
1 v ← $conc(x_i, x_j)$	1 present x_i and x_j then 2 if ?x_i <> ?x_j then 3 emit error 4 else 5 emit v(?x_i) 6 else 7 present x_j then 8 emit v(?x_j) 9 else 10 present x_i then 11 emit v(?x_i) 12 end 13 end 14 end
1 v ← $combine(op, x, x_{up})$	1 present x then 2 present x_{up} then 3 emit v(?x op ?x_{up}) 4 else 5 emit v(?x) 6 end 7 else 8 present x_{up} then 9 emit error 10 end 11 end
with: op operator	

Table 4.12. Translation patterns for merge functions using valued signals

Chapter 5

Implementation

This chapter describes the implementation of the concept presented in Chapter 4. The implementation is part of the KIELER project, which provides a programming environment and compiler for the languages related to the SC MoC. The KIELER project already contains EMF meta-models or Xtext grammars for SCL, SCG, Esterel, and more, implemented in previous work [Smy13; Rüe11]. This facilitates the implementation of an SSA transformation and translation into Esterel. At first, Section 5.1 describes the integration of these two transformations into the KIELER project. Secondly, Section 5.2 presents the SSA transformation based on SCGs. Subsequently, the translation into Esterel is presented in Section 5.3. Both sections illustrate their transformation on the example program `P10` from Listing 1.3 on page 3.

5.1 Integration into KIELER

The KIELER project with its generic compiler infrastructure KiCo provides multiple translations between different synchronous and SC languages. Figure 2.8 on page 16 illustrates the language translations available before this thesis. Figure 5.1 shows the diagram extended by the SSA form, presented in this thesis. The node highlighted in red represents the SSA form based on an SCG, and the red arrow starting at the general SCG illustrates the transformation into this form. The implementation of the SSA form is based on an SCG, because it is a general and basic form of representing an arbitrary SC program. With respect to SCCharts, SyncCharts, and Esterel the SCG is the lowest common basis in the current compile chain. An SCG representation is fully equivalent to its SCL program, but is based on a different meta-model allowing the integration of dependencies, basic blocks and scheduling information. The red arrow from SSA-SCG to Esterel represents the translation into Esterel, facilitated by the SSA form.
Nevertheless, Figure 5.1 presents only an abstract view on the available translations. The actual transformations implemented for the KiCo infrastructure are usually smaller modular

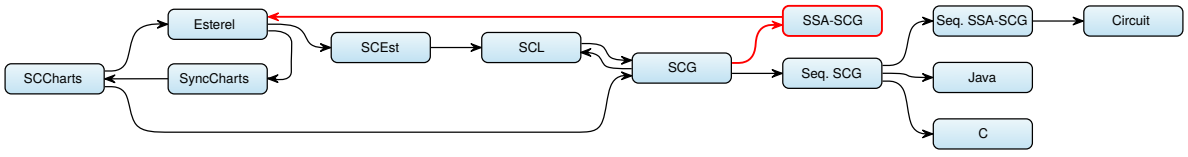


Figure 5.1. KIELER compilation overview with new SSA form for genral SCGs

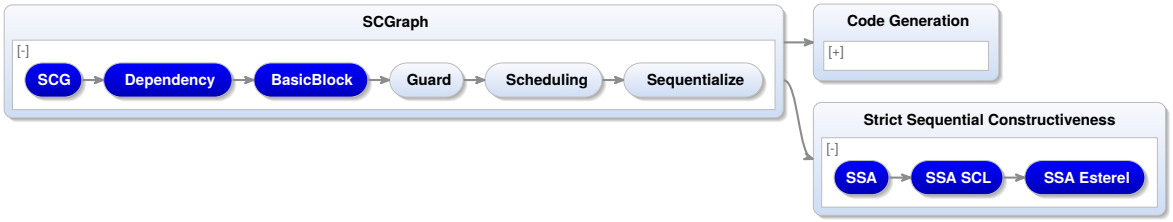


Figure 5.2. SCG compile chains with new compilation path into Esterel

compilation steps. Figure 5.2 shows a screenshot of the transformations presented by KiCo available for the SCG, including the new compile chain to produce Esterel code. The surrounding nodes categorize and bundle the contained transformations. The transformations inside `SCGraph` are the implementation of the dataflow compilation approach, mentioned in Section 2.2. They result in a Sequentialized SCG, which can be further translated into for example C code. The Esterel compile chain includes some of the more general transformation of the dataflow chain, which are highlighted in blue. This way the existing dependency analysis and basic block separation can be re-used. The remaining transformations are skipped and the SSA transformation is performed next, processing an SCG with dependencies and basic blocks. Section 5.2 presents the details of this transformation. Subsequently, the SCG in SSA form is translated into an intermediate SCL representation and finally translated into Esterel, further described in Section 5.3.

5.2 SSA Transformation

The SSA transformation implements the concept presented in Section 4.2. At first, the SCG is traversed to find variable references in assignment and conditional nodes. Each reference is replaced by a merge expression based on all assignments to the corresponding variable. Afterwards, the SSA variable versions are created and each assigned variable is renamed accordingly. Figure 5.3 shows the result of the SSA transformation applied on the SCG representation of the `P10` program from Listing 1.3 on page 3. The assigned variables `x` and `y` are split up into versions and the assignment to y_1 uses a merge expression to resolve the correct definition of `x`. Since the variable is not written by an update, the implementation of the patterns in Table 4.2 is used to create the expression. Furthermore, the expression is reduced to its context and transformed into the compact form, according to Section 4.2.3. The reduction requires dominator relations between the nodes in the SCG. Hence, a dominator analysis is implemented based on the book about compiler implementation by Appel and Palsberg [AP02]. The actual algorithm is developed by Lengauer and Tarjan [LT79]. To reduce the number of nodes processed by the algorithm and thus the runtime, it is applied to the basic blocks structure of the SCG. This allows to determine the dominator relation between the actual nodes. A diagram synthesis is implemented using the KLighD framework, to visualize and inspect the generated dominator tree. Figure 5.4 illustrates the dominator tree for the

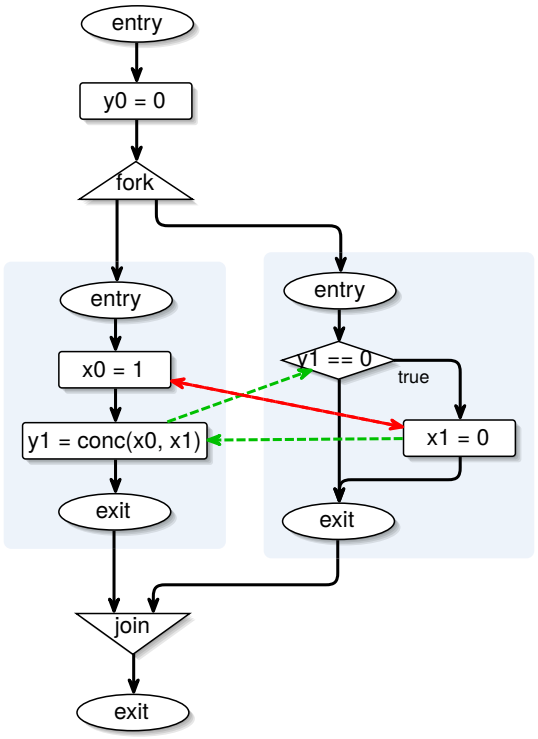


Figure 5.3. SCG representaion of the `P10` program in SSA form

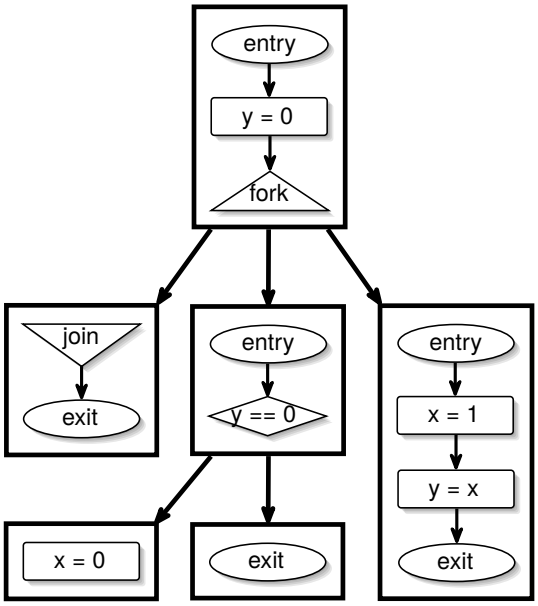


Figure 5.4. Dominator tree of basic blocks in the `P10` SCG

basic blocks of the original `P10` SCG. To enable this analysis the SSA transformation requires a previous basic block transformation. The combination of dependencies in the SCG and dominator relations allows the reduction of the merge expression to their context in the SCG. This way, the merge expression initially introduced for reading the correct value of `x` in the conditional of the second thread of `P10` is reduced to just y_1. This can be made, because y_1 fulfills the requirement for an instantaneously concurrent dominant definition and thus always overrides y_0, described in Section 4.2.3.

Furthermore, the implementation allows the creation of merge expressions for variables written by updates. In such situations, the SCG is copied and all nodes which assign variables other than the analyzed one are removed. Afterwards, the SCG is compiled with KiCo using the dataflow compile chain, illustrated in Figure 5.2, until a static scheduling is generated. Since the copying of an EMF model and the compilation in KiCo allows the tracing of model elements, the scheduled nodes can be associated with the nodes in the original SCG. Hence, the schedule is used to generated a merge expression based on the implementation of Algorithm 1 on page 52.

In an SCG which contains surface and depth nodes, an additional thread is created to handle the persistence of merged variable values. The SCG is restructured to fork the thread before

```
1  module P10-SSA
2  bool x0, x1;
3  bool y0, y1;
4  bool temp0;
5  {
6    y0 = false;
7    fork
8      x0 = true;
9      y1 = <conc(x0, x1)>;
10   par
11     temp0 = y1 == false;
12     if temp0 then
13       x1 = false;
14     end;
15   join;
16 }
```

Listing 5.1. The `P10` program in intermediate SSA form

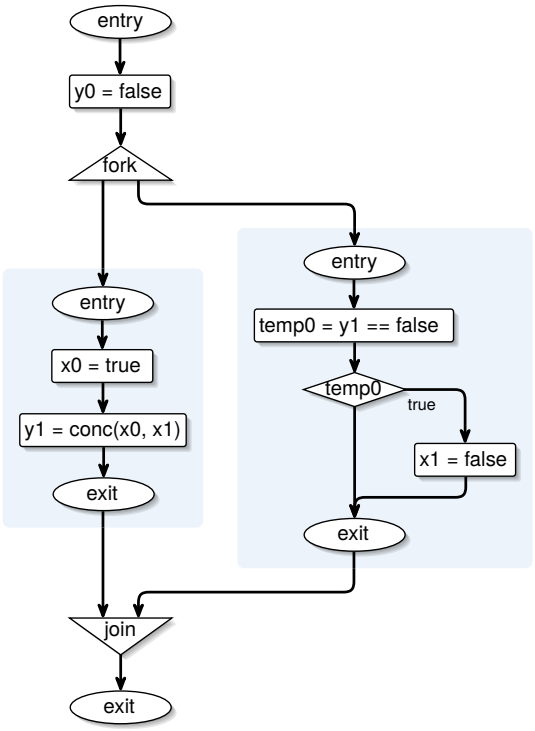


Figure 5.5. SCG representation of program `P10` in intermediate SSA form

the actual program starts and the register variables are created and assigned using generated merge expressions. Since all definitions in the original program are concurrent to this new thread the expressions are not reduced. Moreover, the register variables would be considered in the creation of the merge expressions in the original program.
The reordering of definitions which are located in a loop to prevent sequentiality inversion could not be implemented in the limited time frame of this thesis. Hence, the implementation cannot correctly handle multiple assignments to the same variable in a loop. However, all SCGs without such assignments are correctly translated. The same holds for forward jump structures, as mentioned in Section 4.3.1.

5.3 Translation into Esterel

The translation into Esterel, described in Section 4.3, is split up into two separate transformations. The first is a translation into an intermediate SCL representation. The reason for using SCL as intermediate language is the structure of the implemented meta-model. Similar to Esterel, the statements in SCL are hierarchically nested, whereas an SCG has a flat structure of nodes and edges. Hence, the translation into SCL is used to pre-process the structure of the SCG

and to facilitate the translation into Esterel. Moreover, a translation from SCG to SCL is already implemented in the KIELER project and is re-used in this transformation. Another aspect is the dismantling of expressions, described in Section 4.3.3. It is reasonable to dismantle nested expressions in SCL and to introduce new intermediate variables, since the variables are not yet represented by multiple signals. Prior to this dismantling the merge expressions are normalized, according to Section 4.2.3, to facilitate their dismantling. Furthermore, this transformations translates the interface of variables declared as input and output into separate variables, according to Section 4.3.2.
Listing 5.1 shows the resulting intermediate representation in SCL for the `P10` program. The conditional expression is dismantled, resulting in an assignment to an intermediate variable $\texttt{temp}_0$ in line 11. The assignment to $\texttt{y}_1$ does not require a dismantling because it contains only a single merge function. The pointy brackets around the *conc*-function indicate a function call in the SCL grammar. Figure 5.5 illustrates the SCG representation of this intermediate representation to ease the visual relation to Figure 5.3.
The second transformation is the actual translation into Esterel, implementing the patterns presented in Section 4.3.3. Listing 5.2 shows the result of this transformation for the program `P10`. The assignment of $\texttt{y}_0$ is represented in line 7, emitting the presence signal to indicate that the assignment is active and leaving the value signal unemitted to encode the false value by absent. Lines 11 to 27 show the translated *conc*-function assigned to $\texttt{y}_1$, including the conflict check and the emission of the error signal in such situations. The test for equality to zero is represented by the lines 29 to 34. The constant value in the expression is statically resolved to a negation of the value of $\texttt{y}_1$, storing the result in $\texttt{temp}_0$. The subsequent code checks this result and performs the branching with the then branch in line 37. Additionally, the translation introduces a code section reacting on the emission of the error signal. In this case a new thread is created containing the error pattern in lines 44 to 50.
If the program were delayed, i. e., would contain a pause statement, this section would be directly inserted into the concurrent thread handling the register variables. Similar to the implemented SSA transformation, the implementation of the translation into Esterel does not handle loops. This does not mean that the transformation fails for such programs, but it does not take the backward jumps into account. The code is generated such that the gotos and labels are not translated. This allows to manually insert the necessary loop and trap statements afterwards. A future implementation may contain this translation, but in the limited time frame of this thesis it was not possible to implement an analysis which automatically determines the statements contained in a loop body.

```
module P10:
signal x0p, x0, x1p, x1 in
signal y0p, y0, y1p, y1 in
signal temp0p, temp0 in
signal error in
[
 emit y0p;
 [
  emit x0p;
  emit x0;
  present x0p and x1p then
   present (not x0 and x1) or (x0 and not x1) then
    emit error
   else
    emit y1p;
    present x0p then
     emit y1
    end
   end
  else
   present x0p or x1p then
    emit y1p;
    present x0 or x1 then
     emit y1
    end
   end
  end
 ||
  present y1p then
   emit temp0p;
   present not y1 then
    emit temp0
   end
  end;
  present temp0p then
   present temp0 then
    emit x1p
   end
  else
   emit error
  end
 ]
||
 signal errorhelper in
 present error then
  present errorhelper else
   emit errorhelper
  end
 end
 end signal
]
end signal
end signal
end signal
end signal
end module
```

Listing 5.2. The `P10` program translated into Esterel

Chapter 6

Evaluation

This chapter evaluates the concept of checking the Esterel constructiveness of SC programs by performing an SSA transformation and a subsequent translation into Esterel. First, Section 6.1 summarizes the different restrictions to the SC programs introduced in Chapter 4 to evaluate the number of programs supported by the presented approach. Secondly, Section 6.2 presents some characteristic programs and tests their compliance with the class of SSC programs. The programs are checked for their constructiveness in Esterel and tested for a semantically equivalent behavior. Finally, Section 6.3 presents limitations to the Esterel constructiveness analysis caused by the presented SSA definition and translation into Esterel.

6.1 Supported Programs

The concepts for the SC-specific SSA transformation and the translation into Esterel in Chapter 4 introduce some restrictions to the supported programs. Restrictions such as prohibiting complex forward jump structures are made to be able to handle more important structures and aspects without providing a general solution. This explicitly limits the number of programs which can be tested for SSC, disregarding their actual compliance with constructiveness in Esterel.
Figure 6.1 shows the supported SC program classes. The classification is based on the language features presented in Section 4.2, illustrated by a Venn diagram. It shows all possible combinations of concurrent, delayed, cyclic, and programs with updates. Moreover, the diagram illustrates the restricted subsets of these classes. The gray areas indicate not supported classes of programs, based on the explicit restrictions in Chapter 4.
Sections 4.2.3 and 4.3.1 describe that forward jumps which cannot directly expressed by if-then-else constructs without gotos are not supported. This is done to simplify the analysis and to prevent the problem of jumps targeting into other structures, such as loops or conditional branches. Since this limitation restricts the general structure, it affects all SC program classes and is represented by the gray bar in the middle of the diagram intersecting with all classes.
Additionally, Section 4.3.1 restricts the class cyclic SC programs to those which form reducible flowgraphs. This is done to comply with the loop structures in Esterel. Moreover, Esterel itself is restricted to non-instantaneous loops. Hence, the class of cyclic programs supported by the concept lies within the class of delayed programs. Another restriction concerning loop structures is introduced in Section 4.2.5. The concept of reordering variable versions in merge expressions which consists of multiple definitions located in a loop requires at least one

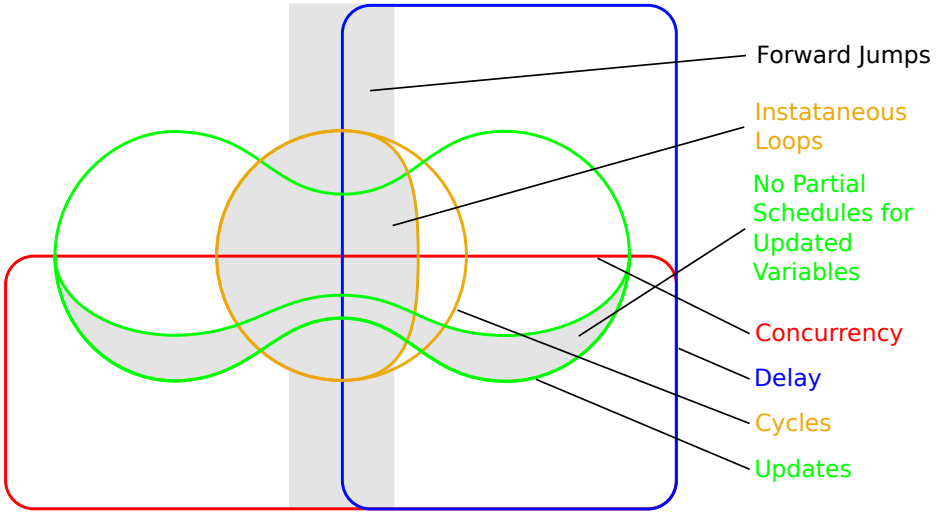


Figure 6.1. Supported SC program classes

pause statement which is reached in every iteration. In the diagram, the unsupported class of instantaneous cycles also covers loops which do not contain such a pause statement.
The last restriction is made in Section 4.2.6. This section uses partial static schedules to create merge expressions for variables written by updates. Consequently, this requires that a program reduced to that variable is statically schedulable. The separated area inside the class of programs using updates indicates this class of programs with have cyclic dependencies between statements including a variable which is written by updates. Since such dependency cycles can only occur in concurrent programs the area does not intersect with the class of non-concurrent programs.
Section 7.2.3 describes possible approaches to overcome these restrictions in future work.

6.2 Test Cases

This section presents some characteristic SC programs to evaluate the capabilities of detecting SSC programs. The programs are translated into SSA form and subsequently into Esterel using the implementation in the KIELER project, presented in Chapter 5. The concepts which could not be implemented in the scope of this thesis were additionally transformed manually according to their definition in Chapter 4. For the constructiveness analysis, the Esterel compiler in version V5_92[1] is used. The analysis is invoked using the following options:

```
esterel -Icheck <file>
```

The `-Icheck` options causes the compiler to perform a complete constructiveness analysis and additional checks for single emission of single signals. No executable code is generated in this process, but the analysis is based on a translation into a circuit representation of the program.

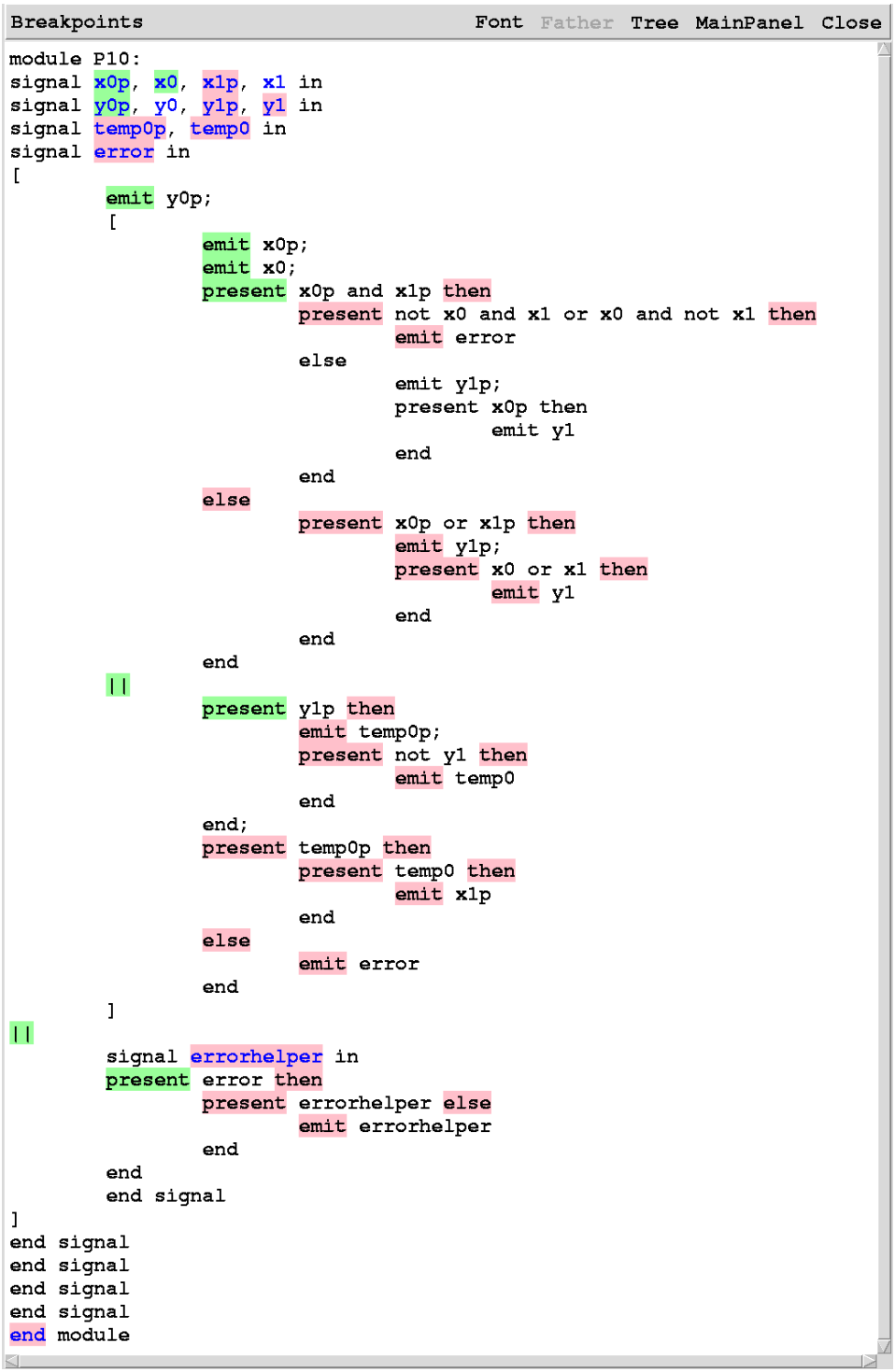
[1] `http://www-sop.inria.fr/esterel-org/filesv5_92/home.htm`

6.2.1 P10

The program `P10` is the motivating example of this thesis. Section 1.2 presents the program in Listing 1.3 on page 3. It is considered SC but cannot considered constructive in the sense of Esterel. Hence, it should be rejected as non-SSC. In Chapter 5 the program `P10` is used to illustrate the translation into SSA form and subsequently into Esterel code. Listing 5.2 on page 80 shows the translated program in Esterel.
This program can be analyzed for its constructiveness using the Esterel compiler. As expected the compiler yields the result that the program, particularly the analyzed circuit, is not considered constructive in the sense of Esterel. Figure 6.2 illustrates the reasoning provided by the compiler. The highlighting in the program represents the point where the constructiveness analysis stopped because no further values could be constructively determined. In the signal declaration at the top of the program, the green highlighting indicates that the signal state is present. The red highlighting marks signals which state is unknown. In the program code, statements highlighted in green were executed in the presented tick and statements highlighted in red cannot be executed because they depend on signals which state could not be constructively determined. In the case of `P10` the reasoning points out that the causal dependency between `y` and `x` is the reason for the classification as non-constructive in Esterel. In the second thread, $\mathtt{x}_1$ is written based on the evaluation of $\mathtt{y}_1$ in the conditional. The present test for the $\mathtt{y}_1^p$ signal indicates that the value of `y` is required and merged for the conditional expression. However, $\mathtt{y}_1^p$ is emitted in the first thread, but requires the merged value of `x`. This includes the value potentially assigned in the second thread, indicated by the presence test of $\mathtt{x}_1^p$. Hence, the program is rejected because of this causality cycle between the write and read accesses on `y` and `x`.

6.2.2 ABO

Another program which is characteristic for the SC MoC is `ABO`, shown in Listing 2.2 on page 12. The program uses shared variables and assignments to multiple different values in the same tick. Especially the variable `O1` is interesting in this context, because both threads set the value to true in the same tick. However, since both assigned values are the same, the writes are not conflicting. Moreover, when the last of the concurrent threads terminates, it sets `O1` to true and after the join it is assigned to false. Assuming a similar Esterel program with signals instead of variables, this behavior is considered as an unemit in the sense of SCEst.
To check the Esterel constructiveness of the `ABO` program, it is translated into SSA form. Listing 6.1 shows the resulting program. The loop starting at line 31 handles the definitions of the variables in each tick and calculates the output values. It is constructed using the concepts described in Sections 4.2.4 and 4.2.7. First the definitions of `O1` and `O2` are merged and stored in their corresponding register variables to preserve the current value for the next tick. Then the outputs are conveyed. The assignment of `A` might seem superfluous but since it is an input and output variable, the input value is convey as output. The assignment of `B` is a similar case but with an additional definition inside the program influencing the value of `B`, illustrating



```
Breakpoints                                    Font  Father  Tree  MainPanel  Close
module P10:
signal x0p, x0, x1p, x1 in
signal y0p, y0, y1p, y1 in
signal temp0p, temp0 in
signal error in
[
        emit y0p;
        [
                emit x0p;
                emit x0;
                present x0p and x1p then
                        present not x0 and x1 or x0 and not x1 then
                                emit error
                        else
                                emit y1p;
                                present x0p then
                                        emit y1
                                end
                        end
                else
                        present x0p or x1p then
                                emit y1p;
                                present x0 or x1 then
                                        emit y1
                                end
                        end
                end
        ||
                present y1p then
                        emit temp0p;
                        present not y1 then
                                emit temp0
                        end
                end;
                present temp0p then
                        present temp0 then
                                emit x1p
                        end
                else
                        emit error
                end
        ]
||
        signal errorhelper in
        present error then
                present errorhelper else
                        emit errorhelper
                end
        end
        end signal
]
end signal
end signal
end signal
end signal
end module
```

Figure 6.2. Screenshot of the Esterel compiler reasoning on the constructiveness of the `P10` program

```
1  module ABO-SSA
2  input output bool A, B;
3  bool B0;
4  bool O1reg, O1_0, O1_1, O1_2, O1_3;
5  bool O2reg, O2_0, O2_1;
6  bool term = false;
7  {
8    fork
9      O1_0 = false;
10     O2_0 = false;
11     fork
12       HandleA:
13       if !A then
14         pause;
15         goto HandleA
16       end;
17       B0 = true;
18       O1_1 = true
19     par
20       HandleB:
21       pause;
22       if !seq(B, B0) then
23         goto HandleB
24       end;
25       O1_2 = true
26     join;
27     O1_3 = false;
28     O2_1 = true;
29     term = true
30   par
31     Loop:
32     O1reg = seq(seq(seq(pre(O1reg), O1_0),
           conc(O1_1, O1_2)) O1_3);
33     O2reg = seq(seq(pre(O2reg), O2_0), O2_1);
34     A = A;
35     B = seq(B, B0);
36     O1 = O1reg;
37     O2 = O2reg;
38     if !term then
39       pause;
40       goto Loop
41     end
42   join
43 }
```

Listing 6.1. The ABO program in SSA form

the necessity of these assignments. The outputs O1 and O2 are written using the merged value of the register.
Subsequently, the program in SSA form is translated into Esterel. The current implementation does not translate loops, but still produces appropriate code ignoring the loop. Therefore, the resulting Esterel program in Listing 6.2 contains manually added loop and trap statements. According to the pattern in Table 4.4, loop and trap statements are inserted in lines 18 to 20, 34 to 36, 42 to 44, 71 to 73, 82 & 83, and 218 & 219. Additional exit statements are added in lines 28, 33, 65, 70, and 215.
Performing the constructiveness analysis with the Esterel compiler yields the result that ABO is constructive. Additionally, a simulation of the program using test traces shows that the behavior is equivalent to the original SCL program. Thus, ABO can be considered SSC.
This example shows that SC programs using multiple sequential and concurrent writes can be correctly translated into Esterel and analyzed for their compliance with the constructiveness in Esterel. This includes the correct dynamic detection of non-conflicting concurrent writes using the *conc*-function. It also shows that the original SCL program is significantly more compact than the generated equivalent Esterel code.

```
module ABO:
output Ao;
output Bo;
input Ai;
input Bi;
output O1;
output O2;
signal Aregp, Areg in
signal Bregp, Breg, B0p, B0
    in
signal O1regp, O1reg, O1_0p,
     O1_0, O1_1p, O1_1,
    O1_2p, O1_2, O1_3p, O1_3
     in
signal O2regp, O2reg, O2_0p,
     O2_0, O2_1p, O2_1 in
signal temp0p, temp0, temp1p,
     temp1, temp2p, temp2,
    temp3p, temp3, temp4p,
    temp4, temp5p, temp5,
    temp6p, temp6 in
signal error, term in
[
  emit O1_0p;
  emit O2_0p;
  [
    trap loop0Exit in
    loop
      trap loop0Entry in
      pause;
      emit temp0p;
      present not Ai then
        emit temp0
      end;
      present temp0p then
        present temp0 then
          exit loop0Entry
        end
      else
        emit error
      end;
      exit loop0Exit
      end trap
    end loop
    end trap;
    emit B0p;
    emit B0;
    emit O1_1p;
    emit O1_1
  ||
    trap loop1Exit in
    loop
      trap loop1Entry in
      pause;
      present B0p then
        emit temp2p;
        present B0 then
          emit temp2
        end
      else
        emit temp2p;
        present Bi then
          emit temp2
        end
      end;
      present temp2p then
        emit temp1p;
        present not temp2 then
          emit temp1
        end
      end;
      present temp1p then
        present temp1 then
          exit loop1Entry
        end
      else
        emit error
      end;
      exit loop1Exit
      end trap
    end loop
    end trap;
    emit O1_2p;
    emit O1_2
  ];
  emit O1_3p;
  emit O2_1p;
  emit O2_1;
  emit term
||
  trap loop2Exit in
  loop
    present B0p then
      emit Bregp;
      present B0 then
        emit Breg
      end
    else
      emit Bregp;
      present Bi then
        emit Breg
      end
    end;
    present O1_0p then
      emit temp4p;
      present O1_0 then
        emit temp4
      end
    else
      present pre(O1regp) then
        emit temp4p;
        present pre(O1reg)
            then
          emit temp4
        end
      end
    end;
    present O1_1p and O1_2p
        then
```

Listing 6.2. (1/2) The ABO program translated into Esterel

```
    present (not O1_1 and
        O1_2) or (O1_1 and
        not O1_2) then
      emit error
    else
      emit temp5p;
      present O1_1 then
        emit temp5
      end
    end
  else
    present O1_1p or O1_2p
        then
      emit temp5p;
      present O1_1 or O1_2
          then
        emit temp5
      end
    end
  end;
  present temp5p then
    emit temp3p;
    present temp5 then
      emit temp3
    end
  else
    present temp4p then
      emit temp3p;
      present temp4 then
        emit temp3
      end
    end
  end;
  present O1_3p then
    emit O1regp;
    present O1_3 then
      emit O1reg
    end
  else
    present temp3p then
      emit O1regp;
      present temp3 then
        emit O1reg
      end
    else
      emit error
    end
  end;
  present O2_0p then
    emit temp6p;
    present O2_0 then
      emit temp6
    end
  else
    present pre(O2regp) then
      emit temp6p;
      present pre(O2reg)
          then
        emit temp6
      end
    end
  end;
  present O2_1p then
    emit O2regp;
    present O2_1 then
      emit O2reg
    end
  else
    present temp6p then
      emit O2regp;
      present temp6 then
        emit O2reg
      end
    else
      emit error
    end
  end;
  present Ai then
    emit Ao
  end;
  present B0p then
    present B0 then
      emit Bo
    end
  else
    present Bi then
      emit Bo
    end
  end;
  present O1regp then
    present O1reg then
      emit O1
    end
  else
    emit error
  end;
  present O2regp then
    present O2reg then
      emit O2
    end
  else
    emit error
  end;
  signal errorhelper in
  present error then
    present errorhelper else
    emit errorhelper
    end
  end
  end signal;
  present term then
    exit loop2Exit
  end;
  pause;
 end loop
 end trap
]
end signal
end signal
end signal
end signal
end signal
end signal
end module
```

Listing 6.2 (2/2) The ABO program translated into Esterel

6.2.3 The Token Ring Arbiter

The Token Ring Arbiter is mentioned in Section 4.2.6. It is a constructive program containing a static cyclic dependency [Pan02]. An Esterel implementation of the stations in the Token Ring is given in the Esterel Primer [Ber99]. Listing 6.3 shows the `TokenRingArbiter` program with three stations implemented in SCL. The program uses variables initialized to false in every tick and updates which combine the value with true using a logical or, to model the reset and emit behavior of signals.
The idea of the Token Ring Arbiter is that each station can request the token in each tick using an input `R`. A station gets the token indicated by `G`, if the token starts in this tick at this station or a previous station passes the token to it. If a station could get the token but does not request it, the token is immediately passed to the next station, indicated by `P`. Moreover, in each tick the starting point for the token indicated by `T` will move to the next station. In the given Token Ring Arbiter three stations are connected to each other and the stations form a static cycle because each of them listens for the previous one if the token is passed to them.

```
module TokenRingArbiter
input bool R1, R2, R3;
output bool G1, G2, G3;
bool P1, P2, P3, T1, T2, T3;
{
  fork
    T1 = T1 | true
  par
    Loop1_1: // Station 1
      if T1 | P1 then
        if R1 then
          G1 = G1 | true
        else
          P2 = P2 | true
        end
      end;
      pause;
    goto Loop1_1
  par
    Loop1_2:
      if T1 then
        pause;
        T2 = T2 | true
      else
        pause
      end;
    goto Loop1_2
  par
    Loop2_1: // Station 2
      if T2 | P2 then
        if R2 then
          G2 = G2 | true
        else
          P3 = P3 | true
        end
      end;
      pause;
    goto Loop2_1
  par
    Loop2_2:
      if T2 then
        pause;
        T3 = T3 | true
      else
        pause
      end;
    goto Loop2_2
  par
    Loop3_1: // Station 3
      if T3 | P3 then
        if R3 then
          G3 = G3 | true
        else
          P1 = P1 | true
        end
      end;
      pause;
    goto Loop3_1
  par
    Loop3_2:
      if T3 then
        pause;
        T1 = T1 | true
      else
        pause
      end;
    goto Loop3_2
  par
    SignalReset:
      G1 = false;
      G2 = false;
      G3 = false;
      P1 = false;
      P2 = false;
      P3 = false;
      T1 = false;
      T2 = false;
      T3 = false;
      pause;
    goto SignalReset
  join
}
```

Listing 6.3. The `TokenRingArbiter` program with three stations

Nevertheless, the program is constructive in Esterel because the token is initialized to some station and will always start at any of the stations, dynamically breaking the cycle.
This program allows to evaluate whether the presented approach is capable of handling such programs and allows to detect Esterel constructiveness in the presence of static cycles. Due to the usage of updates in the program the SSA form requires a static schedule to create the merge expression for these variables. In combination with the static cyclic dependency in the program it requires a partial scheduling of the program. Section 5.2 describes that the implementation can create merge expression based on such partial schedules using the static scheduling approach implemented in the compile chain for the SC MoC in KIELER. Listing 6.4 shows the generated intermediate SCL program in SSA form with dismantled expressions. Further translating the program into Esterel and analyzing its constructiveness yields that the generated program is not constructive.

```
1  module TokenRingArbiter-SSA
2  input bool R1, R2, R3;
3  output bool G1, G2, G3;
4  bool G1reg, G1_0up, G1_1;
5  bool G2reg, G2_0up, G2_1;
6  bool G3reg, G3_0up, G3_1;
7  bool P1reg, P1_0up, P1_1;
8  bool P2reg, P2_0up, P2_1;
9  bool P3reg, P3_0up, P3_1;
10 bool T1reg, T1_0up, T1_1up, T1_2;
11 bool T2reg, T2_0up, T2_1;
12 bool T3reg, T3_0up, T3_1;
13 pure term;
14 bool temp0, temp1, temp2, temp3, temp4, temp5,
        temp6, temp7, temp8, temp9, temp10,
       temp11, temp12, temp13, temp14, temp15,
       temp16, temp17, temp18, temp19, temp20,
       temp21, temp22, temp23, temp24, temp25,
       temp26, temp27, temp28, temp29, temp30,
       temp31, temp32;
15 {
16   fork
17     fork
18       T1_0up = true;
19     par
20       loop0:
21       temp3 = <seq(pre(T1reg), T1_2)>;
22       temp2 = <combine("OR", temp3, T1_0up)>;
23       temp1 = <combine("OR", temp2, T1_1up)>;
24       temp5 = <seq(pre(P1reg), P1_1)>;
25       temp4 = <combine("OR", temp5, P1_0up)>;
26       temp0 = temp1 | temp4;
27       if temp0 then
28         if R1 then
29           G1_0up = true;
30         else
31           P2_0up = true;
32         end;
33       end;
34       pause;
35       goto loop0;
36     par
37       loop6:
38       temp8 = <seq(pre(T1reg), T1_2)>;
39       temp7 = <combine("OR", temp8, T1_0up)>;
40       temp6 = <combine("OR", temp7, T1_1up)>;
41       if temp6 then
42         pause;
43         T2_0up = true;
44       else
45         pause;
46       end;
47       goto loop6;
48     par
49       loop7:
50       temp11 = <seq(pre(T2reg), T2_1)>;
51       temp10 = <combine("OR", temp11, T2_0up)>;
52       temp13 = <seq(pre(P2reg), P2_1)>;
53       temp12 = <combine("OR", temp13, P2_0up)>;
54       temp9 = temp10 | temp12;
55       if temp9 then
56         if R2 then
```

Listing 6.4. (1/2) The `TokenRingArbiter` in SSA form with dismantled expressions

```
57        G2_0up = true;
58      else
59        P3_0up = true;
60      end;
61    end;
62    pause;
63    goto loop7;
64  par
65    loop13:
66    temp15 = <seq(pre(T2reg), T2_1)>;
67    temp14 = <combine("OR", temp15, T2_0up)>;
68    if temp14 then
69      pause;
70      T3_0up = true;
71    else
72      pause;
73    end;
74    goto loop13;
75  par
76    loop14:
77    temp18 = <seq(pre(T3reg), T3_1)>;
78    temp17 = <combine("OR", temp18, T3_0up)>;
79    temp20 = <seq(pre(P3reg), P3_1)>;
80    temp19 = <combine("OR", temp20, P3_0up)>;
81    temp16 = temp17 | temp19;
82    if temp16 then
83      if R3 then
84        G3_0up = true;
85      else
86        P1_0up = true;
87      end;
88    end;
89    pause;
90    goto loop14;
91  par
92    loop20:
93    temp22 = <seq(pre(T3reg), T3_1)>;
94    temp21 = <combine("OR", temp22, T3_0up)>;
95    if temp21 then
96      pause;
97      T1_1up = true;
98    else
99      pause;
100   end;
101   goto loop20;
```

```
102 par
103   loop21:
104   G1_1 = false;
105   G2_1 = false;
106   G3_1 = false;
107   P1_1 = false;
108   P2_1 = false;
109   P3_1 = false;
110   T1_2 = false;
111   T2_1 = false;
112   T3_1 = false;
113   pause;
114   goto loop21;
115 join;
116 term = true;
117 par
118   loop22:
119   temp23 = <seq(pre(G1reg), G1_1)>;
120   G1reg = <combine("OR", temp23, G1_0up)>;
121   temp24 = <seq(pre(G2reg), G2_1)>;
122   G2reg = <combine("OR", temp24, G2_0up)>;
123   temp25 = <seq(pre(G3reg), G3_1)>;
124   G3reg = <combine("OR", temp25, G3_0up)>;
125   temp26 = <seq(pre(P1reg), P1_1)>;
126   P1reg = <combine("OR", temp26, P1_0up)>;
127   temp27 = <seq(pre(P2reg), P2_1)>;
128   P2reg = <combine("OR", temp27, P2_0up)>;
129   temp28 = <seq(pre(P3reg), P3_1)>;
130   P3reg = <combine("OR", temp28, P3_0up)>;
131   temp30 = <seq(pre(T1reg), T1_2)>;
132   temp29 = <combine("OR", temp30, T1_0up)>;
133   T1reg = <combine("OR", temp29, T1_1up)>;
134   temp31 = <seq(pre(T2reg), T2_1)>;
135   T2reg = <combine("OR", temp31, T2_0up)>;
136   temp32 = <seq(pre(T3reg), T3_1)>;
137   T3reg = <combine("OR", temp32, T3_0up)>;
138   G1 = G1reg;
139   G2 = G2reg;
140   G3 = G3reg;
141   if ! term then
142     pause;
143     goto loop22;
144   end;
145 join;
146 }
```

Listing 6.4 (2/2) The `TokenRingArbiter` in SSA form with dismantled expressions

Analyzing the reason for this result reveals that the static cycle could not be dynamically resolved because the dismantling of conditional expressions and their translation into Esterel introduced additional constraints to the analysis. The static cycle is formed by the test for an already present or passed token in each station. In Listing 6.3, Station 1 performs this test in line 10. Listing 6.5a also shows this line to illustrate the problem. Listing 6.5b presents the line in SSA form. Since both `T1` and `P3` have multiple assignments in the program, they require merge expressions. According to the rules for translating composed expressions, presented in Section 4.3.3, the expression is dismantled before each part is translated into Esterel. Listing 6.5c shows the dismantled conditional expression of intermediate SSA code present in lines 21 to 27 of Listing 6.4. This dismantling places all sub-expression sequentially before the their surrounding expression and before the conditional. Consequently, both merge expression must be completely evaluated before the `or` compares the two results. Since the P_i variables form the static cycle, their value cannot be constructively determined at this point using the Esterel analysis. However, the `T` variable normally allows to dynamically break this cycle. The analysis uses three valued function evaluation and allows short-circuit evaluation of expression such as `or`. Hence, even if the value for `P` is unknown, the `or` expression could be constructively evaluated to true if `T` is true. This is the reason why the Token Ring Arbiter in Esterel is constructive. However, the dismantling explicitly demands that all sub-expression must be evaluated to a known value sequentially before the `or` can be evaluated. Thus, the concept of translating programs into Esterel as described here limits the capabilities of the constructiveness analysis in Esterel. The following section describes such limitations more generally.

```
1 if T1 | P1 then
```

(a) Conditional in the original program

```
1 if combine("OR", combine("OR", seq(pre(T1reg), T1_2), T1_0up), T1_1up) | combine("OR", seq(
      pre(P1reg), P1_1), P1_0up) then
```

(b) Conditional in SSA form

```
1 temp3 = <seq(pre(T1reg), T1_2)>;
2 temp2 = <combine("OR", temp3, T1_0up)>;
3 temp1 = <combine("OR", temp2, T1_1up)>;
4 temp5 = <seq(pre(P1reg), P1_1)>;
5 temp4 = <combine("OR", temp5, P1_0up)>;
6 temp0 = temp1 | temp4;
7 if temp0 then
```

(c) Conditional in SSA form with dismantled expression

Listing 6.5. Conditional expression in the first station of the `TokenRingArbiter` program

$A \vee B$		B		
		F	U	T
A	F	F	F	F
	U	F	U	U
	T	F	U	T

(a) AND

$A \vee B$		B		
		F	U	T
A	F	F	U	T
	U	U	U	T
	T	T	T	T

(b) OR

A	$\neg A$
F	T
U	U
T	F

(c) NOT

Table 6.1. Truth tables for boolean `and`, `or`, and `not` operations

6.3 Limitations

The presented concept for the SSA form and translation into Esterel explicitly mentions restrictions of the supported programs, presented in Section 6.1. Moreover, this concept causes further limitations to the programs tested for their constructiveness in Esterel.

6.3.1 Short-Circuit Evaluation

The example of the Token Ring Arbiter illustrates that the current rules for dismantling expressions and their translation into Esterel restricts the analysis capabilities. The constructiveness analysis in Esterel uses three valued logic to determine a known value for each output wire. This allows short-circuit evaluation of boolean logical expressions even if one part is still unknown. Table 6.1 shows the truth tables for basic operations in this logic. However, if sub-expressions are present in an expression, the current rules define that they are dismantled, illustrated in Listing 6.5 for the `TokenRingAbiter`. This is done because merge expressions, which may be contained in other expressions, are defined using nested present tests and emit for example the error signal. Ordering these sub-expressions sequentially before the surrounding expression forces the Esterel constructiveness analysis to determine these sub-expressions before the surrounding expression is evaluated. Hence, it prevents the short-circuit evaluation. Section 7.2.4 presents ideas for a translations allowing such short-circuit evaluation in composed expressions.

6.3.2 Ineffective Writes

Another aspect concerning the Esterel constructiveness analysis are ineffective writes. In Esterel a signal is present iff it is emitted and its state is globally consistent along the entire tick. Hence, even if a signal is emitted multiple times in the same tick, its state is fixed if the analysis determines that some first emit is executed. Listing 6.6a shows the `IneffectiveWrite` program in Esterel. The signal `x` is emitted in the first statement then two thread start. The first thread emits `y` if `x` is present and the second emits `x` if `y` is present. Hence, both threads form a static causal cycle with `x` and `y`. However, since `x` is always present due to the initial emit, the presence test for `x` can be evaluated in the Esterel constructiveness analysis and the cycle dissolves. Consequently, the program is considered constructive in the sense of Esterel.

```
module IneffectiveWrite:
signal x, y in
emit x;
[
  present x then
    emit y
  end
||
  present y then
    emit x
  end
]
end signal
end module
```

(a) Written in Esterel

```
module IneffectiveWrite
int x, y;
{
  x = 1;
  fork
    if x then
      y = 1
    end
  par
    if y then
      x = 1
    end
  join
}
```

(b) Written in SCL

```
module IneffectiveWrite-SSA
int x0, x1, y;
{
  x0 = 1;
  fork
    if seq(x0, x1) then
      y = 1
    end
  par
    if y then
      x1 = 1
    end
  join
}
```

(c) Written in SCL and transformed into SSA form

Listing 6.6. The `IneffectiveWrite` program

Listing 6.6b shows the `IneffectiveWrite` program implemented in SCL. The same behavior is modeled using integers and should yield the same result as the Esterel programs with respect to the variable values. Listing 6.6c illustrates the program in SSA form. Since both definitions of x reach the conditional statement, a *seq*-function is introduced to merge them. According to the definition of the *seq*-function, first x_1 is checked whether it is active. If it is active, then its value will be returned, otherwise x_0 will be checked and returned. However, this first check of the presence signal of x_1 causes this program to be considered non-constructive in Esterel, because it forms a causality cycle with y. The essence of this problem is that a merge expressions may check the presence of all reaching definitions before it yields a result, independent from the actual value of these definitions. In this case the assignment to x_1 does not change the value available from x_0, but is still checked leading to a rejection of the program as non-constructive in Esterel. Translating this program into Esterel using the provided implementation and performing the Esterel constructiveness analysis confirms this. Section 7.2.4 presents a possible approach to solve this problem.

Chapter 7

Conclusion

This final chapter summarizes the presented approach for detecting Esterel constructiveness in SC programs in Section 7.1. Section 7.2 closes with ideas for extending and continuing this approach in possible future work.

7.1 Summary

This thesis presented the concept of Strict Sequential Constructiveness and described a practical approach to detect programs of this class. The approach includes the translation of SCL programs into semantically equivalent Esterel programs and a check of their constructiveness by performing an analysis using an Esterel compiler which implies whether the source program is considered SSC.
First of all, an SSA form is used to transform the sequential and concurrent variable accesses into an Esterel compatible form according to the SC MoC. Afterwards, the program can be translated into Esterel, encoding shared variables by signals. Then a subsequent constructiveness analysis with an Esterel compiler yields the result whether an Esterel program is considered constructive in the sense of Esterel. SC program which form constructive Esterel programs are classified as SSC. This provides a physical foundation for Strict Sequential Constructiveness based on the constructive circuit semantics of Esterel.
This concept was partially implemented in the context of the KIELER project, providing a transformation based on SCGs. This translation from SCGs into Esterel not only allows to check the Esterel constructiveness, but also closes a gap between Esterel and SC languages. It allows to utilize an Esterel compiler for code generation of SC programs. Especially in case of SCEst it provides the possibility of compiling this extended form of Esterel back into pure Esterel.
The evaluation showed that the presented translation has restrictions limiting the constructiveness analysis in Esterel, but is capable of correctly accepting or rejecting some SC programs. However, the implementation and evaluation did not cover the complete range possible SC classes, consequently this requires further extensive testing and future work.

7.2 Future Work

The main focus of this thesis is to develop an SSA form for the SC MoC to detect SSC programs. In the limited time frame, not all arisen ideas and concepts could be evaluated or pursued. Hence, a few ideas for future development and enhancements are given here.

7.2.1 Compiler Advancement

The approach of utilizing an Esterel compiler for analyzing the constructiveness of an SC program is only one possible solution. It is also possible to implement a Must/Can analysis, similar to the constructive behavioral semantics of Esterel, direly for SCGs. However, the usage of variables may require to handle large value ranges when analyzing all possible input combinations. Another approach for programs restricted to boolean variables is an analysis based on circuits. The current compile chain provided for the SC MoC supports hardware synthesis, mentioned in Section 3.2.1. Such a boolean circuit could be used to apply Malik's procedure and determine the constructiveness of the circuit.
Another advancement of the currently provided compilation chain could be the compilation of cyclic programs using the dataflow approach which is typically restricted to acyclic programs. However, Lukoschus presents the removal of cycles in Esterel programs [Luk06]. The requirement for his procedure is Esterel constructiveness. Hence, an adaptation of this approach for SSC programs could also remove cycles in these programs and enable them for the acyclic compilation approach.
Furthermore, the presence of a complete SSA form for SC programs, especially SCGs, allows the implementation of compiler optimizations based on this form. Since the initial motivations for SSA were code optimization techniques for compilers, it facilitates many optimizations. Examples are global value numbering, register allocation, dead code elimination, and constant propagation with conditional branches [WZ91]. Lee et al. show that in the presence of explicit concurrency, a classical algorithm for sparse conditional constant propagation can be extended to handle such programs [LMP98]. Such an adapted algorithm could enhance the current SC compiler and improve the quality of compilation results with respect to code size and execution time.

7.2.2 Sequential Optimization

The SC-specific SSA form conservatively introduces merge expressions on every read access to a variable. Even if these expressions are reduced using domination relations, they are more extensively added than the minimal placement of ϕ-functions in regular SSA. As Section 4.2.1 describes, the regular SSA form requires additional adjustment to fit into the SC domain. However, despite these additions, the regular SSA form might be capable of introducing fewer merge functions because it facilitates the sequential re-usage of the results of merged reaching definitions. Hence, a mixture of the concepts of regular SSA with the SC-specific form to enable correct handling of concurrency could create a more efficient form of SSA for SC programs.
A promising starting point could be the *seq*-function. Since both the *seq*-function and the ϕ-function handle the merge of definitions with a sequential relation, it might be possible to appropriately replace *seq*- by ϕ-functions. This especially when the ϕ-functions is further replaced by an assignment to the merged variable in each incoming branch, as described in Section 4.2.1. This form also complies with a single emission in Esterel under restriction to mutual exclusive branches and non-instantaneous non-schizophrenic loops.

```
1  module NoStaticSchedule
2  input bool i;
3  int x, y;
4  {
5    fork
6      if i then
7        x = 0
8      end
9    par
10     if !i then
11       x = 1
12     end
13   par
14     x = x + 1;
15     y = x
16   join
17 }
```

Listing 7.1. The `NoStaticSchedule` program

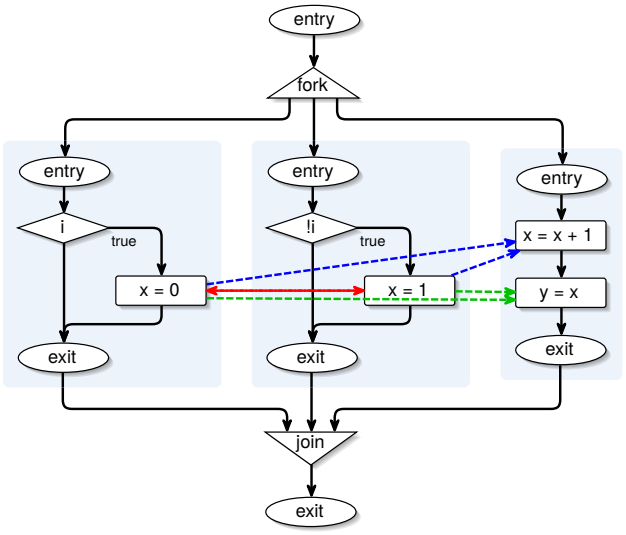


Figure 7.1. SCG representation of the `NoStaticSchedule` program with dependencies

7.2.3 Reducing Restrictions

Section 6.1 evaluates the capabilities of the SC-specific SSA form. The presented approaches for handling the different aspects and structures often introduce restrictions to enable or facilitate the presented solutions. To increase the number of supported programs and further expand the capabilities of this SSA form, it is necessary to overcome these restrictions.

Non-Static Update Handling

The approach for handling updates, presented in Section 4.2.6, requires a partial static schedule to create merge expressions for variables which are used in updates. Consequently, if a write-write dependency exists in the program, no static schedule is possible even if the actual conflict never occurs. Listing 7.1 shows the `NoStaticSchedule` programs which contains two concurrent initialization of `x` depending on `i`. The assignment in line 7 is executed if `i` is true and the one in line 11 if `i` is false. Since `i` is an input and never assigned in the program, the assignments are mutually exclusive and they cannot conflict. Additionally, there is a concurrent update and read access on `x`. Figure 7.1 depicts the SCG representation of the program including dependencies. Due to the static write-write dependency, the SCG is considered non-ASC and the current scheduling approach cannot generate a static schedule. Hence, no merge expression can be created and the programs needs to be rejected by the current approach. A more advanced static scheduling analysis could detect that the two initialization cannot conflict, but testing all possible situations would be very similar to performing an Esterel constructiveness analysis.

However, regarding the definition of the merge functions there is a trivial merge expression for the assignment in line 15:

$$\mathsf{y} = combine(+, conc(\mathsf{x}_0, \mathsf{x}_1), \mathsf{x}_{2up})$$

One possible approach to create such an expression would be the assumption that no conflict can occur. Then the static scheduler could ignore the write-write dependency and create a schedule. Consequently, the resulting merge expression needs to be post-processed to include *conc*-functions to assure the actual confluence or reject the program dynamically in case of a conflict.

A similar process could also handle potential conflicts between updates with different combination functions. The merge expression might not be extended by *conc*-functions, because the updates are handled by *combine*-functions. However, in addition to the merge expression a dynamic assertion could be introduced in the program checking the two updates for their confluence if both are present in the same tick. The implementation would be similar to the *conc*-function but without returning the selected value.

Instantaneous Loops

Another restriction introduced in Section 4.2.5 is the exclusion of instantaneous loops. Moreover, the reordering requires a dominant pause which is executed in each iteration. A more advanced analysis could also reorder the statements for programs with delayed loops but without such a pause. Independent form this reordering, Esterel prohibits instantaneous loops. To support instantaneous loops which are generally allowed in the SC MoC, they could be statically unrolled if they are finitely bounded. This way, the backward jump would be eliminated and the SSA form would rename and order the assignments sequentially. The only requirement for this procedure is that a finite bound for each instantaneous loop can be statically determined. This is not possible in general. Another problem is that the program size will significantly grow when unrolling loops.

However, assuming some analyses detects that the loop present in Listing 7.2a will always be executed five times, then it is possible to fully unroll this loop. Listing 7.2b shows the result of such an unrolling. The loop structure is removed and the loop body is pasted five times into the program.

In general, an exact number of iterations is unusual for loops, but sometimes a maximum can be determined. The unrolling would be similar, but each of the unrolled iterations would be guarded by the loop condition, such that the calculations would be skipped if the original loop would have been exited.

In addition to the use case of constructiveness in Esterel, also the dataflow compilation approach could benefit from an unrolling of instantaneous loops to support more input programs.

```
1  module LoopUnrolling
2  int x, y;
3  {
4    x = 0;
5    y = 5;
6    LoopBegin:
7    x = x + 1;
8    if x < y then
9      goto LoopBegin
10   end
11 }
```

(a) Original

```
1  module LoopUnrolling
2  int x, y;
3  {
4    x = 0;
5    y = 5;
6    x = x + 1;
7    x = x + 1;
8    x = x + 1;
9    x = x + 1;
10   x = x + 1;
11 }
```

(b) With unrolled loop

Listing 7.2. The `LoopUnrolling` program

Structural Restrictions

Another restriction is made on forward jumps. The SSA form and translation into Esterel only handle forward jumps which are equivalent to if-then-else structures. This restriction is made to facilitate the analysis and translation of the programs. A solution for translating forward jumps is proposed by Rathlev in his future work section [Rat15]. However, his approach includes code duplication when jumps target into nested structures. Section 4.3.1 presents a program with such a structure. Incorporating the proposed approach into the concept of this thesis should consider possible effects of the code duplication on the SSA form and the analysis in the SSA transformation.

7.2.4 Enhancing the Constructiveness Analysis

Section 6.3 presents some limitation caused by the presented SSA definition and translation into Esterel. This section presents ideas how the presented concept can be extended to enhance the constructiveness analysis in Esterel. This could allow to accept more programs which are constructive in the sense of Esterel, but are rejected due to limitations introduced by the concept presented here.

Short-Circuit Evaluation

The problem of inhibiting short-circuit evaluation in expressions is caused by the sequential dismantling such expressions. Consequently, a possible approach could be to not dismantle them. For conditionals, the expression could be translated into a single boolean expression. However, this requires to encode merge expressions into boolean signal expression with the same merging behavior. In the case of *conc*-function this includes the possible emission of the error signal, which is not possible in a signal expression in Esterel. Hence, some additional code could be inserted outside the presence test, which asserts that no conflict occurs or the error is emitted.

The conditional expression in Listing 6.5 on page 91 could be represented by the following logical signal expression in Esterel.

$$
\begin{aligned}
temp_3: &\quad ((\texttt{T1}_2^p \texttt{ and } \texttt{T1}_2) \texttt{ or } (\texttt{not } \texttt{T1}_2^p \texttt{ and } (\texttt{T1}_{reg}^p \texttt{ and } \texttt{T1}_{reg})))\\
temp_2: &\quad ((\texttt{T1}_{0up}^p \texttt{ and } \texttt{T1}_{0up}) \texttt{ or } temp_3)\\
temp_1: &\quad ((\texttt{T1}_{1up}^p \texttt{ and } \texttt{T1}_{1up}) \texttt{ or } temp_2)\\
temp_5: &\quad (\texttt{P1}_1^p \texttt{ and } \texttt{P1}_1) \texttt{ or } (\texttt{not } \texttt{P1}_1^p \texttt{ and} (\texttt{P1}_{reg}^p \texttt{ and } \texttt{P1}_{reg})))\\
temp_4: &\quad ((\texttt{P1}_{0up}^p \texttt{ and } \texttt{P1}_{0up}) \texttt{ or } temp_5)\\
temp_0: &\quad (temp_1 \texttt{ or } temp_4)
\end{aligned}
$$

$$\Downarrow$$

$$
\begin{aligned}
&(((\texttt{T1}_{1up}^p \texttt{ and } \texttt{T1}_{1up}) \texttt{ or } ((\texttt{T1}_{0up}^p \texttt{ and } \texttt{T1}_{0up}) \texttt{ or } ((\texttt{T1}_2^p \texttt{ and } \texttt{T1}_2) \texttt{ or } (\texttt{not } \texttt{T1}_2^p \texttt{ and } (\texttt{T1}_{reg}^p \texttt{ and } \texttt{T1}_{reg})))))\\
&\qquad \texttt{or } ((\texttt{P1}_{0up}^p \texttt{ and } \texttt{P1}_{0up}) \texttt{ or } (\texttt{P1}_1^p \texttt{ and } \texttt{P1}_1) \texttt{ or } (\texttt{not } \texttt{P1}_1^p \texttt{ and} (\texttt{P1}_{reg}^p \texttt{ and } \texttt{P1}_{reg})))))
\end{aligned}
$$

The upper part shows the expressions for each of the intermediate variables in Listing 6.5c. The lower part presents the composition of these sub-expressions into one single expression, which can be used in a presence test to represent the `if` statement in Listing 6.5b. Note that this expression assumes that some value is always present to read and does not check the variables for uninitialized reads, because this would involve an emission of the error signal. This approach also allows to translate assignments, for example using merge expressions, with fewer presence tests. Listing 7.3a shows the assignment of $\texttt{O2}_{reg}$ in line 33 of the ABO program in SSA form from Listing 6.1 on page 85. Listing 7.3b illustrates an optimized translation into Esterel. The merge expression is represented by two presence test, merging all definitions of O2. The use of boolean logic for the entire merge expressions without dismantling the expression, allows the Esterel constructiveness analysis to perform short-circuit evaluation for this merge expression.

```
1 O2reg = seq(seq(pre(O2reg), O2_0), O2_1)
```

(a) Original

```
1 present O2_1p or O2_0p or pre(O2regp) then
2   emit O2regp;
3   present (O2_1p and O2_1) or
4             (not O2_1p and ((O2_0p and O2_0) or
5               (not O2_0p and pre(O2_reg))))
6    emit O2reg
7   end
8 end
```

(b) Translated into Esterel using optimized present tests

Listing 7.3. Assignment of $\texttt{O2}_{reg}$ in the ABO program

Expression	Esterel

`v ← x_i and x_j`	`present (x_i^p and not x_i) or (x_j^p and not x_j) then`
	`  emit v^p;`
	`else`
	`  present (x_i^p and x_i) and (x_j^p and x_j) then`
	`  emit v^p;`
	`  emit v`
	`  end`
	`end`

`v ← x_i or x_j`	`present (x_i^p and x_i) or (x_j^p and x_j) then`
	`  emit v^p;`
	`  emit v`
	`else`
	`  present x_i or x_j then`
	`  emit v^p`
	`  end`
	`end`

Table 7.1. Possible translation pattern for non-strict operator expressions using pure signals

Another approach could be the elimination of the sequential ordering between the dismantled sub-expressions. For example by moving the evaluation of each sub-expressions in its own thread, parallel to the original program. When the controlflow of the program reaches a present test with a dismantled expression it activates the calculation of each sub-expression. Since the evaluation is performed concurrently, the sub-expressions can constructively perform their operations if the necessary values have reached a known state. Hence, only data-dependencies restrict the Esterel constructive analysis of each sub-expression and no explicit sequential ordering. However, such an approach should assure that the concurrent threads are able to terminate correctly and do not raise new restrictions to the Esterel constructiveness analysis. Furthermore, the definitions of the Esterel operator expressions using augmented signals should be non-strict, to allow short-circuit evaluation. Table 7.1 presents possible translation patterns for a non-strict `and` and `or` operation.

Ineffective Writes

Ineffective writes are not taken into account when executing merge functions. Each reaching definition is checked if it can affect the merged value disregarding whether it actually changes the value. This testing of the presence of signals may introduce causality cycles which are unnecessary if the assignments performing the ineffective write could be ignored. In the case of signals the state is either absent or present and cannot change from present to absent in the same tick. In contrast to that, variables can be assigned to any valid value. Moreover, the value can be the result of an arbitrarily complex expression. Hence, in general it is not possible to

determine whether an assignment is ineffective. However, in the case of constant assignments a constant propagation can detect ineffective writes. For example in the `IneffectiveWrite` program in Listing 6.6c on page 93, both definitions of `x` are constant. Since both are written to true the *seq*-function can be replaced in the following way.

$$seq(\mathtt{x}_0, \mathtt{x}_1) \rightarrow \mathtt{x}_0^p \texttt{ or } \mathtt{x}_1^p$$

The same holds for *conc*-functions which are in this case no longer required to check for a conflict. In combination with the short-circuit evaluation of the used `or` expressions this would allow the Esterel constructiveness analysis to accept the translated program and results in a classification as SSC. Furthermore, this approach would be more permissive than Esterel, because it allows reads before writes.

Bibliography

[And03] Charles André. *Semantics of SyncCharts*. Tech. rep. ISRN I3S/RR–2003–24–FR. Sophia-Antipolis, France: I3S Laboratory, Apr. 2003.

[And96] Charles André. *SyncCharts: A visual representation of reactive behaviors*. Tech. rep. RR 95–52, rev. RR 96–56. Sophia-Antipolis, France: I3S, Rev. April 1996.

[AP02] Andrew W. Appel and Jens Palsberg. *Modern compiler implementation in java, 2nd edition*. Cambridge University Press, 2002. ISBN: 0-521-82060-X.

[AWZ88] B. Alpern, M. N. Wegman, and F. K. Zadeck. "Detecting equality of variables in programs". In: *Proceedings of the 15th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. POPL '88. San Diego, California, USA: ACM, 1988, pp. 1–11. ISBN: 0-89791-252-7. DOI: 10.1145/73560.73561. URL: http://doi.acm.org/10.1145/73560.73561.

[BC84] Gérard Berry and Laurent Cosserat. "The ESTEREL Synchronous Programming Language and its Mathematical Semantics". In: *Seminar on Concurrency, Carnegie-Mellon University*. Vol. 197. LNCS. Springer-Verlag, 1984, pp. 389–448. ISBN: 3-540-15670-4.

[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*. Vol. 91. Piscataway, NJ, USA: IEEE, Jan. 2003, pp. 64–83.

[Ber00] Gérard Berry. *The Esterel v5 language primer, version v5_91*. ftp://ftp-sop.inria.fr/esterel/pub/papers/primer.pdf. Centre de Mathématiques Appliquées Ecole des Mines and INRIA. 06565 Sophia-Antipolis, 2000.

[Ber02] Gérard Berry. *The constructive semantics of pure Esterel*. Centre de Mathématiques Appliqées, Ecole des Mines de Paris and INRIA, 2004 route des Lucioles, 06902 Sophia-Antipolis CDX, France: Draft Book, Version 3.0, Dec. 2002.

[Ber99] Gérard Berry. *The Esterel v5 language primer*. ftp://ftp-sop.inria.fr/meije/esterel/papers/primer.ps. 1999.

[BG92] Gérard Berry and Georges Gonthier. "The Esterel synchronous programming language: Design, semantics, implementation". In: *Science of Computer Programming* 19.2 (1992), pp. 87–152.

[BGM+09] Loïc Besnard, Thierry Gautier, Matthieu Moy, Jean-Pierre Talpin, Kenneth Johnson, and Florence Maraninchi. "Automatic translation of C/C++ parallel code into synchronous formalism using an SSA intermediate form". In: *Electronic Communications of the EASST* 23 (2009).

[Bou98] Frédéric Boussinot. *SugarCubes implementation of causality*. Research Report RR-3487. INRIA, Sept. 1998.

[BS91] Frédéric Boussinot and Robert de Simone. "The ESTEREL language. another look at real time programming". In: *Proceedings of the IEEE* 79.9 (Sept. 1991), pp. 1293–1304.

[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". In: *ACM Transactions on Programming Languages and Systems* 13.4 (Oct. 1991), pp. 451–490.

[Edw05] Stephen A. Edwards. "The challenges of hardware synthesis from C-like languages". In: *Proceedings of the conference on Design, Automation and Test in Europe - Volume 1*. DATE '05. Washington, DC, USA: IEEE Computer Society, 2005, pp. 66–67. ISBN: 0-7695-2288-2. DOI: 10.1109/DATE.2005.307. URL: http://dx.doi.org/10.1109/DATE.2005.307.

[Fuh11] Hauke Fuhrmann. "On the pragmatics of graphical modeling". Dissertation. Kiel: Christian-Albrechts-Universität zu Kiel, Faculty of Engineering, 2011.

[GGB+91] Paul Le Guernic, Thierry Goutier, Michel Le Borgne, and Claude Le Maire. "Programming real time applications with SIGNAL". In: *Proceedings of the IEEE* 79.9 (Sept. 1991), pp. 1321–1336.

[Har87] David Harel. "Statecharts: A visual formalism for complex systems". In: *Science of Computer Programming* 8.3 (June 1987), pp. 231–274.

[HCR+91] Nicolas Halbwachs, Paul Caspi, Pascal Raymond, and Daniel Pilaud. "The synchronous data-flow programming language LUSTRE". In: *Proceedings of the IEEE* 79.9 (Sept. 1991), pp. 1305–1320.

[HDM+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: ACM, June 2014.

[HMA+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. ISSN 2192-6247. Christian-Albrechts-Universität zu Kiel, Department of Computer Science, Aug. 2013.

[HMA+14] 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". In: *ACM Transactions on Embedded Computing Systems, Special Issue on Applications of Concurrency to System Design* 13.4s (July 2014), 144:1–144:26.

[HP85] David Harel and Amir Pnueli. "On the development of reactive systems". In: *Logics and models of concurrent systems* (1985), pp. 477–498.

[HU72] Matthew S. Hecht and Jeffrey D. Ullman. "Flow graph reducibility". In: *SIAM J. Comput.* 1.2 (1972), pp. 188–202. DOI: 10.1137/0201014. URL: http://dx.doi.org/10.1137/0201014.

[HU74] Matthew S. Hecht and Jeffrey D. Ullman. "Characterizations of reducible flow graphs". In: *J. ACM* 21.3 (1974), pp. 367–375. DOI: 10.1145/321832.321835. URL: http://doi.acm.org/10.1145/321832.321835.

[Joh13] Gunnar Johannsen. "Hardwaresynthese aus SCCharts". http://rtsys.informatik.uni-kiel.de/~biblio/downloads/theses/gjo-mt.pdf. Master thesis. Kiel University, Department of Computer Science, Oct. 2013.

[KTB+06] Hamoudi Kalla, Jean-Pierre Talpin, David Berner, and Loïc Besnard. "Automated translation of C/C++ models into a synchronous formalism". In: *13th Annual IEEE International Symposium and Workshop on Engineering of Computer-Based Systems (ECBS'06)*. Mar. 2006, pages. DOI: 10.1109/ECBS.2006.27.

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

[Lee06] Edward A. Lee. "The problem with threads". In: *IEEE Computer* 39.5 (2006), pp. 33–42.

[Lee99] Jaejin Lee. "Compilation Techniques for Explicitly Parallel Programs". PhD thesis. University of Illinois at Urbana-Champaign, Oct. 1999. URL: http://www.cse.msu.edu/~jlee/Papers/thesis.pdf.

[LMP98] Jaejin Lee, Samuel P. Midkiff, and David A. Padua. "Concurrent static single assignment form and constant propagation for explicitly parallel programs". In: *Languages and Compilers for Parallel Computing: 10th International Workshop, LCPC'97 Minneapolis, Minnesota, USA, August 7–9, 1997 Proceedings*. 1998, pp. 114–130. ISBN: 978-3-540-69788-6. DOI: 10.1007/BFb0032687. URL: http://dx.doi.org/10.1007/BFb0032687.

[LT79] Thomas Lengauer and Robert Endre Tarjan. "A fast algorithm for finding dominators in a flowgraph". In: *ACM Transactions on Programming Languages and Systems* 1.1 (1979), pp. 121–141. DOI: 10.1145/357062.357071. URL: http://doi.acm.org/10.1145/357062.357071.

[Luk06] Jan Lukoschus. “Removing cycles in Esterel programs”. http://eldiss.uni-kiel.de/macau/receive/dissertation_diss_2015. PhD thesis. Christian-Albrechts-Universität zu Kiel, Faculty of Engineering, July 2006.

[Mal94] Sharad Malik. “Analysis of cyclic combinational circuits”. In: *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 13.7 (July 1994), pp. 950–956.

[MG97] Giovanni De Micheli and Rajesh K. Gupta. “Hardware/software co-design”. In: *Proceedings of the IEEE* 85.3 (Mar. 1997), pp. 349–365.

[MHH13] Christian Motika, Reinhard von Hanxleden, and Mirko Heinold. “Programming deterministice reactive systems with Synchronous Java (invited paper)”. In: *Proceedings of the 9th Workshop on Software Technologies for Future Embedded and Ubiquitous Systems (SEUS 2013)*. IEEE Proceedings. Paderborn, Germany, 17/18 06 2013.

[MSH14] 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)*. Vol. 8802. LNCS. Corfu, Greece, Oct. 2014, pp. 443–462. DOI: 10.1007/978-3-662-45234-9.

[Nas15] Stanislaw Nasin. “Transformaion from sccharts to esterel”. http://rtsys.informatik.uni-kiel.de/~biblio/downloads/theses/sna-mt.pdf. Master Thesis. Kiel University, Department of Computer Science, Oct. 2015.

[Pan02] Paritosh Pandya. “The saga of synchronous bus arbiter: on model checking quantitative timing properties of synchronous programs”. In: *Electronic Notes in Theoretical Computer Science*. Ed. by Florence Maraninchi, Alain Girault, and Éric Rutten. Vol. 65. Elsevier, 2002.

[PEB07] Dumitru Potop-Butucaru, Stephen A. Edwards, and Gérard Berry. *Compiling Esterel*. Springer, May 2007.

[Plo81] Gordon D. Plotkin. *A Structural Approach to Operational Semantics*. Technical Report DAIMI FN-19. http://homepages.inf.ed.ac.uk/gdp/publications/SOS.ps. University of Aarhus, Denmark, 1981.

[PS91] Amir Pnueli and M. Shalev. “What is in a step: on the semantics of Statecharts”. In: *Proc. Int. Conf. on Theoretical Aspects of Computer Software (TACS’91)*. London, UK: Springer, 1991, pp. 244–264.

[PST05] D. Potop-Butucaru, R. de Simone, and J.-P. Talpin. “The synchronous hypothesis and synchronous languages”. In: *Embedded Systems Handbook*. Ed. by R. Zurawski. CRC Press, 2005.

[Rat15] Karsten Rathlev. “From Esterel to SCL”. http://rtsys.informatik.uni-kiel.de/~biblio/downloads/theses/krat-mt.pdf. Master thesis. Kiel University, Department of Computer Science, Mar. 2015.

[RSM+15] Karsten Rathlev, Steven Smyth, Christian Motika, Reinhard von Hanxleden, and Michael Mendler. "SCEst: Sequentially Constructive Esterel". In: *Proceedings of the 13th ACM-IEEE International Conference on Formal Methods and Models for System Design (MEMOCODE'15)*. Austin, TX, USA, Sept. 2015.

[RSM+16] Francesca Rybicki, Steven Smyth, Christian Motika, Alexander Schulz-Rosengarten, and Reinhard von Hanxleden. "Interactive model-based compilation continued – interactive incremental hardware synthesis for SCCharts". In: *Proceedings of the 7th International Symposium on Leveraging Applications of Formal Methods, Verification and Validation (ISoLA 2016)*. LNCS. Accepted. 2016.

[Rüe11] Ulf Rüegg. "Interactive transformations for visual models". Bachelor Thesis. Kiel University, Department of Computer Science, Mar. 2011.

[RWZ88] B. K. Rosen, M. N. Wegman, and F. K. Zadeck. "Global Value Numbers and Redundant Computations". In: *Proceedings of the 15th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. POPL '88. San Diego, California, USA: ACM, 1988, pp. 12–27. ISBN: 0-89791-252-7. DOI: 10.1145/73560.73562. URL: http://doi.acm.org/10.1145/73560.73562.

[Ryb16] Francesca Rybicki. "Interactive incremental hardware synthesis for SCCharts". http://rtsys.informatik.uni-kiel.de/~biblio/downloads/theses/fry-bt.pdf. Bachelor Thesis. Kiel University, Department of Computer Science, Mar. 2016.

[SBT96] Thomas R. Shiple, Gérard Berry, and Hervé Touati. "Constructive Analysis of Cyclic Circuits". In: *Proc. European Design and Test Conference (ED&TC'96), Paris, France*. Paris, France: IEEE Computer Society Press, Mar. 1996, pp. 328–333.

[Sch10] Klaus Schneider. *The synchronous programming language Quartz*. Internal Report. Kaiserslautern, Germany: Department of Computer Science, University of Kaiserslautern, 2010.

[SHW93] Harini Srinivasan, James Hook, and Michael Wolfe. "Static single assignment form for explicitly parallel programs". In: *In Proceedings of the 20th ACM Symposium on Principles of Programming Languages*. Jan. 1993, pp. 260–272.

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

[SSH13] Christian Schneider, Miro Spönemann, and Reinhard von Hanxleden. "Just model! – Putting automatic synthesis of node-link-diagrams into practice". In: *Proceedings of the IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC'13)*. San Jose, CA, USA, Sept. 2013, pp. 75–82. DOI: 10.1109/VLHCC.2013.6645246.

[Tar04] Olivier Tardieu. "Goto and concurrency—introducing safe jumps in Esterel". In: *Proceedings of Synchronous Languages, Applications, and Programming (SLAP'04)*. Barcelona, Spain, Mar. 2004.

[Tar74] Robert Tarjan. “Finding dominators in directed graphs”. In: *SIAM Journal on Computing* 3.1 (1974), pp. 62–89.

[TS04] Olivier Tardieu and Robert de Simone. “Curing schizophrenia by program rewriting in Esterel”. In: *Proceedings of the Second ACM-IEEE International Conference on Formal Methods and Models for Codesign (MEMOCODE'04)*. San Diego, CA, USA, 2004.

[VJB+07] Peter Vanbroekhoven, Gerda Janssens, Maurice Bruynooghe, and Francky Catthoor. “A practical dynamic single assignment transformation”. In: *ACM Trans. Design Autom. Electr. Syst.* 12.4 (2007). DOI: 10.1145/1278349.1278353. URL: http://doi.acm.org/10.1145/1278349.1278353.

[Wei15] Tibor Weiß. “Von nebenläufigkeit zur parallelität in sccharts”. Bachelor thesis. Kiel University, Department of Computer Science, 2015.

[WZ91] Mark N. Wegman and F. Kenneth Zadeck. “Constant Propagation with Conditional Branches”. In: *ACM Trans. Program. Lang. Syst.* 13.2 (Apr. 1991), pp. 181–210. ISSN: 0164-0925. DOI: 10.1145/103135.103136. URL: http://doi.acm.org/10.1145/103135.103136.

List of Acronyms

ASC	Acyclic Sequentially Constructive
CCFG	concurrent controlflow graph
CFG	controlflow graph
CSSA	Concurrent Static Single Assignment
DSA	Dynamic Single Assignment
DSL	Domain Specific Language
ELK	Eclipse Layout Kernel
EMF	Eclipse Modeling Framework
GCC	GNU Compiler Collection
IDE	Integrated Development Environment
iur	initialize-update-read
KiCo	KIELER Compiler
KIELER	Kiel Integrated Environment for Layout Eclipse Rich Client
KLighD	KIELER Lightweight Diagrams
MoC	Model of Computation
OSGi	Open Services Gateway initiative
RCP	Rich Client Platform
SC	Sequentially Constructive
SCCharts	Sequentially Constructive Statecharts
SCEst	Sequentially Constructive Esterel
SCG	Sequentially Constructive Graph
SCL	Sequentially Constructive Language
SLIC	Single-Pass Language-Driven Incremental Compilation

SSA	Static Single Assignment
SSC	Strictly Sequentially Constructive
SyncCharts	Synchronous Charts
WCET	Worst-Case Execution Time

List of Listings

List of Figures

List of Tables