

Priority-Based Compilation of SCCharts

Lars Peiler

Master's Thesis

2017

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

Safety-critical systems such as airbags or the control systems of a nuclear reactor must be reliable as their correct execution directly influences the safety of surrounding human beings or the mission itself. However, traditional imperative programming languages suffer from non-deterministic behavior such as race conditions, limiting the reliability of programs developed in those languages. *Synchronous languages* overcome these shortcomings and guarantee deterministic behavior.

Traditionally, synchronous languages use *signals* for communication in concurrent execution, which require a globally consistent state. This restriction is lifted by the Sequentially Constructive (SC) Model of Computation (MoC), a set of rules specified for the design of synchronous languages. Synchronous languages further divide the execution of a model into *ticks* – discrete divisions of time that are considered to be instantaneous. KIELER SCCharts, a visual synchronous language, implements this SC MoC using the data-flow approach to generate deterministic code, which requires models to be acyclic to be compilable. This thesis presents the concept and implementation of an alternative low-level compilation approach towards Sequentially Constructive Charts (SCCharts) models. It assigns priorities to determine the scheduling order of statements in the program depending on the order stipulated by the SC MoC. When executing the generated code, the priorities assigned to statements determine dynamically which statement is to be executed next. Due to this approach, cycles that do not contain any concurrent scheduling constraints are allowed unlike the data-flow approach. An optimization is introduced to reduce the number of required priorities as well as the number of priority changes in models. Using this optimization, context switches can be reduced, improving execution times. Additionally, this thesis covers the handling of schizophrenia in the priority-based compilation. Schizophrenia is a problem that occurs in synchronous languages, when statements are executed multiple times per tick. If for example a local scope is left and re-entered, the re-initialization of local variables as well as the constraints for concurrency must be considered. The priority-based compilation solves this by allowing cycles as well as sequential writes to variables as stipulated by the SC MoC.

ASC Acyclic Sequentially Constructive
CFG Control-Flow Graph
CCFG Concurrent Control-Flow Graph
ELK Eclipse Layout Kernel
GRC Graph Code
IASC iur-Acyclic Sequentially Constructive
IDE Integrated Development Environment
KiCo KIELER Compiler
KIELER Kiel Integrated Environment for Layout Eclipse Rich Client
KIML KIELER Infrastructure for Meta Layout
KLighD KIELER Lightweight Diagrams
LCA Least Common Ancestor
LOESS Locally Weighted Smoothing
M2M Model to Model
MoC Model of Computation
npr Node Priority
prioID Priority ID
PRET-C Precision Timed C
RCP Rich Client Platform
SASC Structurally Acyclic Sequentially Constructive
SCC Strongly Connected Component
SCCharts Sequentially Constructive Charts
SCG Sequentially Constructive Graph
SCL Sequentially Constructive Language
SCL_P Sequentially Constructive Language with priorities
SC Sequentially Constructive

SIASC Structurally iur-Acyclic Sequentially Constructive
SJ Synchronous Java
SLIC Single-Pass Language Driven Incremental Compilation
tsID Thread Segment ID
WCET Worst Case Execution Time

Contents

1	Introduction	1
1.1	SCCharts and Sequential Constructiveness	1
1.2	Problem Statement	3
1.3	Outline	4
2	Foundations	7
2.1	Sequential Constructiveness	8
2.1.1	SCL and SCG	8
2.1.2	The SC Language and Priorities	12
2.1.3	Synchronous Java	15
2.1.4	The ABO Example	16
2.2	Used Technologies	17
2.2.1	Eclipse	17
2.2.2	KIELER	18
3	Related Work	21
3.1	Data-flow Low-Level Compilation of SCCharts	21
3.2	Esterel	23
3.3	SyncCharts and SyncCharts in C	25
3.4	PRET-C	27
4	Priority-Based Compilation	29
4.1	Priority-Based Compilation	29
4.1.1	Strongly Connected Components	30
4.1.2	Node Priorities	31
4.1.3	Priority IDs and Optimized Priority IDs	34
4.1.4	C Code Generation	41
4.1.5	Java Code Generation	46
4.2	Optimizing the Node Priority Assignment	48
4.2.1	Proof of Correctness	55
4.2.2	Proof of Improvement	56
4.2.3	Further Improvements	57
4.3	Schizophrenia	58
4.3.1	Schizophrenia in the Priority Based Compilation	59

Contents

5	Implementation	63
5.1	Implementation into KIELER	63
5.1.1	Original Approach	64
5.1.2	Optimization	67
5.1.3	Schizophrenia	69
6	Evaluation	71
6.1	Correctness	71
6.1.1	Dataset	71
6.1.2	Results	72
6.2	Comparison to the Data-flow Low-Level Compilation	72
6.2.1	Datasets	74
6.2.2	Results	76
6.3	Optimization Comparison	83
6.4	Low-Level Implementation Evaluation	84
7	Conclusion	87
7.1	Summary	87
7.2	Future Work	88
7.2.1	Further Optimizations	88
7.2.2	Classification of Compilable Models – Improvements	89
7.2.3	Further Alternative Compilations	90

List of Figures

1.1	Overview of core SCCharts and extended SCCharts statements.	2
1.2	The SCG of a simple cyclical model	4
2.1	Visualization of a reactive system	8
2.2	Example SCG exemplifying dependencies and the iur protocol	10
2.3	Execution states of a thread	11
2.4	The different classes of sequential constructivity of programs	11
2.5	Models representing the different classes of SC	13
2.5	The SCG of ABO	16
2.6	KIELER project overview	18
2.7	Overview of the KIELER Compiler	19
3.1	The SyncChart of ABSWO	26
4.1	The SCG of a simple cyclical model with an immediate loop	30
4.2	The SCG of a simple cyclical model, but without an immediate loop	30
4.3	The SCG of ABO with annotated nprs	33
4.4	The SCG of a model with an incorrect npr assignment	33
4.5	The SCG of a model with an immediate cycle containing dependency edges	35
4.6	The SCG of another model with an immediate cycle containing a dependency edge	35
4.7	The SCG of a model with an unreachable immediate cycle containing dependency edges	35
4.8	The SCG of ABO with annotated prioIDs	40
4.9	The SCCharts of a modified cyclic ABO	41
4.10	The SCG of a modified ABO with a rising prioID	41
4.11	The SCG of a model with problematic join priorities	42
4.12	The SCG of ABO II	44
4.13	SCG model, where the assignment of tsIDs causes an unnecessary context switch.	49
4.14	SCG model with reduced context switches	49
4.15	SCG of a model with reduced prio-statements	50
4.16	Example SCG model where the propagation of nprs cannot reduce all unnecessary context switches.	50
4.17	Example SCG where all prio-statements could be optimized away	53
4.18	Example Sequentially Constructive Graph (SCG) where the thread with the highest entry npr also has the lowest exit npr	53

List of Figures

4.19	Comparison of maximum, minimum, and assigned n_{prs} of threads from Figure 4.17	54
4.20	SCG of an optimized model with remaining unnecessary context switches . . .	55
4.22	The SCG of a simple schizophrenic model with its associated priorities	59
4.23	The SCG of a schizophrenic model with a dependency induced cycle	60
4.24	The SCG of a schizophrenic model with a dependency induced cycle containing nodes of an external thread	60
4.25	The SCG of a schizophrenic model with a dependency induced cycle with further dependencies towards concurrent nodes	61
5.1	Overview of the KIELER compilations enhanced with the priority-based compilation highlighted in red	63
5.2	Overview of the two low-level compilation approaches	64
6.1	SCCharts model that failed during the JUnit test	72
6.2	Example <i>JitterProgram</i> with three sequential concurrent regions	75
6.3	Compilation time of all models in ms	77
6.4	Compilation time of all models with less then 200 nodes in the SCG in ns . . .	77
6.5	Number of generated variables of all models	78
6.6	Difference in compiled variables between the priority-based and data-flow compilation approaches	79
6.7	Size of the compiled C program in bytes of all models	80
6.8	Average execution time of all non-scripted models with traces in ns	81
6.9	Average execution time of all non-scripted models with under 200 nodes in their SCG and traces in ns	81
6.10	Average execution time of all scripted <i>JitterProgram</i> models in ns	82
6.11	Average jitter of all models with traces in ns	83
6.12	Example execution times per tick of the <i>JitterProgram148.sct</i> model exemplifying the jitter	83
6.13	Average execution times of the optimized and non-optimized priority-based compilation approach	84
6.14	Average execution times of the priority-based compilation approach with only array-based implementation compared to scalar-based implementation	85
7.1	The classes of SC, extended by the schizophrenia adaptations of this thesis . .	89
7.2	A schizophrenic model that is still not schedulable	90

List of Tables

1.1	Comparison of data-flow vs. priority based low-level compilation approaches	3
2.1	Overview of SCL and SCG elements.	9
2.2	Overview of SCL, SCL _p and SJ elements	15
3.1	Variable values at the end of the ticks of the small PRET-C and SCL programs. .	27

Listings

1.1	C code of a simple cyclical model	4
2.1	Selected SCL _P macros	14
2.2	Simple example of SJ code	16
2.3	Simple example code showing the difference between Xtend and Java	18
3.1	Generated C code of the ABO example from Figure 2.5 by the data-flow compilation of SCCharts	22
3.2	Schizophrenic model in Esterel	24
3.3	Solution to schizophrenia in Esterel by Tardieu and de Simone	24
3.4	Synchronous C translation of the ABSWO example	26
3.5	A simple PRET-C program	27
3.6	A similar program, implemented in SCL	27
4.1	Generated SCL _P code of the ABO example	44
4.2	Generated SJ code of the ABO example	47
4.3	Old and new implementation of the fork4 macro	55
4.4	Simple schizophrenic Esterel model	57
4.5	Generated C code from a schizophrenic model	59
5.1	Xtend implementation of Tarjan's algorithm	65
5.2	Xtend implementation of original npr calculation	67
5.3	Xtend implementation of the optimized npr assignment	68
5.4	Xtend implementation for breaking up schizophrenic SCCs	69

Introduction

Many computer systems constantly react to their environment. For example, an entertainment system receives button presses by a user and outputs sound and video signals. An airbag reads acceleration values from various sensors and either blows up or not. The control elements of an airplane receive input by the pilot, which they translate into movement of wing elements or the rudder. To deal with a constant stream of inputs from users and multiple sensors at the same time, they often use concurrency as a means to process all these inputs at once. Depending on the application of these *reactive systems*, timing and safety may be of importance. If an airbag triggers too late, too early or not at all, the passenger may suffer injuries or even die.

One difficult aspect of concurrency is *determinism*. According to Lee et al. [Lee06], many widely used approaches to solve concurrency such as *threads* suffer from *race conditions* and other non-deterministic behavior. A system that does not act predictably and does not always react identically to the same input in the same environment is undesirable. Not only would its reaction time be difficult to predict, but if an airbag blows up sometimes but not every time under the same conditions, the safety of the passenger cannot be guaranteed. Therefore, *Synchronous Languages* were developed to deal with deterministic concurrency [BCE+03]. Each synchronous language has a specific set of rules or Model of Computation (MoC) that stipulates whether a program is valid or invalid. Besides deterministic concurrency, other goals of synchronous languages according to Benveniste et al. are simplicity and synchrony. Simplicity allows the programmer to create a model in the cleanest way possible and to easily make formal reasoning of the model. Synchrony on the other hand means that a synchronous language should support simple and frequently used standard models where all actions are assumed to take finite memory and time [BCE+03].

1.1 SCCharts and Sequential Constructiveness

One MoC that adheres to these ideals as described in the previous sections is the Sequentially Constructive (SC) MoC as introduced by von Hanxleden et al. [HMA+14]. To guarantee deterministic concurrency, it dictates that during concurrent execution of statements *writes* to a variable have to be executed before *updates* to the same variable. An update in turn has to be executed before *reads* to this variable. If such a *scheduling* is not possible, the model is rejected. In sequential execution, however, the order of writes and reads to a variable is insignificant as their order does not influence determinism.

1. Introduction

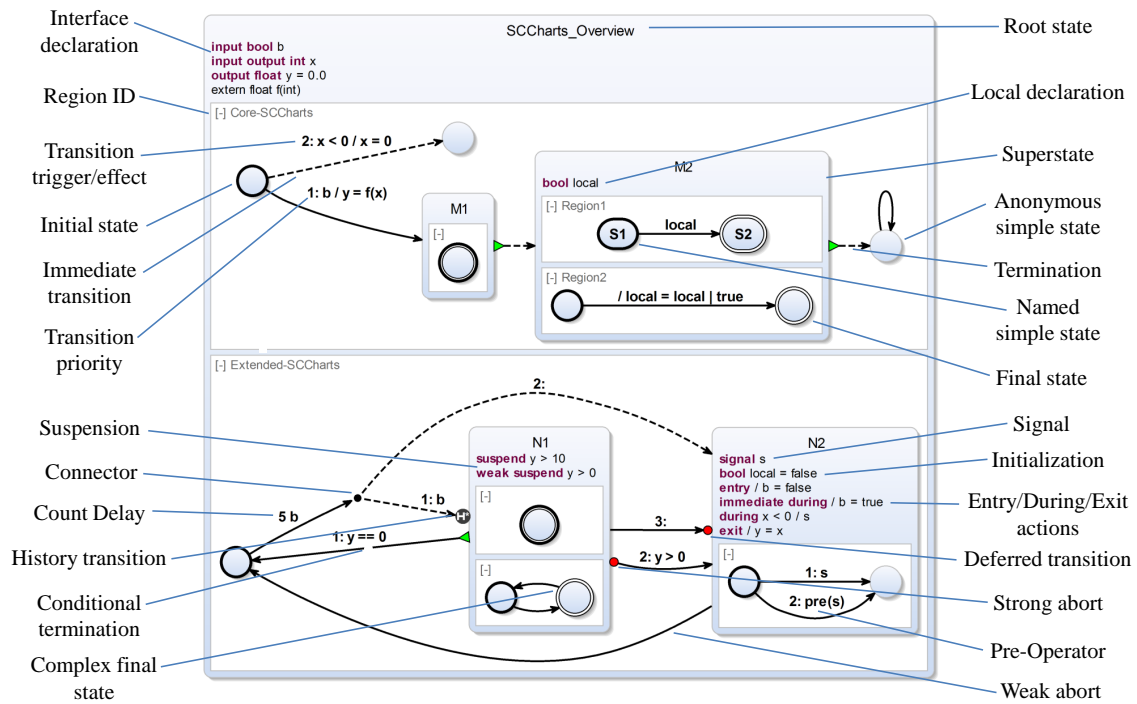


Figure 1.1. Overview of *core SCCharts* (above) and *extended SCCharts* (below) statements [HDM+14]

This SC MoC is then implemented in the modeling language *Sequentially Constructive Charts (SCCharts)* [HDM+14]. *SCCharts*, based on a statecharts dialect [Har87], has been designed with *modeling pragmatics* in mind. The syntax of the language is designed such that an easy synthesis of an automatic layout is possible, relieving the modeler from having to create a readable layout. This facilitates the simplicity of the synchronous language. *SCCharts* consists of a small set of statements, the *core SCCharts*. To simplify the use of *SCCharts*, some further statements were added. This larger set of statements, called *extended SCCharts*, can be transformed to *core SCCharts* using various Model to Model (M2M) transformations. The result of such a transformation is semantically identical to the extended statement. Figure 1.1 depicts both *core SCCharts* statements in the top and *extended SCCharts* statements in the bottom of the image. The reduction to a small set of simple statements in *core SCCharts* facilitates an easier compilation of the model to executable code. In fact, a *core SCCharts* can very simply be translated into an SCG, a Control-Flow Graph (CFG) based representation of the model. Such an SCG can then be used to analyze a model for schedulability and thus determinism and further to generate code in a target language that can easily be compiled and executed.

	Data-flow	Priority
Accepts all ASC SCCharts	+	+
Accepts all data-flow acyclic SCCharts	+	-
Accepts all iur acyclic SCCharts	-	+
Can synthesize hardware	+	-
Can synthesize software	+	+
Size scales well (linear in size of SCCharts)	+	+
Speed scales well (execute only "active" parts)	-	+
Instruction-cache friendly (good locality)	+	-
Pipeline friendly (little/no branching)	+	-
WCRT predictable (simple control flow)	+	+/-
Low execution time jitter (simple/fixed flow)	+	-

Table 1.1. Comparison of data-flow vs. priority based low-level compilation approaches, the lower part only applies to software [HDM+14]

1.2 Problem Statement

Due to the strict demands to synchronous languages as described in Section 1.1, their underlying MoCs define a precise set of rules. These rules must then be taken into account during the compilation and execution of a program written in such a language. Writing a compiler for a synchronous language that adheres to all rules of the corresponding MoC is a difficult task. Often, many revisions are required to perfect the compiler for all special cases, while sometimes multiple approaches towards the compilation are feasible. As an example, there are multiple versions of the official compiler for the synchronous language Esterel – the current version is the seventh iteration – and there are various different approaches to its low-level compilation each with its own advantages and disadvantages [BCE+03; Ber05; EKH06; PS04]. Similar to Esterel, there have been two different approaches proposed for the low-level compilation of SCCharts: The data-flow approach, which is at the point of this thesis used as the standard compilation approach for SCCharts, and the priority-based approach, which has been used as a compilation approach to SyncCharts, a statecharts dialect similar to SCCharts [And96], to generate deterministic C code [Han09b]. Both approaches were introduced by von Hanxleden et al. [HDM+14]. Table 1.1 depicts the advantages and disadvantages of both approaches as noted by von Hanxleden et al. As can be seen, in the second and third row of the table, the two approaches are able to compile some models that the other cannot, even though the models adhere to the MoC of SCCharts.

As an example, Figure 1.2 depicts an SCG of a model containing a cycle in the control-flow with functionally identical C code next to it in Listing 1.1. The code is naturally deterministic, as long as `foo` is deterministic, since the program does not contain any concurrency. However, the data-flow approach cannot schedule this model. The generated code of this approach tries to create a linear sequence of code where each line of code must only be traversed once per tick, while loops or `gotos` are not allowed. Since the code generation cannot determine

1. Introduction

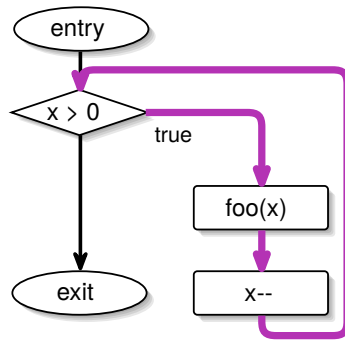


Figure 1.2. The SCG of a simple cyclical model, the cycle is highlighted in purple

```
1 void main() {
2   int x;
3
4   // Read x
5
6   while (x > 0) {
7     foo (x);
8     x--;
9   }
10 }
```

Listing 1.1. C code of a simple cyclical model

the number of iterations of the loop, as it depends on an input, the code cannot be flattened beforehand. Thus, a loop or goto-statement would be required. The priority-based compilation approach on the other hand allows goto-statements and re-execution of parts of the code.

Next to cyclical models, there are also *schizophrenic* models the data-flow compiler struggles with. In short, schizophrenic models contain a set of statements that is executed more than once during a tick. In these models, the scheduler of the data-flow approach currently detects a cycle in the model and rejects these. There are already proposed solutions to this problem that show promise. The priority-based compilation, however, might be more suited for these schizophrenic models, since it already allows cycles in the model.

Table 1.1 further delineates differences in the execution time of the compiled code. It is stated that the execution time of priority-based compiled models scales better in the size of the model. In fact, the execution time in the data-flow compilation approach scales almost linearly in the size of the model. Models for embedded, reactive systems may not necessarily be small and the demands to a short reaction time can sometimes be crucial as the example of the airbag shows. Therefore, the better scalability with the size of the model inherent to the priority-based compilation may outshine its disadvantages.

This demand for short reaction times further necessitates an optimized compiler. One aspect of an optimized compiler is the way it handles context switches. If it can predict context switches reliably and reduce unnecessary ones, it allows even better execution times. Thus, the handling of context switches in the priority-based compilation was improved. Extended testing via benchmarks and comparisons towards the old implementation as well as other compilation approaches should be performed to prove these improvements.

1.3 Outline

This thesis begins with an introduction into various theoretical foundations of synchronous languages and sequential constructivity in Chapter 2. This includes the theoretical basis

for the SC MoC as well as the imperative language SCL, a minimal programming language adapted from C, and SJ, an extension to Java similar to SCL. Important used technologies will be highlighted including the research project KIELER, which provides the basis for the implementation of the SC MoC. Chapter 3 then continues to give a short overview over different work related to topics of this thesis. It will cover the data-flow approach that is currently implemented in SCCharts and how SyncCharts, PRET-C and Esterel compile to executable code. The chapter also covers how each of these languages deal with schizophrenia. In Chapter 4 the conceptual basis of the thesis will be explained. It first begins with the theory behind the priority assignment and code generation. Subsequently, it will discuss the optimizations done to the priority assignment that enhance the resulting code by minimizing context switches. The chapter will cover the topic of schizophrenia in the priority-based compilation. Afterwards, Chapter 5 will deal with the implementation of these concepts in the Kiel Integrated Environment for Layout Eclipse Rich Client (KIELER) project. Chapter 6 then compares the implemented approach implemented in this thesis with the data-flow approach. Finally, Chapter 7 concludes the thesis and shows further possible development for the priority-based compilation and for SCCharts in general.

Foundations

Any embedded, reactive system constantly interacts with its environment, by reading values as inputs, processing them and outputting new values towards the environment. Conceptually, each computation beginning with the input and ending with the output can be regarded as one singular, atomic execution. The *synchrony hypothesis* utilizes this behavior to abstract from continuous time towards discretized physical time [PST05]. Each discrete point in time is called a *tick*, containing one computational reaction from input to output of a system. For the purpose of the environment, the outputs of the system happen at the same point of time as the inputs are provided. A system with such characteristics is considered to be in *perfect synchrony* [Ber00a]. Figure 2.1 pictures such a reactive system. In reality, no system is able to react in absolutely zero time after receiving an input. Instead, if the system reacts so fast that any calculations of outputs are generated before the next interaction with the environment, this reaction can be regarded as instantaneous. Therefore, an implementation of a system like this must guarantee that all computations during a tick are executed in the time between each interaction, which may often require extensive Worst Case Execution Time (WCET) analysis of these systems.

One MoC that utilizes this abstraction of zero-time computation is the synchronous MoC as exemplified by programming languages such as Esterel, Lustre or SynchCharts [BCE+03]. It describes in general how a programming language adhering to it should behave, one important aspect being concurrency. Even if simultaneous threads share accesses to variables, all race conditions must be solved deterministically. A simplistic way to guarantee that a program is deterministic is to only allow one write to a variable per tick and to schedule all writes to any variable before all reads to any variable. Thus, the program starts with an initialization of variables and all actions are dependent on these initial writes. While this limits programming languages unnecessarily, it is easy to implement and to check for. The synchronous MoC loosens this restriction to scheduling by allowing interleaving variable accesses as long as writes to a variable are scheduled before reads to the same variable, conserving the determinism of the system while allowing more freedom for the programmer. It further establishes *signals* for communication between threads: a signal can be either *present* or *absent* during a tick, and its state must be consistent across the tick, analogous to the behavior of a wire connection in a circuit.

2. Foundations

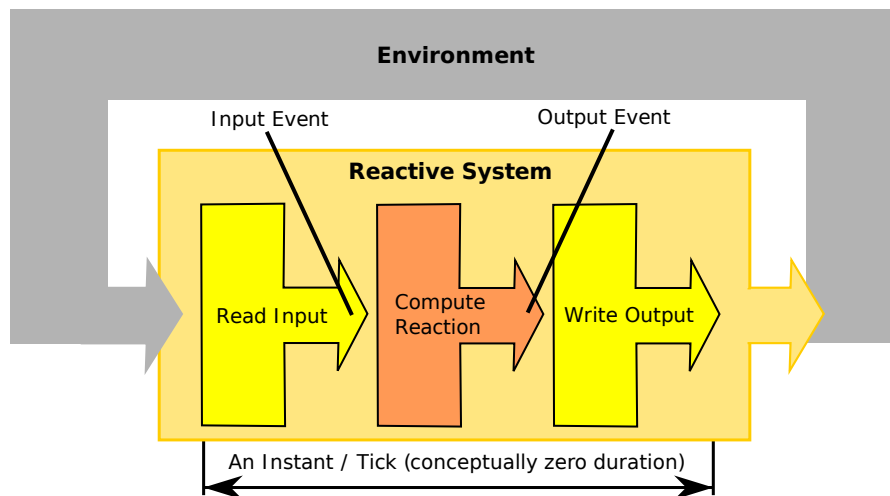


Figure 2.1. Visualization of a reactive system [MHH13]

2.1 Sequential Constructiveness

The SC MoC as presented by von Hanxleden et al. [HMA+14] describes a way to schedule synchronous execution of programs to achieve deterministic concurrency. In contrast to other synchronous MoCs, however, the SC MoC allows sequential writes to variables. If the order of execution is determined by the order of the instructions, then the program is still deterministic. For example, a simple code snippet such as `if (x < 3) { x = 3 }` would be rejected by the synchronous MoC since any writes to a variable have to be executed before reads. No race condition exists in this example, however, since there is no parallelism that can cause scheduling conflicts, and the program is inherently deterministic. Therefore no reason should speak against accepting this example. The SC MoC further allows the use of signals based on the semantics of imperative programming language. This simplifies the access for developers coming from traditional programming languages. Since variables can emulate the semantics of signals, all models using signals in other synchronous MoCs can be ported to the SC MoC preserving their original behavior. The SC MoC otherwise follows the same semantic rules as the synchronous MoC. It thus retains the determinism of traditional synchronous MoCs while extending them conservatively in order to help to abstract away from the hardware-centric approach of other synchronous languages closer towards traditional programming paradigms known from C or other imperative programming languages. The following section is mostly based on von Hanxleden et al. [HMA+14].

2.1.1 SCL and SCG

The semantics of the SC MoC can be described using a minimal Sequentially Constructive Language (SCL). As depicted in Table 2.1, SCL can in turn be represented using an SCG for elaboration, analysis and code generation. The SCG is a labeled graph $G = (N, E)$ consisting

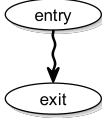
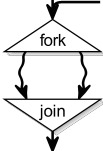
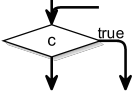
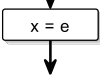
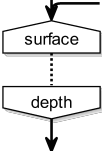
	Thread	Parallel	Sequence	Conditional	Assignment	Delay
SCL	t	fork t_1 par t_2 join	$s_1; s_2$ goto l	if (c) s_1 else s_2	$x = e$	pause
SCG			↓			

Table 2.1. Overview of SCL and SCG elements, based on [RSM+15] and [Sch16]

of *statement nodes* N representing the statements of a model and *edges* E corresponding to the control-flow or data-dependency induced order between statements. Control-flow edges can be either instantaneous or delayed. Instantaneous edges simply imply the sequential order of the execution between two nodes, while delayed edges induce a pause happening between the execution of two nodes, causing a divide in the execution into different ticks. Everything before the pause has to be performed before the end of the tick, while everything after the pause will be executed during the following tick after the environment could react to the previous tick. To explain dependency edges, first we must introduce *concurrency*.

A program may consist of multiple parts that are executed in parallel. For this, *threads* are introduced. At the initialization of the program, a *root thread* is spawned that starts the execution. New threads are spawned at *forks* and terminated at *joins* or, in the case of the root thread, at the end of the program. A thread can be uniquely identified by its *entry* and *exit* nodes, and all nodes of the program belong to their immediately enclosing thread or enclosing entry and exit nodes. If a thread is forked, the thread corresponding to the fork node is its *parent thread*. The set of *ancestor threads* is defined as the transitive closure of the parent relationship. Therefore, the ancestors of a thread include the thread itself as well as its parent thread until the root thread is reached. The root thread does not have any parent threads, thus contains only itself in its ancestors. The *descendant* threads of a thread is the inverse ancestor relation.

Concurrency and Scheduling Order in SCL

It is important to distinguish between static *threads* and dynamic *thread instances*: While static threads describe threads in the structure of the program, thread instances describe the actual parallel *execution* of statements during the run of the program. A static thread may thus correspond to multiple thread instances at different points in time of the execution of the program. Two threads are called *concurrent* if they are descendants of distinct threads sharing a common fork node. This fork node is also called the Least Common Ancestor (LCA). Threads further have a *thread state* as depicted in Figure 2.3 that determines their behavior during a tick. All threads other than the root thread are initially *disabled*. If a thread is forked or if it is the root thread at system start its state is changed to *enabled*. After a thread is enabled, it

2. Foundations

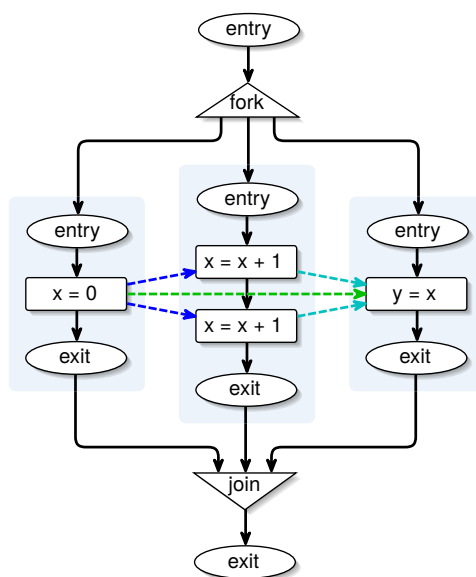


Figure 2.2. Example SCG exemplifying dependencies and the iur protocol [Sch16]

initially is *active*, and if it runs into a pause, it is *paused*. All enabled but paused threads are set to active at the start of the next tick. If a thread forks children, it is set to waiting, as it waits for the execution of its children to finish, and continues to be active after all children terminated and joined.

Two nodes are called *statically concurrent*, if the static threads they belong to are concurrent. They may further be *runtime concurrent*, if their *node instances*, which are analogous to thread instances, occur in the same tick, the threads they belong to are statically concurrent, and if these two threads have been instantiated by the same *instance* of the LCA fork. If two nodes are statically concurrent and they are assignments accessing the same variable, there may be a dependency between these nodes. The SC MoC differentiates between three different types of assignments: *Initializations*, *updates* and *reads*. An initialization to a variable is an absolute write, while an update is a relative write. The *iur protocol* specified by the MoC then determines the order of concurrent assignments accessing the same variable. The example SCG in Figure 2.2 shows this behavior. The assignment $x = 0$ in the leftmost thread acts as an initialization to x . The iur protocol then dictates that this initialization has to be executed before any updates and reads to x as highlighted by the dependencies in blue and green. The two updates to x , $x = x + 1$ in the middle thread, therefore are executed afterwards, but also before the read of x in the assignment $y = x$ in the right thread. All these dependencies can be conflated to *write-read* dependencies, where a variable has to be written before it may be read from. Sometimes however, the order cannot be determined this easily. If two nodes in parallel try to write different values to the same variable (without one of the two being an update), no order can be determined, since the outcome would differ based on the order of execution and the program would be non-deterministic. The two nodes are thus *conflicting* and the

2.1. Sequential Constructiveness

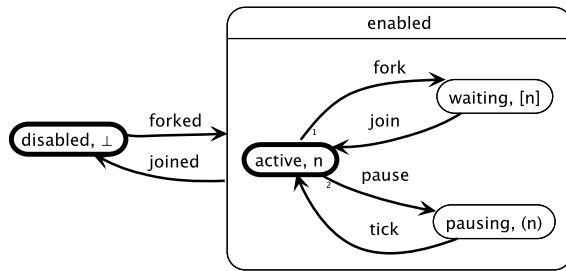


Figure 2.3. Execution states of a thread, shown with a Statecharts notation; Initial states have a bold outline [HMA+14]

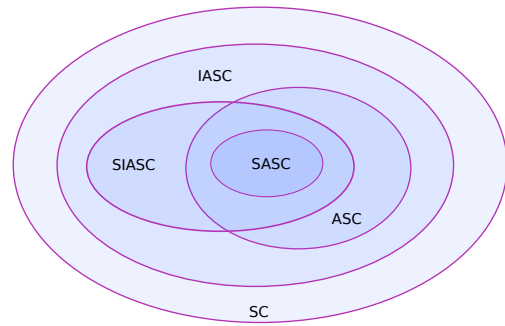


Figure 2.4. The different classes of sequential constructiveness of programs

program should be rejected, if such a *write-write* dependency exists. If both assignments try to update the same variable, one must find out whether the order of the two influences the result of the calculations. Thus, it is checked whether there exist two sequences of execution steps where the outcome differs, yet the only difference between these execution steps lies in the order of the two updates. As an example, $x = x + 3$ and $x = x + 5$ are both updates, but if they are executed concurrently, the resulting value of x is independent from the order of execution. The order of the two statements $x = x + 3$ and $x = x * 5$, however, influences the result of the calculation, even though both statements are updates. If therefore two sequences exist, where the outcome is dependent on the order of execution of the statements, the node instances are again *conflicting*, and the program is non-deterministic; otherwise the node instances are *confluent*, and they can be scheduled freely. If two node instances are confluent and there exists a dependency between the two, the dependency is also called confluent. If finally both assignments only read from the same variable, their order of execution does not change the outcome of the program. Therefore, no dependencies exist between such nodes and the scheduler may order the execution freely.

Schedulability in SCL

If no conflicts exist in the schedule of a tick, and the iur protocol as well as the sequential order of the nodes have been considered, the schedule is called *SC-admissible*. If all ticks during the execution of a sequence of ticks of a program are *SC-admissible*, this *run* is an *SC-admissible run*. If there exists at least one *SC-admissible run* of the program, and all *SC-admissible runs* generate the same deterministic trace of macro ticks, the whole program is considered *SC-schedulable*. Since the analysis of all possible *SC-schedules* is not feasible, especially for larger programs, further program classes were introduced. Figure 2.4 shows the different classes and their relation towards each other, while Figure 2.5 depicts various models highlighting the differences between the classes. The most restricting class, Structurally Acyclic Sequentially Constructive (SASC), allows only models where the SCG does not contain any immediate

2. Foundations

cycles in the graph, including those that only contain immediate control-flow edges and those that may contain iur dependencies. Figure 2.5e shows such an example. In contrast, Structurally iur-Acyclic Sequentially Constructive (SIASC) allows those models that contain cycles in the control-flow as long as they do not contain any iur dependencies, as Figure 2.5d depicts. Models that are Acyclic Sequentially Constructive (ASC) and iur-Acyclic Sequentially Constructive (IASC) then follow rules analogous to SASC and SIASC, yet allow models that contain (iur dependency) cycles as long as they cannot be reached in any run of the model. Figure 2.5c shows a model that is ASC but not SASC, as the immediate cycle executing `l++` cannot be reached in any run of the model. `l` is never smaller than 0 and thus the control-flow never reaches that part of the program. Analogously, the dependency cycle highlighted in red in Figure 2.5b can never be reached. Since `l` is always greater than 0, all runs of the model will take the right path and increase `l` until it reaches 10. Due to the immediate cycle, the model is not ASC. Lastly, Figure 2.5a depicts a model that is only SC but not any other class. A run of the model begins by setting both `x` and `y` to 0. Afterwards, in the concurrent regions, `x` is incremented and `y` is set to the value of `x`. Due to the dependency, the increment of `x` must be executed before it is written to `y`. After the threads terminated, it is checked whether `y` now has a value below 2. As it was just set to 1, this condition holds. Therefore, the loop leads back to the fork of the two threads, creating an immediate cycle. Due to this immediate cycle, the program is not ASC. Since this immediate cycle further contains a dependency edge, the model contains an iur dependency cycle and is not IASC. Still, during the second execution of the two parallel threads, the iur protocol is kept. Due to the dependency between the two assignments, the same order as in the first iteration is executed. Even though the first incarnation of the `y = x` assignment was performed before the second incarnation of the `x = x + 1` assignment, and therefore an update to a variable was executed after a read, the two instances were not runtime concurrent. Thus, the dependency between the nodes does not need to be regarded between those two node instances. The run of the program continues after the join, when the conditional evaluates to false after `y` was set to 2. Thus, the run ends. As this run is valid and no other SC admissible runs exist for this model, the model is SC.

2.1.2 The SC Language and Priorities

SCL as introduced earlier was implemented in C via various macros as Sequentially Constructive Language with priorities (SCL_p). Listing 2.1 shows a selection of the more important macros. Each thread of the program is assigned a uniquely identifying integer using a scalar. The integers *active* and *enabled* save the states of each thread in the corresponding bit of the integer. If a thread is forked, it is set to active and enabled and the program counter of its continuation is saved for later use. Due to the use of a single integer for the storage of states for threads, the number of threads in a program is limited to the size of an integer. The implementation can also use unsigned long integers extending the maximum number of threads, yet it still is limited. For each number of forked and joined threads, a macro exists or must be pre-generated. If a thread runs into the *par*-macro, it gets disabled, and if a join is performed, the joining thread checks for the enabled-state of all threads identified by the

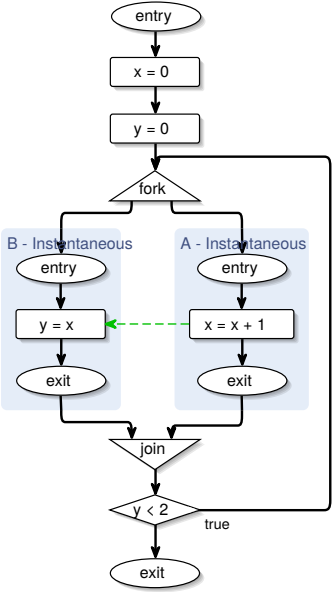


Figure 2.5 (a) Example SCG of a model that is only SC

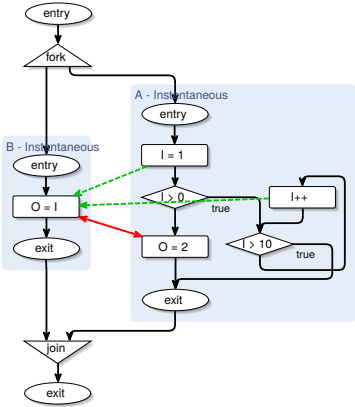


Figure 2.5 (b) Example SCG of a model that is IASC but neither ASC nor SIASC

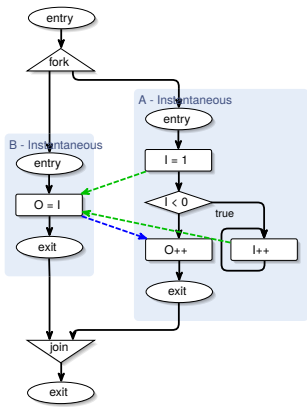


Figure 2.5 (c) Example SCG of a model that is ASC but not SASC

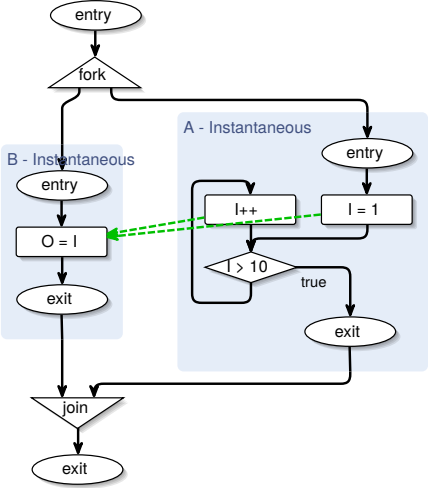


Figure 2.5 (d) Example SCG of a model that is SIASC but not SASC

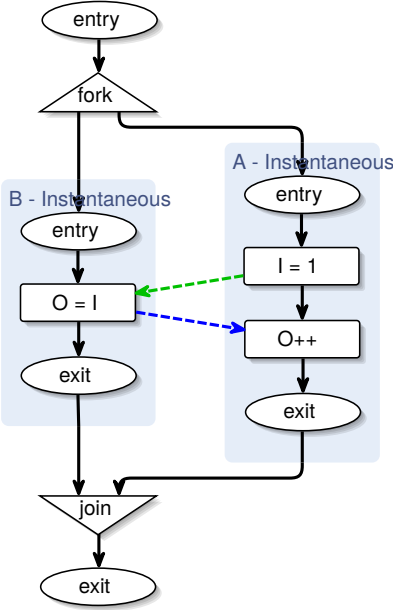


Figure 2.5 (e) Example SCG of a model that is SASC

Figure 2.5. Models representing the different classes of SC

2. Foundations

```
1 // Enable or disable threads with prioID p, activate analogous
2 #define u2b(u)      (1 << u)
3 #define enable(p)   enabled |= u2b(p); \
4                   active |= u2b(p);
5 #define isEnabled(p) ((enabled & u2b(p)) != 0);
6 #define disable(p)  enabled &= ~u2b(p)
7
8 // Pause and resume here in the next tick
9 #define _LABEL_     _L ## __LINE__
10 #define _pause(label) _setPC(_cid, label); \
11                    deactivate(_cid); \
12                    goto _L_DISPATCH
13 #define pause      _pause(_LABEL_); \
14                    _LABEL_:
15
16 // Fork and Join sibling with prioID p
17 #define fork1(label, p) _setPC(c, label); enable(p);
18 #define join1(p)        _LABEL_: if(isEnabled(p)) \
19                    { _pause(_LABEL_); }
20
21 // Terminate thread
22 #define par            goto _L_TERM;
23
24 // Tick End
25 #define tickreturn    goto _L_TERM; \
26                    _L_TICKEND: return (enabled != (1 << 0)); \
27                    _L_DISPATCH: __asm volatile("bsrl %1,%0\n":"=r" (_cid): "r" (
28                    active)); goto *_pc[_cid];
29
30 // Thread Priorities
31 #define prio(p)        deactivate(_cid); \
32                    disable(_cid); \
33                    _cid = p; \
34                    enable(_cid); \
35                    setPC(_cid, _LABEL_); \
36                    goto _L_DISPATCH; \
37                    _LABEL_:
```

Listing (2.1) Selected SCL_P macros, the field `_cid` stores the currently active thread; `__LINE__` expands to the current line number.

arguments of the join. If they are all currently disabled, the thread continues after the fork. Otherwise, the thread pauses and checks again during the following tick. This approach towards implementing the SC MoC in C uses the associated number of a thread not only as an identifier, but also as its *priority*. The differences to the original minimal SCL are highlighted in Table 2.2. Instead of only allowing at most one thread to be forked at a time, multiple fork and join macros were implemented allowing a set of threads to be forked and joined by only one macro respectively. During the execution, a scheduler determines which thread executes at which point in time using the active scalar. Each time the dispatcher is called, the macro as expanded in line 27 in Listing 2.1 determines the thread with the currently highest priority using the *bit scan reverse* assembler instruction for the x86 instruction set. This instruction returns the index of the currently highest set bit in the active scalar and the program subsequently jumps to the label as saved in the `_pc` array. If the system does not

	Thread	Parallel	Sequence	Conditional	Assignment	Delay	Priority
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	–
SCL _P	t	forkN t_1 par [...] t_N joinN	$s_1; s_2$ goto l	if (c) s_1 else s_2	$x = e$	pause	prio(x)
SJ	t	fork() $t_1 t_2$ join()	$s_1; s_2$ gotoB l	if (c) s_1 else s_2	$x = e$	pauseB	prioB(x)

Table 2.2. Overview of SCL, SCL_P and SJ elements, based on [RSM+15] and [Sch16]

run the x86 instruction set, the currently highest priority must be found using a loop over bits of the integer representing the enabled threads. This necessitates that no two run-time concurrent threads may be assigned the same priority, as the priority also uniquely identifies each thread. Conservatively, this can be extended to statically concurrent threads to minimize analysis. The new *prio*-macro allows a thread to change its priority by essentially disabling the old thread and spawning a new thread with a new priority that continues after the *prio*-statement of the old thread. The dispatcher is called as it may be possible that another thread now has a higher priority than the newly lowered priority of the original thread. The order of execution determined by the priorities now allows the program to adhere to the SC MoC. Naturally, the assigned priorities therefore have to specify an order that satisfies this MoC.

2.1.3 Synchronous Java

Similar to SCL and SCL_P, Synchronous Java (SJ) enhances Java with "macros" for deterministic concurrency using priorities. However, since Java does not have traditional macros and more importantly neither a goto-statement nor labels with the same functionality as C-labels, different concepts were used, as delineated in Table 2.2. To replace macros, all required functionalities were implemented using standard Java methods. The class containing the tick-function has to extend the class containing those methods. Instead of traditional gotos, the content of the tick-function is surrounded by a while-loop that is repeated as long as at least one thread is still active during the current tick as shown in Listing 2.2. The `isTickDone` method that is called each time the loop restarts checks whether any active threads remain. The loop then contains a switch-case statement, where each case represents a "label" analogous to C. The switch-statement then decides using enabled- and active-arrays similar to the scalars in C which thread to continue at which "label" in the state method. A `gotoB`-method was introduced that assigns a continuation "label" to the currently active thread to continue at a later point. After each `gotoB` as well as after all other macros where a jump towards another label follows, a `break` has to be inserted. The switch-case statement will be left, the while-loop begins anew and the next continuation point of the program is chosen. Both the remodeled `prioB` and `pauseB` corresponding to the `prio` and `pause` macros require continuation

2. Foundations

```

1  protected final void tick() {
2  while (!isTickDone()) {
3  switch (state()) {
4  case init:
5  pauseB(start);
6  break;
7  case start:
8  if (I) {
9  gotoB(middle);
10 break;
11 }
12 gotoB(start);
13 break;
14 case middle:
15 pauseB(middle_2);
16 break;
17 case middle_2:
18 O = true;
19 gotoB(end);
20 break;
21 case end:
22 haltB();
23 break;
24 }
25 }
26 }

```

Listing 2.2. Simple example of sj code

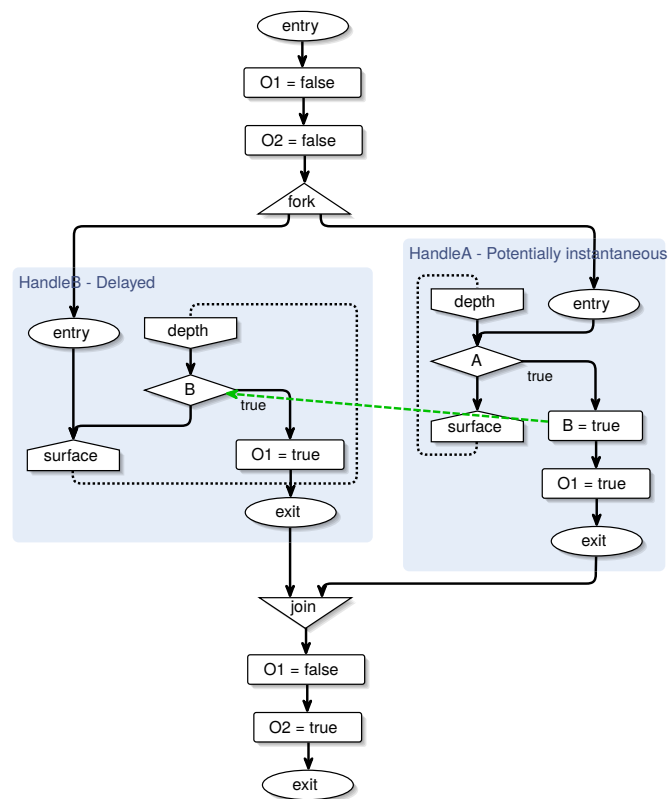


Figure 2.5. The SCG of ABO

"labels". Since there are no expanding macros in Java, dynamic labels cannot be created. Thus, the continuations for the next thread or after the pause must be specified statically by the program.

2.1.4 The ABO Example

ABO is a simple example model showing characteristics of SC and SCL. In its simplest form, the model has two inputs, A and B, as well as an output O. When both A and B were set to true at any point during the execution, O is also set to true, except during the first tick, when the model ignores all inputs. The ABO model depicted in Figure 2.5 shows a variation of the standard model. In the beginning of the execution, the output variables O1 and O2, which replace O, are set to false. Then, in two concurrent threads, both A and B are awaited. While HandleB has to wait at least one tick after the beginning of the execution to react to inputs, HandleA may react immediately. When A is set to true by the environment, B is also set to true, causing a dependency in the SCG between the two threads. If A or B is set to true at any point in time, O1 is also set to true, indicating that exactly one of the two inputs has been set to

true. After both inputs have been set to true and they reached the exit node, the threads are joined, O1 is set to false again and O2 is set to true. This model therefore shows the concepts of immediate control-flow edges in HandleA, delayed control-flow edges in HandleB as well as dependency edges between the two threads.

2.2 Used Technologies

The implementations of the aforementioned concepts and foundations as well as the concepts proposed in this thesis are part of the KIELER project. KIELER itself is based on the Eclipse platform and utilizes further technologies and plugins provided by Eclipse. The following sections give a short overview over Eclipse as well as KIELER due to their prevalence in the creation of this thesis.

2.2.1 Eclipse

Eclipse¹ is an open source Integrated Development Environment (IDE) developed by the Eclipse Foundation. Built upon the Java programming language, it not only provides support for software development in Java, but also in various other programming languages such as C++, XML and more. The architecture of Eclipse consists of a small set of components called *plugins*. New plugins created by third parties may then extend the original architecture to create new functionality such as new UI Elements or even whole new modeling languages. The extensibility is achieved by using so called *extension points*. A plugin can define an extension point to provide an access point for another newly developed plugin, which can then utilize functionality supplied by the original plugin. The Eclipse Rich Client Platform (RCP) is a small set of plugins serving as an open tools platform to provide components for the development of client applications in Eclipse. Other RCPs can build upon this platform to build applications ranging from IDEs to browsers and textual editors.

Xtend

Xtend² is a dialect of Java, which compiles to readable Java 5 compatible source code. Xtend allows easy use of extension methods and lambda expressions. It is integrated into the Eclipse Java Development Tools. The initial intention of Xtend was to ease M2M transformations. However, Xtend has been used in various other domains such as general Android development³ or UI development⁴ as well. Compared to Java, it gets rid of a lot of overhead. Listing 2.3 shows example Xtend code for sorting a list of albums for favorites on the left. The corresponding Java 8 code on the right depicts how difficult to read the lambda expressions

¹<http://www.eclipse.org>

²<http://www.eclipse.org/xtend/>

³<http://faturice.com/blog/android-development-has-its-own-swift>

2. Foundations

```
1 List<Album> sortedFavs =  
2 albums.stream()  
3   .filter(a -> a.tracks.anyMatch(t -> (t.rating >= 4)))  
4   .sorted(comparing(a -> a.name))  
5   .into(new ArrayList<>());
```

```
1 val sortedFavs =  
2 albums.stream  
3   .filter[tracks.anyMatch[rating >= 4]]  
4   .sorted[comparing[name]]  
5   .into(new ArrayList)
```

Listing 2.3. Simple example code showing the difference between Xtend and Java⁵

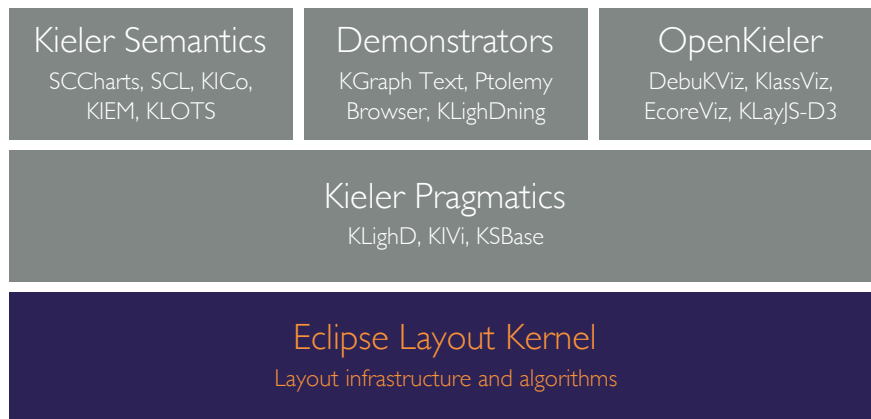


Figure 2.6. KIELER project overview⁷

may get in a realistic example. Both listings first filter a list of albums for a rating higher than 4, then sort by name and then add them into a new `ArrayList`.

2.2.2 KIELER

The KIELER⁶ project is an academic open source research project by the Real-Time and Embedded Systems Group at Kiel University. It is an RCP aiming to make graphical model-based design of complex systems easier by minimizing the time required for the visualization of programs. Due to the focus on modeling *pragmatics*, KIELER can improve comprehensibility, maintainability and analysis of complex models.

Thus, the first aspect of the KIELER project as delineated in the overview in Figure 2.6 is the KIELER *Pragmatics* [Fuh11], which used to be based on KIELER Infrastructure for Meta Layout (KIML) but now builds upon the Eclipse Layout Kernel (ELK)⁸. ELK provides infrastructure for automatic graph generation and layout creation as well as algorithms for different types of graphs including graphs with hierarchy or different requirements for port placements. Graph layouts alone, however, do not suffice. It is further important to be able to efficiently synthesize

⁴<https://www.javacodegeeks.com/2013/02/building-vaadin-ui-with-xtend.html>

⁵<http://blog.efftinge.de/2012/12/java-8-vs-xtend.html>

⁶<http://rtsys.informatik.uni-kiel.de/kieler>

⁷<http://rtsys.informatik.uni-kiel.de/confluence/display/KIELER/Overview>

⁸<http://www.eclipse.org/elk>

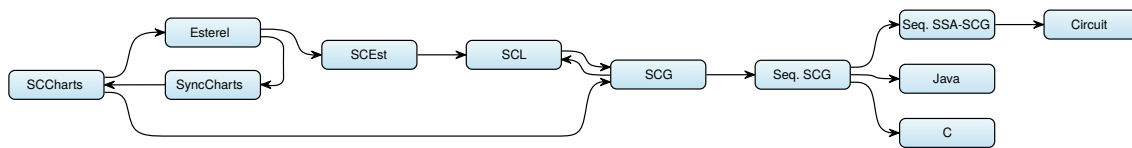


Figure 2.7. Overview of the KIELER Compiler [Sch16]

these graphs and visualize them for the viewer. KIELER Lightweight Diagrams (KLighD) [SSH13] provides transient and lightweight representations of a graph without the necessity of introducing possibly complex graphical editors. Another important aspect of KLighD is the concept of *focus and context*. Currently unimportant parts of the model can be minimized while more important parts can be highlighted.

The utility of these projects is demonstrated in the *OpenKieler* project, the *Demonstrators* and the *KIELER Semantics*. The latter focuses on the semantics of systems, particularly on synchronous languages. It offers a flexible compiler, the KIELER Compiler (KiCo), that is adaptable to multiple compilation chains. One tool that uses the compiler is KIELER SCCharts: It allows the user to create models for real-life applications using a textual language. A model can be synthesized in real-time side-by-side with the textual model by KLighD using layout provided by ELK.

Kieler Compiler

The KIELER Compiler [Mot17] is a compilation framework based on the Single-Pass Language Driven Incremental Compilation (SLIC) strategy as introduced by Motika et al. [MSH14] that utilizes the Eclipse plugin concept. It provides an interface for the application of multiple M2M transformations to an initial model. Due to the SLIC approach, each transformation is only called once and should provide a certain set of features to be added or taken away after the transformation. A M2M transformation can be registered using an extension point defined by KiCo and can specify other transformations that must be performed before this transformation. It is further possible to specify certain features that this transformation cannot handle. These features must therefore be guaranteed to be removed before the application of the transformation. KiCo then automatically calculates an order of all registered transformations to be applied that factors in all requirements.

The KIELER project utilizes this compilation framework for different compilation approaches and language translations. These transformations are visualized in Figure 2.7. The different supported languages as well as model representations in their general or special form are depicted as nodes, while the edges show the different compilation steps that are possible due to the M2M transformations. Each compilation step may be comprised of multiple M2M transformations. The actual KiCo implementation in KIELER further allows the user to view the model after every M2M transformation.

Related Work

This chapter introduces various work related to the content of this thesis. It begins in Section 3.1 with an alternative compilation approach inherent to SCCharts. The section highlights the differences between the two approaches and shows the result of this compilation so the results of this thesis can effectively be compared to them. Section 3.2 then presents Esterel, a synchronous language based on the synchronous MoC, in detail. Afterwards, Section 3.3 describes the synchronous language SyncCharts and how it can be compiled. Lastly, Section 3.4 presents another language with inherent deterministic concurrency in PRET-C.

3.1 Data-flow Low-Level Compilation of SCCharts

The data-flow compilation in SCCharts as introduced by von Hanxleden et al. [HDM+14] as well as Smyth et al. [SMH15] is an approach to generate code in various different languages or to synthesize hardware from a model in SCCharts. As opposed to the priority-based approach, this compilation chain generates a sequential netlist and therefore sequentialized code which does not require computed gotos. As stipulated by the SC MoC the approach allows sequential assignments to a variable.

Instead of compiling the code directly from the SCG as generated from SCCharts, the SCG first undergoes a few transformations itself. First, the SCG is divided into *basic blocks*. These specify parts of the code that are always executed together. If one statement in this code block is executed, all other statements in this code block will also be executed at some point during this tick. These basic blocks are then further broken up into *scheduling blocks* every time a statement is targeted by an incoming dependency. This subdivision facilitates interleaving of threads and helps to determine a schedule. After the scheduling blocks were created, all basic blocks are assigned *guards*. A guard describes the state of a block that may be *active* or *inactive*. It can rely on the completion of its *preceding* basic block, on the evaluation of an if-statement or on the state of the preceding basic block in the previous tick – for example for tick boundaries. For basic blocks beginning at joins, the guards cannot merely rely on the preceding basic blocks during this tick or the previous tick, but must rely on the termination of all of its preceding basic blocks in any previous tick. Thus, they check whether any preceding thread is still active by looking at all basic blocks ending in pauses of these threads in addition to checking whether the thread has finished during this tick. After these steps, the compiler has to find out whether the program is schedulable. In order to do that, the blocks are ordered depending on their guards and dependencies. If a block has a

3. Related Work

```
1  if(!_PRE_G0 == 1){
2    _G0 = 0;
3  }
4  {
5    g0 = _G0;
6    if(g0){
7      01 = 0;
8      02 = 0;
9    }
10   g1 = g0;
11   g5 = PRE_g4;
12   g2 = g1 || g5;
13   _cg2 = A;
14   g3 = g2 && _cg2;
15   if(g3) {
16     B = 1;
17     01 = 1;
18   }
19   g4 = g2 && (!_cg2);
20   g6 = g0;
21   g8 = PRE_g7;
22   g8b = g8;
23   _cg8 = B;
24   g7 = (g8b && (!_cg8)) || g6;
25   g9 = g8b && _cg8;
26   if(g9) {
27     01 = 1;
28   }
29   g3_e1 = !g4;
30   g9_e2 = !g7;
31   g10 = (g3_e1 || g3) && (g9_e2 || g9) && (g3 || g9);
32   if(g10) {
33     01 = 0;
34     02 = 1;
35   }
36 }
37 PRE_g4 = g4;
38 PRE_g7 = g7;
39 _PRE_G0 = _G0;
40 return;
```

Listing 3.1. Generated C code of the ABO example from Figure 2.5 by the data-flow compilation of SCCharts; Further code backends are also possible.

guard or dependency towards another block, it has to be ordered after this block. If this order cannot be defined due to any cycles, the program is not ASC schedulable and will be rejected. Any valid resulting schedule is not necessarily the only valid schedule, yet all schedules will produce the same deterministic output. From this scheduling order, a sequentialized program – the aforementioned netlist – can be generated, which then in turn can be translated to C or Java code via pattern matching from top to bottom.

The code in Listing 3.1 shows the ABO example from Figure 2.5 translated to C code by the data-flow compilation. All variables beginning with *g* are various guards created by the basic block and scheduling block generation. They determine whether any parts of the program in the *if*-blocks are to be executed or not, as they check for the active state of this part of the program. *PRE*-guards depend on the state of a guard in the previous tick. The first *if*-statement checking for the value of *g0* checks whether the program is in the very first tick, as *g0* is only dependent on the value of *_G0*, which is only true in the very first tick. Due to the dependency from the thread *HandleB* towards the thread *HandleA*, the latter has to be scheduled before the first in a sequentialized program. Thus, the next *if*-statement in lines 16 to 19 handles the statements from this thread. The guard *g3* depends on the completion of the very first block or the previous value of *g4* and the value of the input *A* due to the conditional checking for it. The value of *g4* is only true, if this block has not already been executed in a previous tick. This is due to the pause in the SCG that is executed, if *A* evaluates to false when checking for the condition. Therefore, the condition evaluates to true, if *A* evaluates to true and either the pause was taken during the previous tick, or the previous block as guarded by *g3* was executed during this tick. In the *if*-block, the values of *B* and *01* are set to true as determined by the SCG. The following *if*-block in lines 27 to 29 corresponds to the thread *HandleB* analogous to the previous block, with the exception that it cannot be executed during

the first tick, since the guard `g8b` depends on values of the guard `g7` during the previous tick. This happens due to the pause in the SCG directly before the conditional. Subsequently, the last `if`-block corresponds to the join. Its values depend on the termination values of the two threads as it checks for the termination of the threads or whether any thread ran into a guard signifying a pause. If either `g4` or `g7` evaluate to true, the thread will not terminate as then either `HandleA` or `HandleB` ran into a pause. If the conditional evaluates to true, the final part of the SCG is executed and the function terminates.

Even though this code is difficult to read and to comprehend, it does not require any additional macros and is compilable by all commonly used C compilers. Overall, no priorities are required and the execution of a tick simply goes through the whole netlist and executes all parts of the code that are currently active depending on the previously executed parts, instead of using computed gotos to find the next code to be executed.

Due to the restriction that no cycles should be in the static schedule of an SCG, the data-flow compilation struggles with schizophrenia, as multiple executions of a statement require some path through the SCG that induces an immediate cycle. However, even if this cycle cannot happen in a run of the model, the model will be rejected due to the static cycle. Some solutions to this problem include for example *depth-joins* or *surface copies*. A depth-join is a join where at least one of its threads is potentially instantaneous, while at least one other thread is certainly delayed. A static structural analysis over a control-flow graph would find a cycle, yet due to one of the threads being delayed, the cycle cannot be instantaneous. Thus, the schedule should be accepted. The depth-join itself would then require a second exit of the thread in the control-flow graph that could only be taken if the thread was entered in the same tick as the exit is taken. This exit would then not connect to the join in the control-flow graph, thus breaking the instantaneous cycle. In each subsequent tick, the thread should, however, be able to be terminated ordinarily. The alternative surface copy requires – as the name implies – the surface of the thread to be copied, therefore requiring a lot more effort. Its concept will be illustrated in the next section, as it is easier to explain with Esterel.

3.2 Esterel

Esterel is a synchronous language based on the synchronous MoC [Ber00a]. Contrary to SCL it does not extend C but is its own language designed for reactive systems. It utilizes signals as a means of communication between concurrent executions. A signal can either be *present* or *absent* during a tick. This state must be globally consistent for the entire tick. To guarantee deterministic behavior, the signals must always be written first before they are read from. If a signal was presumed absent and sequentially later *emitted* and therefore set to present, the program must be rejected.

To be compiled to executable code, Esterel can use one of multiple approaches: Edwards [Edw99] first proposed the translation to an intermediate code produced by the front end of Berry's compilers, which then is translated to a Concurrent Control-Flow Graph (CCFG). Similar to SCGs, CCFGs are control flow graphs, where a lot of the characteristics of the un-

3. Related Work

```
1 loop
2   signal S in
3     present S then emit 0 end;
4     pause;
5     emit S
6   end;
7   present I then emit 0 end
8 end
```

Listing 3.2. Schizophrenic model in Esterel

```
1 loop
2   signal S in
3     present S then emit 0 end;
4     gotopause l;
5   end;
6   signal S in
7     present S then emit 0 end;
8     l: pause;
9     emit S;
10  end;
11  present I then emit 0 end
12 end
```

Listing 3.3. Solution to schizophrenia in Esterel by Tardieu and de Simone

derlying programming language were already translated to easier constructs. Afterwards, concurrency is stripped, respecting dependencies in the graph. Any cycles in these dependencies or in the graph itself would result in the compiler rejecting the program. The synthesized C code then contains unnested if-then-else structures checking if a part of the program should be activated at this instance or not, similar to the data-flow compilation of SCCharts.

Edwards et al. [EKH06] later proposed a way to execute Esterel code using linked lists to track which blocks of code are to be executed at what time, eliminating conditional tests. However, it still uses a variant of the Graph Code (GRC) by Potop-Butucaru [PS04], which in itself is a variation of the CCFG containing a control flow part and a selection tree describing the behavior of the program in each cycle. The code generation still requires a sequentialization of the graph. There are further approaches to the compilation of Esterel programs, however, due to the nature of Esterel, immediate loops are not allowed, making an approach with dynamic assignment of priorities in threads unnecessary.

Schizophrenia in Esterel is a highly discussed topic [Ber00b; SW01; TS04]. According to Berry [Ber00a], schizophrenia problems themselves occur rarely. Whenever a scope is left and reentered during a tick, the schedulability analysis and code creation, however, have to consider the *reincarnation* of a thread or signal. Listing 3.2 shows a small example Esterel program, where after the first tick the signal S is emitted. Afterwards, the signal scope of the local signal S is left and, depending on the input I, 0 is emitted or not. Subsequently, the loop restarts. The signal S is then reincarnated as a new signal scope of the signal is entered and its presence tested. Since it is in a new signal scope of S, however, S must not be present at this point in time.

A critical problem for any solution of this is that physical gates and wires in synchronous circles are only allowed to assume one single value at each clock cycle. Therefore, if an Esterel program is to be translated to circuitry, the different incarnations of a signal have to be distinguished carefully. A simple solution to schizophrenia in general is *loop unrolling* by duplicating the loop bodies that are to be reentered. This means that, instead of one statement having multiple incarnations during a tick, this statement is duplicated in the code and each of the duplicated statements has at most one incarnation during each tick. For Esterel to be

valid, the original statement must never terminate within one tick, as there must never be complete immediate loops in a program. Therefore, the duplicated loop body can also never terminate within one tick and no schizophrenia can happen at all. However, due to nested schizophrenic statements, the size of the program can increase exponentially, which is not desired.

Tardieu and de Simone [TS04] on the other hand propose a new *gotopause* statement. This statement acts as a normal pause in Esterel but jumps to the declared label after the pause was executed. Each pause statement in the loop body will further receive a unique label where a *gotopause* may jump towards. Then, instead of duplicating the whole loop body analogous to the naive solution, only the surface of the loop body is duplicated and each pause with its corresponding label in the surface is replaced with a *gotopause*. Since the label of the replaced pause also corresponds to a label of a pause in the surface of the original body, a *gotopause* will jump towards this pause in the next tick. The result of this transformation applied to the program from Listing 3.2 is shown in Listing 3.3. In it, the depth of the program until the first pause in line 4 in the original is copied and put into the beginning. The pause in line 4 is then replaced with a *gotopause* `l`. The signal scope of the depth is subsequently ended in line 5 and a new signal scope begins in line 6. This second scope encompasses the copy of the whole program. Line 8 shows that the original pause has been replaced with a labeled pause. The *gotopause* in the depth jumps towards this pause. As evident, the `emit S` statement in line 9 relates to another instance of `S` as the present `S` check in line 3 during the execution of the second tick, as the scope of `S` was left after the *gotopause*. Thus, no signal is reincarnated and no statement executed multiple times. Tardieu and de Simone argue that this solution of schizophrenia not only solves all schizophrenic models for generated code and circuits, but also that the implementation runs quasi-linear in practice.

3.3 SyncCharts and SyncCharts in C

SyncCharts [And96] are a variant of Statecharts [Har87] – a visual formalism of state machines and state diagrams – that allow concurrency as well as hierarchy and preemption while building upon the synchronous MoC to ensure deterministic concurrency. It aims for easy creation, editing and visualization of models for safety-critical embedded systems, similar to SCCharts. Its set of statements is based on Esterel and thus inherits the concepts of signals and valued signals as well as the underlying MoC from Esterel. Due to this, the compilation from SyncCharts to Esterel is simple. Figure 3.1 shows an example SyncChart of the ABSWO model. It works similar to the ABO model in Section 2.1.4, yet highlights the functionality of *strong* and *weak aborts*. The inner behavior of the `abo` state works almost the same as the one described in Section 2.1.4, with the exception that the model does not react immediately to the input `A`, but only delayed. The presence of the signal `S` – representing the strong abort – results in the whole model re-initializing before any inner behavior of `abo` can be executed during this tick. Thus, when `S`, `A` and `B` are all present during a tick, `O` will not be present, as the inner behavior has been preempted. Since the inner behavior is also non-immediate, the

3. Related Work

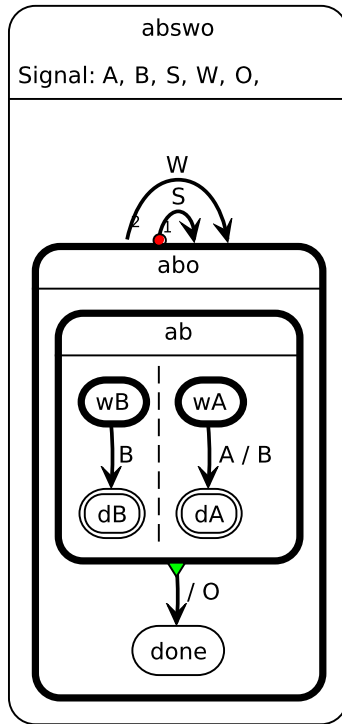


Figure 3.1. The SyncChart of ABSWO, highlighting strong and weak preemption

```

1      TICKSTART(5);
2  L_abo:  FORK(L_abo_ab, 2);
3          FORKE(L_abo_main);
4  L_abo_ab: FORK(L_abo_ab__wA, 4);
5          FORK(L_abo_ab__wB, 3);
6          FORKE(L_abo_ab_main);
7  L_abo_ab__wA: AWAIT(sig_A);
8              EMIT(sig_B);
9              TERM;
10 L_abo_ab__wB: AWAIT(sig_B);
11             TERM;
12 L_abo_ab_main: JOIN;
13              EMIT(sig_O);
14              HALT;
15 L_abo_main: PAUSE;
16           if (PRESENT(sig_S)) {
17             ABORT;
18             PRIO(5);
19             GOTO(L_abo);
20           }
21           PRIO(1);
22           if (PRESENT(sig_W)) {
23             ABORT;
24             PRIO(5);
25             GOTO(L_abo);
26           }
27           PRIO(5)
28           GOTO(L_abo_main);
29           TICKEND;
    
```

Listing 3.4. Synchronous C translation of the ABSWO example

presence of A and B is ignored at the restart of abo. The signal W on the other hand represents the weak abort. If W, A and B are all present during a tick, but S is absent, then O will be present during this tick. The inner behavior of abo is executed before the transition triggered by W is taken and abo is re-entered. Thus, the model can already react again during the next tick.

SyncCharts in C [Han09b] is an approach to embed SyncCharts constructs in C. It utilizes C macros and priorities assigned by hand by the modeler or passed down by a dependency analysis in SyncCharts to create *Synchronous C*. The priority assignment as introduced by Traulsen et al. [TAH10; TAH11] works on an intermediate representation of the SyncChart that highlights the order of hierarchy, transitions, dependencies and lastly control-flow, similar to the SCG. In contrast to the SCG, however, this *priority graph* also contains complex instructions such as strong and weak transitions. In contrast to the minimal language of SCL, Synchronous C implements more macros adapted from the semantics of SyncCharts. The translation of the ABSWO example in Listing 3.4 shows that Synchronous C also implements macros for awaiting a signal and the general signal control. Further macros that are implemented include the abort and suspend statements. As a precursor to SCL, one can, however, already see some

```

1 PAR(Thread1, Thread2);
2 [...]
3 void Thread1() {
4   A = 0;
5   EOT;
6   A = B + 1;
7 }
8 void Thread2() {
9   B = A;
10  EOT;
11  B = 7;
12 }

```

Listing 3.5. A simple PRET-C program [And13]

```

1 fork(Thread2) {
2   Thread1:
3   A = 0;
4   pause;
5   A = B + 1;
6 } par {
7   Thread2:
8   B = A;
9   pause;
10  B = 7;
11 }

```

Listing 3.6. A similar program, implemented in SCL

SCL	A	0	8
	B	0	7
PRET-C	A	0	1
	B	0	7

Table 3.1. Variable values at the end of the ticks of the small PRET-C and SCL programs.

concepts inherited by SCL. Both languages rely on labels and goto-statements and each fork is called with a label and a priority. A dispatcher in the background memorizes the continuation of each thread and decides which thread executes next based on the priority of all threads.

Due to its relation to Esterel, schizophrenia is similarly problematic for SyncCharts. Thus, the same proposed solutions can be applied after the translation to Esterel. However, von Hanxleden proposes that the translation to synchronous C solves schizophrenia problems inherently due to the way local variables are handled in C [Han09a].

3.4 PRET-C

PRET-C [And13] is, similar to SCL, an extension of C via macros that allows easy facilitation of concurrent programming for safety-critical systems. It further provides determinism, reactivity and thread-safe communication via shared memory access. It also allows sequential accesses to a variable that SCCharts allows but Esterel prohibits. PRET-C further uses variables like SCCharts and not signals like Esterel for communication between threads. However, in PRET-C each thread is given a static priority at its fork that determines the order of all threads. Therefore, PRET-C allows no interleaving of threads and requires more planning and knowledge of the underlying principles from the programmer. This approach, if implemented in SCCharts, would violate the iur protocol of the SC MoC, since it is possible that writes to a shared variable happen after reads in concurrent threads due to the scheduling of these threads. Still, the behavior of PRET-C is deterministic and predictable due to the fixed order of execution. Listing 3.5 shows a small example PRET-C program with an SCL translation with a similar behavior in Listing 3.6 that has to adhere to the SC MoC. The trace in Table 3.1, however, shows the difference in the execution. Due to the fixed order of the threads in PRET-C, $A = B + 1$ in line 6 in the program must be executed before the assignment $B = 7$ in line 12. The SC MoC, however, stipulates that writes have to be executed before reads. Therefore, the two statements are executed the other way around and the final value of A is 8 in SCL and 1 in PRET-C.

3. Related Work

Similar to the semantics of Esterel and the netlist-based compilation of SCCharts, PRET-C restricts loops by requiring at least one EOT (or end of tick, similar to a pause in SCL) in the loop body. When regarding schizophrenia, this could still lead to reincarnation of variables. However, PRET-C does not provide any local variables. Since only global variables exist, the scope of a signal can never be left and reentered. Therefore, reincarnation is not problematic in PRET-C. It is further not problematic for a statement to be executed twice during a tick. The semantics of PRET-C allows multiple accesses to a variable in a tick as long as they are sequentially ordered. Since this is the case, as the second iteration of a loop must be scheduled after the first, there is no problem concerning schizophrenic behavior in PRET-C.

Priority-Based Compilation

This chapter introduces the concept of the priority-based compilation of SCCharts. Section 4.1 covers the general, original approach to the compilation. It explains how priorities were calculated as well as how those priorities were used to create deterministic code of concurrent programs. It also covers peculiarities of the underlying macros and how those were adapted to the approach. Section 4.2 subsequently introduces certain optimizations to the calculation of priorities. Since the original approach can be improved upon regarding context switches, some changes to the priority calculations were made. It is then proven that these optimizations do not cause the program to be non-deterministic or to violate the iur protocol and that they actually improve the execution time of the program by reducing context switches. Section 4.3 finally covers the topic of schizophrenia in the priority-based compilation approach. It will talk about how schizophrenia is handled in the original approach and which adaptations were made to accept and correctly translate schizophrenic programs.

4.1 Priority-Based Compilation

The priority-based compilation approach as introduced by von Hanxleden et al. [HMA+13; HMA+14] describes a way to compile SCCharts models to C code. In the alternative, data-flow approach – as introduced by von Hanxleden et al. [HMA+14] and implemented by Smyth et al. [SMH15] as described in Section 3.1 – an SCCharts model is translated to an SCG model which is then sequentialized. Afterwards, the C code is generated using this sequential model. The priority-based compilation in contrast utilizes priorities and a dynamic scheduler, thus avoiding the sequentialization of the model. The core idea of the approach is the assignment of priorities to nodes. During the execution of a model with multiple parallel threads, the thread with the highest priority executes first. The priority of the threads is determined by the priority of their next node in the model. However, since no code can be executed on a model, the model must first be translated to SCL_P as introduced in Section 2.1.2. Thus, a translation from the SCG to this C dialect is necessary. The following subsections will introduce this process, beginning with the calculation of *priorities*, for distinguishing purposes in the following called Node Priorities ($nprs$) in Section 4.1.2, over calculating the Priority IDs ($priIDs$) in Section 4.1.3 to finally generating C and Java code in Section 4.1.4 and Section 4.1.5.

4. Priority-Based Compilation

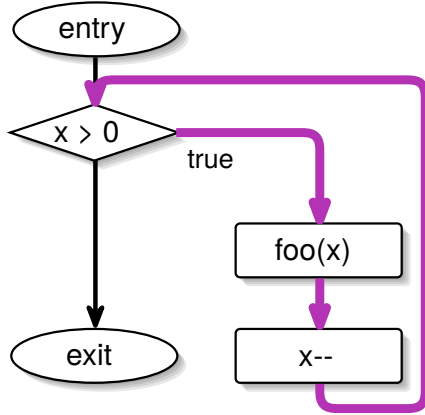


Figure 4.1. The SCG of a simple cyclical model with an immediate loop; The cycle is highlighted in purple.

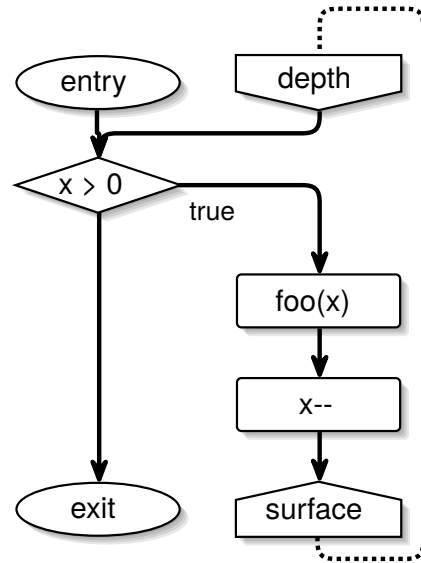


Figure 4.2. The SCG of a simple cyclical model, but without an immediate loop

4.1.1 Strongly Connected Components

Let $G = (N, E)$ be an arbitrary, directed graph with nodes N and edges E . A set of nodes $N' \subseteq N$ is called a Strongly Connected Component (SCC) if for every node in N' there exists a path to all nodes in N' . Let $G = (N, E)$ now be an SCG with nodes N , arbitrary edges E and immediate edges $E_I \subseteq E$. The arbitrary edges include all edges of the model while the immediate edges include all instantaneous control-flow edges as well as all dependency edges of the model. An SCC in an SCG is a set of nodes $N' \subseteq N$ where for all nodes in N' there exists a path traversing only edges from E_I towards all nodes in N' . This means that an SCC never traverses any delayed edges. If there exists an SCC N' and two nodes $n_1, n_2 \in N'$, then these two nodes must have the same priority:

- ▷ If without loss of generality n_1 has a priority that is higher than the priority of n_2 , then n_1 must be scheduled before n_2 . However, there exists a path of immediate edges originating in n_2 and ending in n_1 , therefore n_2 can be scheduled before n_1 by the scheduler. If the two nodes had the same priority, their order of execution does not matter, and the scheduler can choose which node to perform first.

The detection of SCCs is done using an adapted version of *Tarjans algorithm* [Tar72] over the SCG, where the delayed edges are disregarded. Each node in the SCG is assigned exactly one SCC. All nodes that form a directed cycle are assigned the same SCC, while nodes that do not form a directed cycle, e. g., nodes without any outgoing edges or all nodes of an acyclic graph, form an SCC by themselves.

Figure 4.1 and Figure 4.2 depict two SCG models with a similar behavior. Both of them read an integer x and then perform a loop x times, calling a function `foo` each time with x as its parameter. However, the loop in Figure 4.2 contains a pause, breaking the immediate loop of Figure 4.1. The model in Figure 4.1 therefore takes exactly one tick to complete, while the model in Figure 4.2 takes x ticks. The SCC highlighted in purple in the left model is therefore not present in the right model anymore.

The runtime of the algorithm increases linearly based on the number of nodes and edges in the SCG. The result of the algorithm is a new, now acyclic graph consisting of nodes representing a set of nodes. The edges of this graph consist of all arbitrary edges that originated in one SCC and target another SCC. The internal control-flow edges of an SCC are discarded. The following steps will use this data structure and its acyclicity as their foundation.

4.1.2 Node Priorities

As mentioned earlier, a node must be executed before another node, if there exists a concurrent and non-confluent dependency originating in the first node and targeting the latter. Therefore, the priority assigned to the first node has to be higher than the one of the latter. To achieve that, all immediate edges are assigned a weight. Dependency edges receive a weight of 1, while immediate control-flow edges receive a weight of 0. Afterwards, the maximum weighted path originating in an SCC following all immediate edges is calculated. A depth-first search through all SCCs is performed to find this path for all SCCs. Each SCC and all nodes corresponding to the SCC are then assigned the length of the maximum weighted path as their `npr`: A number representing the importance of a node to determine the order of execution. A node instance of a node with a higher priority must be executed before a concurrent node instance of a node with a lower priority. If the two nodes are assigned the same node priority, their order of execution does not matter. Algorithm 1 describes the `npr` in detail. Figure 4.3 depicts the annotated SCG of the ABO-example as introduced in Section 2.1.4.

The depth-first search of the `npr` assignment traverses nodes until it reaches a sink. As the maximum weighted path originating in a node with no outgoing transitions will always have a weight of 0, the `npr` of all surface nodes as well as the final exit node is determined as 0. When the traversal continues backwards from the exit node, the nodes above also receive the `npr` 0 as their only outgoing path traverses towards the exit. When the backwards recursion reaches the `B = true` assignment in thread `HandleA`, its corresponding node is the source of a dependency depicted by a dashed green arrow targeting the conditional testing for the state of `B`. This conditional node has two paths towards sinks. One path that leads directly towards the surface node with an `npr` of 0 and another path towards the final exit. Since both paths do not contain any dependency edges, their weight and thus also the `npr` of the node is 0. As the node containing the `B = 0` assignment has only one further path with a weight of 0, its maximum weighted path is the one traversing through the dependency towards the conditional node and then towards one of the two sinks. Thus, the `npr` assigned to this node is 1. Its predecessor, the conditional testing for the state of `A`, now has four paths towards

4. Priority-Based Compilation

Algorithm 1 Calculating the *npr* of a list of SCCs, starting at the SCC containing the root node

```
1: function LONGESTWEIGHTEDPATH(currentSCC)
2:   dependencies ← currentSCC.outgoingDependencies
3:   neighbors ← currentSCC.successors
4:   prio ← 0
5:   for each d in dependencies do
6:     if ! (d.scc.visited) then
7:       d.scc.visited ← true
8:       LONGESTWEIGHTEDPATH(d.scc)
9:     end if
10:    prio ← max(d.scc.npr + 1, prio)
11:  end for
12:  for each n in neighbors do
13:    if ! (n.scc.visited) then
14:      n.scc.visited ← true
15:      LONGESTWEIGHTEDPATH(n.scc)
16:    end if
17:    prio ← max(n.scc.npr, prio)
18:  end for
19:  currentSCC.npr ← prio
20: end function
```

sinks. The first path leads directly towards the surface node, while the other three paths are the same paths as the $B = 0$ node just beginning at the conditional node. Thus, the maximum weighted path of the node takes the same path as the maximum weighted path of the $B = 0$ node with a weight of 1. The same then applies for all its predecessors. It is not necessary to traverse all paths originating in a node for every node, but only to take the maximum of all the *npr*s of its control-flow neighbors and the *npr*s of its dependencies plus 1, if those were already calculated before. If not all SCCs were visited by the depth-first traversal, another traversal is started at one of those unvisited SCCs.

It is further important that the *npr* of a node is *never* higher than the *npr* of its predecessor to prevent priority inversion, where a node with a lower *npr* is executed before a node with a higher *npr*. Figure 4.4 shows an imaginary, faulty *npr* assignment, where the statement $x = 0$ has a higher priority than its preceding node. If a thread entered the parallel region, thread B would be executed first until its completion, as its entry node has a higher *npr* than the entry node of its sibling thread. Only afterwards is thread A executed until its completion. This, however, means that the dependency from the node containing the statement $x = 0$ to the node containing the statement $x++$ is not considered, the latter statement is executed before the former, and the iur protocol is violated.

Algorithm 1 illustrates that each dependency edge and each control-flow edge is traversed exactly once to visit each node exactly once. If a node was previously visited, only its already

4.1. Priority-Based Compilation

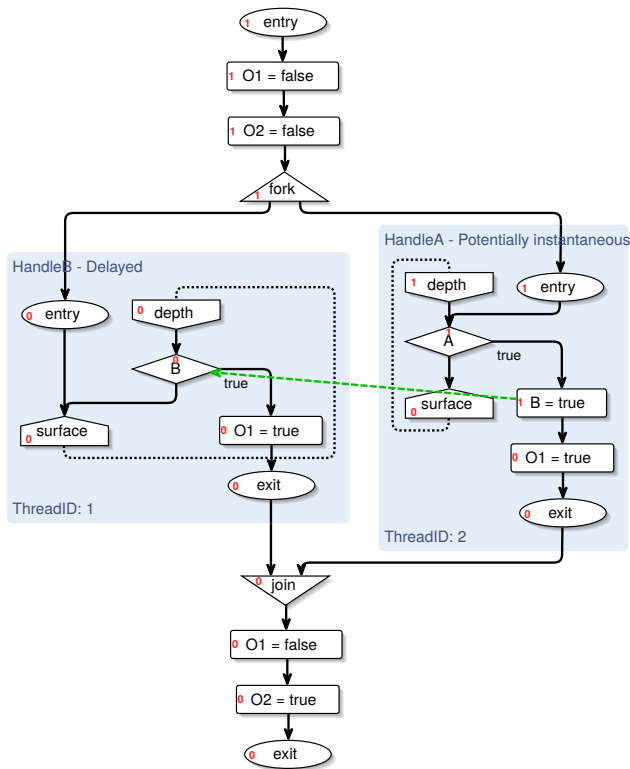


Figure 4.3. The SCG of ABO with nprs in red in the left or top corner of each node

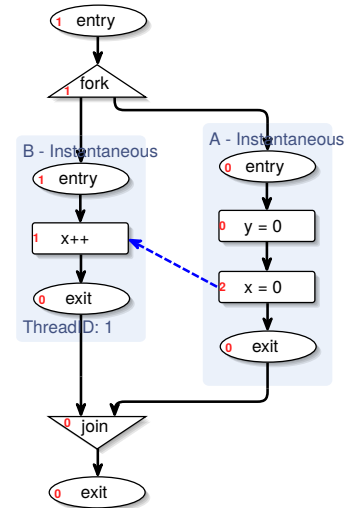


Figure 4.4. The SCG of a model with an incorrect npr assignment

assigned npr is checked. Thus, the algorithm scales linearly in the number of nodes and instantaneous edges in the previously generated graph of SCCs.

Schedulability Analysis

As explained in Section 2.1, not all programs should be accepted, e. g., if any write-write conflicts exist. A schedulability analysis is performed to check for invalid programs. If any positive weighted cycles in the graph exist, a conservative approach is to reject the model. A positive weighted cycle exists in an SCC, if and only if a cycle contains a concurrent, non-confluent dependency edge. Figure 4.5 shows such an SCG. The SCC – highlighted in purple – contains two dependency edges. Due to the iur protocol, the statement $O1 = 0$ has to be executed before statement $O1++$ and statement $O2 = 0$ before $O2++$. However, since the sequentiality necessitates that statements ought to be performed after their immediate predecessors in the SCG, any schedule of this program would result in a violation of the iur protocol. Since no definite order of execution can be determined using the iur protocol, the program is non-deterministic and therefore rejected by the scheduler. Figure 4.6 depicts another example that is prohibited by the schedulability analysis.

4. Priority-Based Compilation

Algorithm 2 Schedulability analysis of an SCG

```
1: function SCHEDULABILITY(SCCList)
2:   for each scc in SCCList do
3:     for each node in scc do
4:       dependencies  $\leftarrow$  node.dependencies
5:       for each d in dependencies do
6:         if d.concurrent && !(d.confluent) then
7:           if (d.type == WRITE_WRITE) || (d.target in scc) then
8:             return false
9:           end if
10:        end if
11:      end for
12:    end for
13:  end for return true
14: end function
```

Statement $O = 0$ has to be executed before statement $O++$ due to the dependency between their corresponding nodes. However, due to the immediate cycle crossing through the join-nodes and fork-nodes, $O = 0$ also has to be executed chronologically after $O++$. The node priority calculation is therefore unable to determine priorities for this example. This schedulability problem, however, only occurs *statically*, and Section 4.3 will argue for accepting similar programs. It can be concluded that the introduced schedulability analysis merely looks at the static structure of a program, not the dynamic execution. Figure 4.7 shows another example, where a problem arises in a part of the program that is unreachable. Since the conditional $x < 1$ can never evaluate to true due to the assignment of the variable x earlier, the dependency cycle can never exist during a run of the program. The program, however, is still rejected, even though it is otherwise schedulable and deterministic. Thus, according to the different classes of SC, the analysis checks whether a program is SIASC, as it allows immediate cycles in the control-flow, but cannot distinguish between cycles that only exist in the static layout of the program and cycles that occur during the execution of the program.

The schedulability analysis shown in Algorithm 2 traverses all SCCs exactly once and checks for all dependency edges of nodes inside these SCCs. Thus, its runtime scales linearly in the number of nodes in the SCG – as each node can be in only one SCC – and dependency edges in the generated graph of SCCs.

4.1.3 Priority IDs and Optimized Priority IDs

Assigning Thread Segment IDs

As mentioned in Section 2.1.2, two nodes in parallel threads must not have the same priority as it also uniquely identifies each thread. Therefore, Thread Segment IDs (tsIDs) are introduced. Each thread in the SCG, including the root thread, receives an identifier that uniquely separates

4.1. Priority-Based Compilation

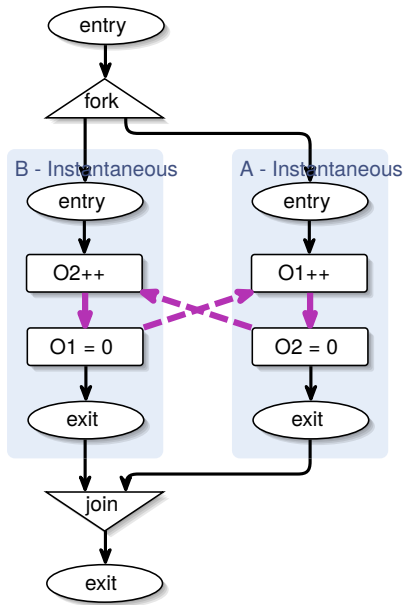


Figure 4.5. The SCG of a model with an immediate cycle containing dependency edges; The cycle is highlighted in purple.

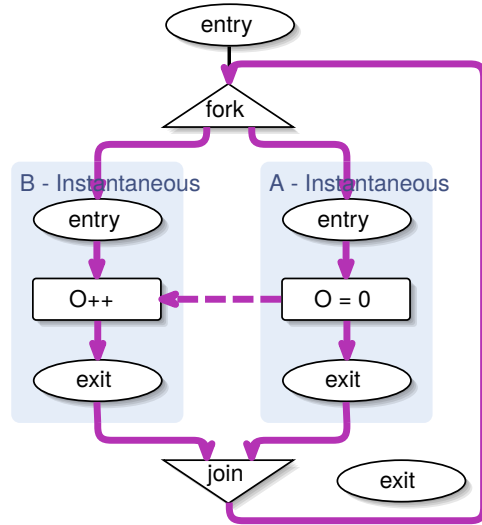


Figure 4.6. The SCG of another model with an immediate cycle containing a dependency edge that traverses a join and fork node; The cycle is highlighted in purple.

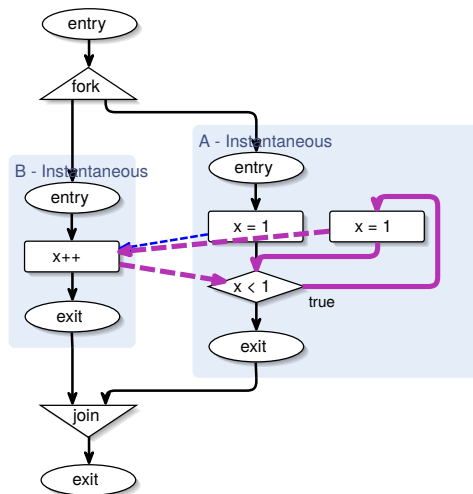


Figure 4.7. The SCG of a model with an unreachable immediate cycle containing dependency edges; The cycle is highlighted in purple.

threads if they are statically concurrent. Two sequential nodes with the same tsID can then be regarded as a part of the same thread. If two threads are not statically concurrent, they are allowed to have the same identifier to reduce the number of required priorities for all nodes.

4. Priority-Based Compilation

Algorithm 3 Assignment of tsIDs

```
1: function ASSIGNTHREADSEGMENTIDS(node, tsID)
2:   successors  $\leftarrow$  node.successors
3:   if !(node.visited) then
4:     node.visited = true
5:     if node instanceof Fork then
6:       sortedSuccessors  $\leftarrow$  successors.sortBy(npr)
7:       for each n in sortedSuccessors do
8:         tsID  $\leftarrow$  assignThreadSegmentIDs(n, findNextAvailabletsID())
9:       end for
10:    else
11:      for each n in successors do
12:        tsID  $\leftarrow$  assignThreadSegmentIDs(n, tsID)
13:      end for
14:    end if
15:    node.tsID  $\leftarrow$  tsID
16:  end if
17:  return tsID
18: end function
```

Before assigning the tsIDs, it is important to think about the way forks and joins are handled. One approach would be to have a designated *parent* thread that forks all its children and waits in parallel for the termination of all children. The parent thread then polls each tick whether its children have terminated before it may continue. This way, the assignment of tsIDs can equal the threads in the SCG and the tsID assigned to each node is dependent on its immediately enclosing thread. However, this approach would cause one thread per fork-join block that only polls for the completion of its child threads each tick. This causes a larger set of required priorities while it also causes more computational effort per tick. However, the number of priorities is limited as mentioned in Section 2.1.2. Instead, one can argue that the parent thread can assume the role of one of its child threads. This way, the number of required threads can be reduced by one for each fork in the program. When performing the fork, the tsID will then be transferred to one of the child threads and when the join is performed, one of the child threads will have to transfer its tsID towards the continuing thread. To minimize changes in priority at the start and end of the parallel statement, the forking or joining thread should be able to continue immediately after forking or joining respectively. Therefore, one of the threads with the highest entry npr inherits the tsID of the parent thread. Thus, the thread can immediately continue after the fork. Since the exit nodes of a parallel segment in contrast all have the same npr assigned, the lowest tsID of any thread is assigned to the thread after the join.

The actual tsID assignment as described in Algorithm 3 is then performed via a depth-first search over all control-flow edges starting at the root node. If the traversal reaches a fork

4.1. Priority-Based Compilation

node, the next nodes are visited ascending in the order of their npr s. Thus, the entry node with the highest npr will be visited last. When entering a thread, the next available, unused $tsID$ is calculated. It will be passed on until a final exit node or an already visited node has been reached. This node is then assigned this $tsID$. The depth-first search subsequently backtracks, applying the same $tsID$ to all nodes on its path, until a node with multiple outgoing control-flow edges is reached. If the node is a conditional, the depth-first traversal continues on the alternative path until it reaches a previously visited node and backtracks from there again. As a thread may only have one exit node, this path will at some point lead to an already visited node. If the node with multiple outgoing control-flow edges is not a conditional, a fork was reached. At a fork node, all threads must be visited before backtracking further. Before traversing each thread, the next available $tsID$ is found again to assign to this thread. After all threads were traversed, the fork node inherits the $tsID$ of the child thread with the largest entry npr as this node was visited last. If multiple threads with the same highest npr exist, the forking thread receives the highest $tsID$ of those threads. This guarantees that the forking thread assumes the role of one of the children, while the transition happens without a change in priority. Since no other thread can have a higher priority, as all sibling threads were assigned lower $tsID$ s and/or npr s, and no other parallel thread exists that preempts the currently active thread, the thread may continue without any interference. Thus, no dispatcher will need to be called. Afterwards, the depth-first search backtracks until the next fork, conditional or initial entry node is reached.

In the ABO example from Figure 4.3, the traversal starts at the initial entry node at the top and continues until the fork node. The highest entry npr of any following entry node is 1 in the `HandleA` thread, thus only `HandleB` may be traversed first. The traversal continues in this thread across the pause and into the conditional checking for B. The traversal may then continue on the true path and finally reach the final exit node after exiting `HandleB`. The backtracking then starts by applying the $tsID$ 1 to all nodes on the path starting at the final exit node. When the backtracking reaches the conditional node, the alternative path is checked and, since it was already visited, ignored. The backtracking continues applying the same npr until the entry node of thread `HandleB` and the previous fork node is reached. Now, the next thread is visited, but with a higher $tsID$ -counter. When the depth-first search reaches the exit node of `HandleA` and recognizes that the following join was already reached, the new $tsID$ of 2 is applied to this exit node. The backtracking then applies this $tsID$ to all nodes on the path analogously to the thread `HandleB`. When the fork is reached again, no further threads can be visited, thus the node receives the $tsID$ 2 as well. The backtracking then continues without interruption until the initial entry node. All nodes on the path also receive the $tsID$ 2. As the model in Figure 4.3 shows, the $tsID$ s are visualized in the bottom left of a thread. The $tsID$ of the surrounding root threads can be deduced from the threads if they are not visualized themselves.

As mentioned earlier, one of the child threads has to perform the join and continue afterwards. The priorities of the threads determines which child will perform the join. If thread t joins and there exists a thread t' with $t'.priority < t.priority$, the check for the conclusion of all siblings will happen *before* the termination of t' . Thus, t would not be able to recognize

4. Priority-Based Compilation

that all siblings are terminated during the tick and would not continue after the join even though the join should be performed in this tick. The joining thread therefore must be the sibling with the lowest priority at its termination.

It is important to note that, with the exception of the forking sibling, the order in which the tsIDs are assigned to the child threads is arbitrary. The tsID is merely supposed to solve ties between two threads with the same npr. If they already have the same npr, their order is insignificant for SC-schedulability as their nodes are confluent. Their order is further unimportant for any optimizations done by the C compiler or most other code optimizations, since there is always a call to the dispatcher at changes of priorities and at the termination of a thread. The macros in Listing 2.1 show that to access the label of the jump towards the next continuation at the end of the dispatcher in line 27 the memory has to be accessed. Due to the statement `goto *_pc[_cid]`, no jump prediction can be made and no pre-caching can be done by a scheduler, as the `_cid` field is overwritten in the statement directly before the `goto` in the *bit scan reverse* instruction. It might be advantageous, however, if a thread that lowered its priority continues directly afterwards, such that some parts of the following code might still be loaded. As an example, if there are only multiple concurrent threads that span over only one tick, it might be of advantage to assign the tsIDs in an descending order depending on their order in the program. This way, if some code has been pre-cached due to the length of the cache-lines, the next thread to be executed after one thread terminated might still be in the cache. This may require much more extensive analysis of the program for larger programs, however. Additionally, if the execution of two threads spans across multiple ticks, the code that is actually executed concurrently may be far apart from each other in the code. Between those two code fragments lies presumably the behavior of parts of the first and second thread. Thus, when the tick in the first thread is loaded to the cache, the code of the second part will presumably not be pre-loaded together with the former.

The assignment of tsIDs traverses each node and each control-flow edge exactly once. Thus, it is linear in the number of nodes and control-flow edges.

Calculating the Priority IDs

After the tsIDs were assigned, the nprs and tsIDs must be combined to accommodate the SCL_P macros. The resulting priorities, the prioIDs, must be unique across all nodes that are pairwise statically concurrent. The actual calculation of the prioIDs of node n is as follows:

$$n.prioID = (t \times n.npr) + n.tsID$$

where t represents the total number of tsIDs across the whole program. Provided that all statically concurrent nodes belong to threads that have different tsIDs, these nodes will all receive unique prioIDs:

Let $n_1, n_2 \in N$ be two statically concurrent nodes with $n_1.prioID = n_2.prioID$ but also with $n_1.tsID \neq n_2.tsID$. Then, since the number of tsIDs is static across the model, for some $m \in \mathbb{N}$ with $m > 0$ must hold $n_1.npr + m = n_2.npr$ without loss of generality. Then:

$$\begin{aligned}
(t \times n_1.\text{npr}) + n_1.\text{tsID} &= (t \times n_2.\text{npr}) + n_2.\text{tsID} \\
(t \times n_1.\text{npr}) + n_1.\text{tsID} &= (t \times (n_1.\text{npr} + m)) + n_2.\text{tsID} \\
(t \times n_1.\text{npr}) + n_1.\text{tsID} &= (t \times n_1.\text{npr}) + (t \times m) + n_2.\text{tsID} \quad | \quad -(t \times n_1.\text{npr}) \\
n_1.\text{tsID} &= (t \times m) + n_2.\text{tsID}
\end{aligned}$$

Since $m > 0$ and $n_2.\text{tsID} > 0$ due to their definitions, $n_1.\text{tsID} > t$ must hold. However, since tsIDs are assigned in an ascending order without gaps between them, the tsID of a thread cannot exceed the maximum number of threads. Thus, there is a contradiction. All prioIDs are therefore unique for all nodes that are pairwise statically concurrent.

Optimizing the Priority IDs

This approach may produce unnecessarily high prioIDs, while many prioIDs will remain unused. If there are five parallel threads t_0 to t_4 , and all nodes of thread t_n had a npr of n , then thread t_0 would be assigned the prioID 1, t_1 6, t_2 12, t_3 18 and t_4 24, leaving 19 prioIDs unused. To minimize this, the prioIDs are reduced towards the optimized Priority IDs (prioIDs) by simply removing unused prioIDs and compacting the rest while retaining the order. In the following, prioIDs will be used for the optimized prioID.

Figure 4.8 shows the prioID assignment of the ABO example. The prioIDs are highlighted in blue to the right of each node while the nprs are still visible in red to the left of each node. The tsIDs of the threads HandleA and HandleB are shown in grey to the bottom left of the threads. A small extra node containing the prio-statement has been added between all pairs of successive nodes within the same thread with two different prioIDs.

While it is important that the npr never rises during a tick to prevent priority inversion, the prioID does not have this restriction. This means that a thread may be allowed to raise its tsID (as the npr must not rise and the number of tsIDs cannot change) if it is necessary. Figure 4.10 depicts the annotated SCG of a modified ABO example with only one output O. The corresponding SCCharts can be seen in Figure 4.9. Instead of finishing after receiving A and B, it restarts directly from the start. The threads themselves do not change the values of any outputs or other variables anymore, only after the completion is the output set to true. It is further important that HandleA is not immediate anymore. As mentioned earlier, the forking thread receives the highest tsID of the threads with the highest prioID. Since both child threads have the same entry prioID, the forking thread receives the tsID 2. Also, the joining thread inherits the lowest tsID of the exit nodes preceding it so the scheduler does not need to be called after performing the join and the thread can simply continue after the join. Thus, the join node inherits the tsID 1. However, since there exists a control-flow path from the join node towards the fork node, the tsID has to rise at some point, causing the prioID to rise as well. During the tsID assignment, after the $O = \text{true}$ node is visited, traversal tries to visit the next node. Since this node was previously visited, the traversal stops and applies the tsID to

4. Priority-Based Compilation

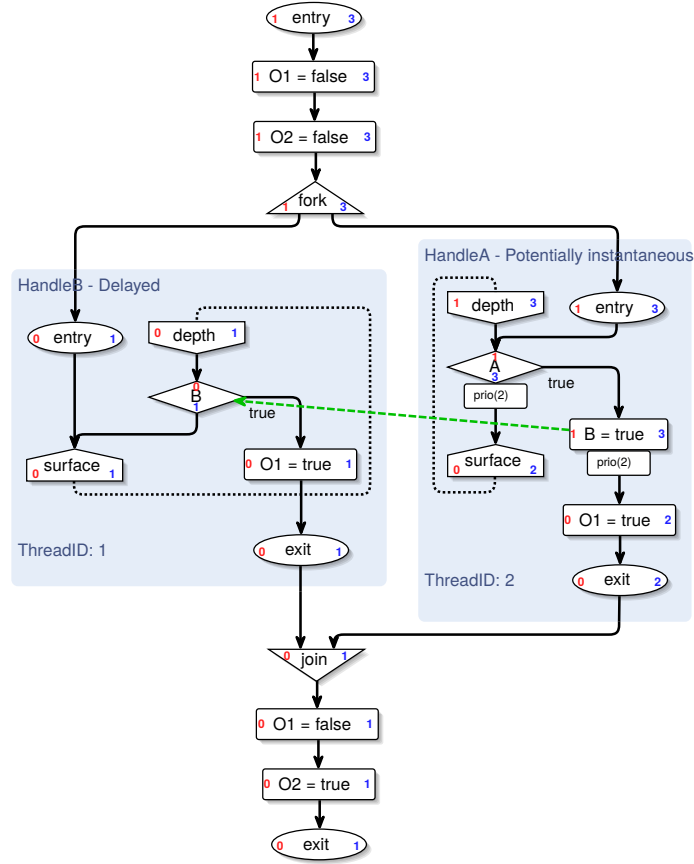


Figure 4.8. The annotated SCG of ABO; The prioIDs in blue to the right or bottom of a node

the $O = \text{true}$ node. The $O = \text{false}$ node on the other hand receives its tsID from its successor.

As only the tsID and therefore the prioID of the thread rises while the npr remains untouched, the thread cannot be preempted by another thread with a lower prioID . The following calculations will use the non-optimized prioIDs . Assuming there existed another parallel thread containing a node n with $n.\text{npr} < \text{join}.\text{npr}$, but with $n.\text{prioID} > \text{join}.\text{prioID}$. Let t be the total number of threads in the program. Then:

$$\begin{array}{rcl}
 n.\text{prioID} > \text{join}.\text{prioID} & & | \text{ Def. prioID} \\
 t \times n.\text{npr} + n.\text{tsID} > t \times \text{join}.\text{npr} + \text{join}.\text{tsID} & & | -t \times \text{join}.\text{npr} \\
 t \times (n.\text{npr} - \text{join}.\text{npr}) + n.\text{tsID} > \text{join}.\text{tsID} & & | n.\text{npr} < \text{join}.\text{npr} \\
 t \times (n.\text{npr} - (n.\text{npr} + 1)) + n.\text{tsID} > \text{join}.\text{tsID} & & | \\
 t \times (-1) + n.\text{tsID} > \text{join}.\text{tsID} & & | \text{ Def. tsID} \\
 -t + n.\text{tsID} > 0 & & | +t \\
 n.\text{tsID} > t & &
 \end{array}$$

4.1. Priority-Based Compilation

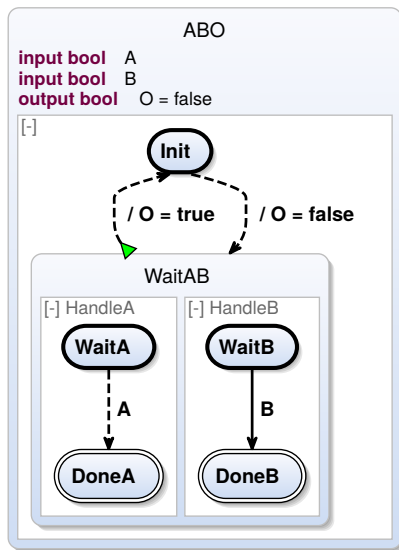


Figure 4.9. The SCCharts of a modified cyclic ABO

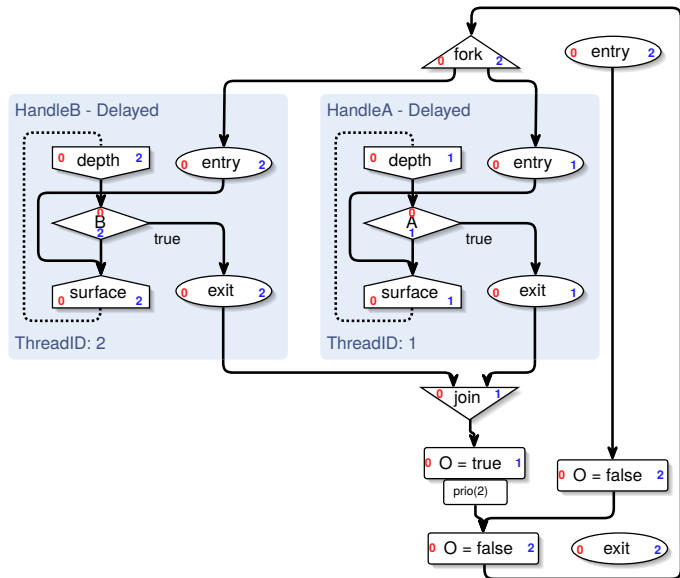


Figure 4.10. The SCG of a modified ABO with a rising prioID

Again, however, the tsID of a thread cannot exceed the maximum number of threads. Therefore, the rising prioID is allowed, and the program is accepted.

During the assignment of prioIDs, each node is visited exactly once to calculate its prioID and once more to assign its prioID. To calculate the optimized prioIDs, the list of all prioIDs has to be traversed once. Since there cannot be more prioIDs than nodes in the SCG, all steps run linear to the number of nodes in the SCG.

4.1.4 C Code Generation

As Table 2.1 in Section 2.1 shows, most statements can be translated one-to-one to the modified C code. For this, the code generation performs a modified, hierarchical depth-first traversal of the SCG. A normal depth-first traversal is performed, yet each new parallel block, as delineated by a fork and a join node, has to be traversed completely before the traversal continues after the join. Thus, a parallel block is not interrupted by the following code, thus improving reading comprehension of the resulting code. If the traversal enters a node with multiple incoming control-flow edges for the first time, a label is created. If at a later point the same node is entered, it will not be translated again, but a goto-statement is inserted specifying the label as the continuation point of the program.

Conditionals are translated using simple if-else blocks. However, predicting when the branches merge again is difficult as the model is not acyclic. Therefore, the program may even return to a point that was previously already translated. Therefore, the two branches are simply translated until they either reach an exit node, a surface node or a node that

4. Priority-Based Compilation

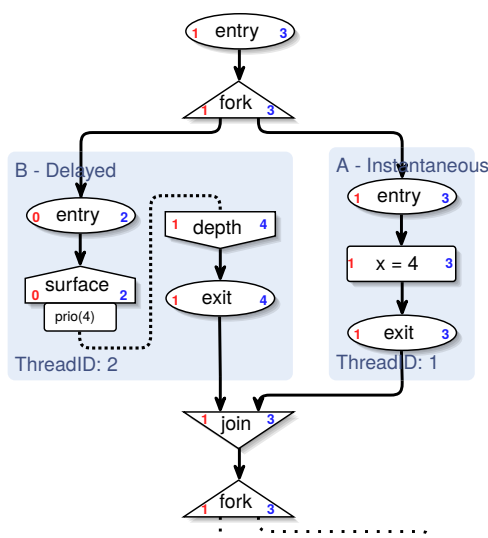


Figure 4.11. The SCG of a model with problematic join priorities

was previously reached. Surface nodes and their corresponding depth nodes are translated to pauses. A prio-statement containing the prioID of the depth node is inserted before the pause-statement, if the prioID of the depth differs from the prioID of the surface. If it is not inserted before the pause but afterwards, the thread would be activated during the next tick with the lower prioID of the previous tick and cannot raise its prioID before any other threads with a higher prioID than the initial prioID have executed, thus causing priority inversion. Assignments are simply translated using their contents.

At the beginning of the translation of each node, its prioID is checked. If its prioID is different from the prioID of its successor, a prio-statement is introduced with the new prioID as its argument. This has to be performed before checking whether this node was already visited to ensure that the prioID is lowered before a possible jump towards the next continuation. The previously visited node is saved using a stack. After the prioID of the previous node was checked, the current node writes itself on the stack. After finishing its and its successor's translation, the node pops the stack, deleting itself from it.

When the traversal finds a fork node, it first needs to find out, which thread has the highest entry prioID and the lowest exit prioID as mentioned in Section 4.1.3. To enhance readability and to save a jump, the thread with the highest entry prioID is translated first. The thread with the lowest exit prioID must be translated last to allow the join to work as intended. Due to the assignment of tsIDs and since all exit nodes will have the same npr inherited from the join node, the forking thread cannot be the joining thread. Therefore, no extra precautions need to be made for this special case. For the remaining threads the order does not matter. However, it looks more pleasing, if the threads are arranged in descending or ascending order according to their entry prioID. The current implementation orders them in descending order.

When performing joins, the joining thread has to check whether any of its siblings is

4.1. Priority-Based Compilation

still enabled. The `joinN`-macro therefore accepts N `prioIDs` and looks them up in the enabled scalar. Since the joining thread is always the thread with the lowest `prioID` during the exit, all parallel threads that exit during the tick have already performed their exit. To check for their completion, the joining thread therefore has to check for the enabled status of the exit `prioIDs` of its sibling threads. However, it can be possible that the joining thread exits in a tick, while a parallel thread merely pauses. While surface nodes will always have the `npr` 0, it might be possible for an exit node to have a higher `npr`. Figure 4.11 depicts a model where after the join is performed, another parallel statement starts, whose actual contents are not important to the problem. Due to the dependencies in this parallel statement, however, the `npr` of the fork is 1. Thus, this `npr` is propagated upwards to the previous join and therefore also to the exits. Since thread B is delayed due to a pause, the upwards propagation of the `npr` ends at the depth node and the surface and entry node have an `npr` of 0. Thus, when the parallel statement starts, thread A begins due to its higher `npr` at its entry node compared to thread B. The thread will then run into the join node and check for the state of its sibling thread. If the thread only checked for the `prioID` in the exit node, the join would be performed, as the thread with the `prioID` 4 is currently not enabled. Only the thread with the `prioID` 2 is currently enabled and active, as it did not reach the pause-statement yet. Thus, the joining node must also check for the enabled state of at least the `prioID` 2 in this case.

One solution to this problem would be instead of checking for static `prioIDs` of exit nodes of potentially active sibling threads, to check for the actual currently dynamically assigned `prioIDs` of the siblings. This requires the program to save the parent-child relation of the threads for each parallel region. Thus, the parent thread would save all the current `prioIDs` of its children (or siblings) and updates it each time any sibling changes its priority. At a join it would then check for the enabled state of all its current siblings. However, this approach requires a lot more information to be saved during the execution of the program as well as additional actions to be performed during each change in priorities. Alternative approaches that require more effort for the compiler instead of the execution may be better for the resulting code.

One easy solution doing exactly this would be to check for all occurring `prioIDs` in all sibling threads of the joining thread. However, in large models with many changes in priorities, this may lead to as many priorities to check as there are dependencies, causing unnecessary overhead. For now, this only requires the creation and subsequent comparison to a bitmask. Later, however, this will not only be the case. To therefore minimize the strain on the scheduler, only those `prioIDs` are checked that may theoretically be enabled at the time the join-macro is executed. Those include all `prioIDs` of exit nodes, all `prioIDs` of depth nodes, whose `prioID` of the corresponding surface node is higher than the exit `prioID` of the joining thread, the `prioID` of those nodes, whose `prioID` is lower than the exit `prioID` of the joining thread, but whose `prioID` of the predecessor is still higher than the exit `prioID` of the joining thread and finally the `prioID` of threads that already started with a lower `prioID` than the joining thread. The exit nodes must be checked. If a thread has run into a surface node before the check for the join, its `prioID` in the surface node must be higher than the exit `prioID` of the joining thread. Since

4. Priority-Based Compilation

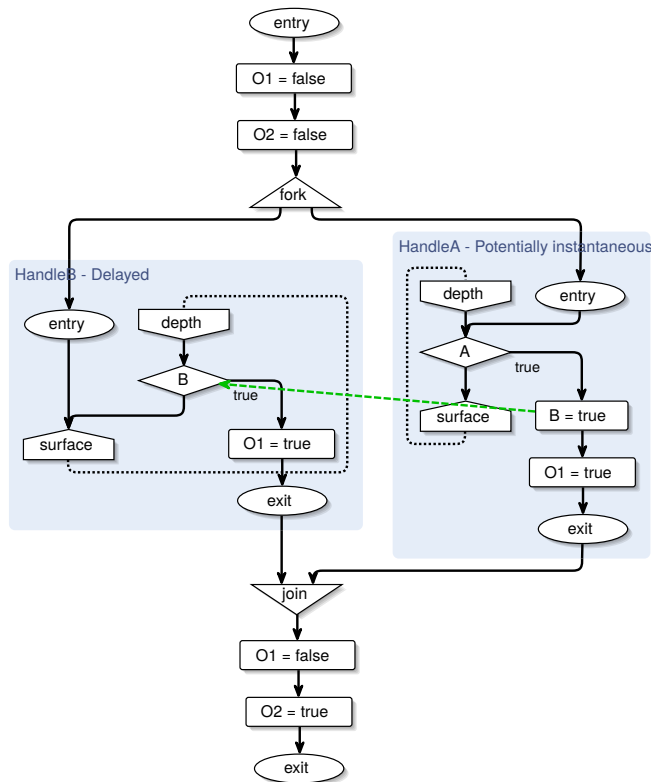


Figure 4.12. The SCG of ABO again, for comparison purposes

```

1 int tick() {
2   tickstart(3);
3
4   O1 = false;
5   O2 = false;
6   fork1(HandleA, HandleB, 1) {
7     HandleA:
8     if (A) {
9       B = true;
10      prio(2);
11      O1 = true;
12    } else {
13      prio(2);
14      prio(3);
15      pause;
16      goto HandleA;
17    }
18  } par {
19    HandleB:
20    pause;
21    if (B) {
22      O1 = true;
23    } else {
24      goto HandleB;
25    }
26  } join1(2, 3);
27  O1 = false;
28  O2 = true;
29
30  tickreturn();
31 }

```

Listing 4.1. Generated SCLP code of the ABO example

the thread has then already changed its prioID to the prioID after the pause, the joining node must check this prioID. This is illustrated in Figure 4.11 in thread B. If thread B were to execute before thread A, after it ran into the surface node, its prioID would be raised to 4. Thus, the join would have to additionally check for 4. If a thread has not run into a surface or exit node when the the joining thread performs the join, the thread itself is still active. Additionally, its prioID must be lower than the prioID of the joining thread. Otherwise, this thread would execute. Thus, the join must also check for the prioIDs of threads whose prioID at one point sinks below the prioID of the joining thread at the moment of the join. This again can be exemplified by the example in Figure 4.11 as explained earlier.

Since the code generation is performed by simply pattern matching each node and translating them as described previously, each node is visited exactly once and all control-flow edges are traversed exactly once. Thus, this step is again linear in the number of nodes and control-flow edges in the SCG. Since all steps so far have been linear in the nodes and some subset of edges in the SCG, it can be concluded that the whole compilation of code from the SCG runs in time linear to the number of nodes and arbitrary edges in the provided SCG.

Listing 4.1 shows the result of the C code generation for the ABO-example with the

4.1. Priority-Based Compilation

corresponding SCG next to it in Figure 4.12. The tick begins with the `tickstart` macro. When called for the first time, it initializes the first thread with the `prioID` passed as the argument as well as the set of enabled threads and implicitly the set of active threads. In any following tick, the `tickstart` macro simply sets all enabled threads to active and calls the dispatcher to jump towards the continuation of the thread with the then highest `prioID`. As the boolean variables `O1` and `O2` are set to false at the beginning of the first tick, these instructions follow the `tickstart` in the generated C code and are only executed in the first tick. Afterwards, as the two regions `HandleA` and `HandleB` happen in parallel in the SCG, a fork is called. This macro first contains the label the forking thread will jump to and then the label of the second thread as well as its initial `prioID`. There are therefore two different active and enabled threads now – one with a `prioID` of 3 starting at the label `HandleA` in line 7 and one with a `prioID` of 1 starting at the label `HandleB` in line 19. Afterwards, the thread with the `prioID` 3 – as this is the highest, currently active `prioID` – is executed. Since the SCG has a conditional node checking for `A`, there is a corresponding if-statement in the C code starting in line 8. If `A` is true, both `B` and `O1` are set to true as well. Between those two assignments, the `prioID` of the thread is lowered to 2 in line 10. Since the assignment of `B` has to be executed before the check in line 21 in the sibling thread, the `npr` of all nodes preceding the assignment must be higher as seen in Figure 4.3. Afterwards, this higher `npr` is not required anymore and the `prioID` is therefore lowered. Subsequently, the thread terminates, by continuing after the if-block in line 17 and running into the `par`-statement, which disables the currently running thread. If `A` evaluates to false earlier, the thread lowers its `prioID` in line 13, as the longest weighted path towards a surface has a weight of 0 and the following nodes therefore have an `npr` of 0. As the thread runs into the pause, it must raise its `prioID` again to the `prioID` of the depth in line 14. It then is deactivated until the next tick through the `pause`-macro. The thread with the `prioID` 1 then activates after its sibling thread has deactivated (either by pausing or by terminating) and starts at the label `HandleB` in line 19. As the SCG runs into a surface node, however, a `pause`-statement follows the label immediately.

In the following tick, if the thread with the `prioID` of 3 did not terminate earlier, it would continue directly after the `pause`-statement in line 15, jump to the `HandleA` label and continue from there identical to the behavior during the first tick. After this thread deactivates in the second tick or if it was never enabled, thread 1 continues after the pause in line 20. The pause of the `HandleB` region in the SCG may now be taken. Thus, the code checks for the value of `B` in line 21. If `B` was false, the thread would jump towards the `HandleB` label in line 19 and then behave identical as in the first tick. If `B` was true, however, `O1` is set to true and the thread continues after the if-block with the `join`-macro. As mentioned earlier, the `prioIDs` in the `join` macro correspond to the possibly still enabled `prioIDs` of sibling threads and their exit `prioIDs`. If the sibling thread paused, its `prioID` before the pause would be higher than the joining threads `prioID`, thus the `prioID` of the depth node needs to be added to the join. If one of those `prioIDs` is still enabled, the joining thread will poll each following tick for the enabled state of the `prioIDs` 2 and 3. If at one point both are disabled, the thread continues by setting `O1` to false and `O2` to true before ending the execution of the program by executing the `tickreturn` macro.

4. Priority-Based Compilation

The transformation of ABO exemplifies that there are multiple points, where the program can be optimized. The lines 13 and 14 alone could simply be removed without changing the program. Section 4.2 will cover the topic of optimizations in detail. The next part though will first cover problems with large models.

Extending the prioID-space

The original macros as mentioned earlier work by using an integer as a scalar to identify threads. This works as long as the number of threads does not surpass the length of an integer on the system. If it does, the macros will try to access a non-existing bit by right shifting on the integer until all bits are set to zero. Therefore, changes were made to adapt to programs with more than 32 or 64 priorities by using an array of integers instead of only one integer, but using it analogous to the scalar. Fortunately, the original macros already implemented most low-level bitwise operations for arrays of integers as well. Once the maximum prioID exceeds the WORD_BIT constant determined by the size of an integer on this system, these alternate macros activate automatically and the standard macros as illustrated in Listing 2.1 deactivate. Merely the u2b-macro, which converts a priority to a bit representation of this number, was not implemented for arrays. Thus, at first a u2b-macro for arrays that uses the same sequence of operations for all accesses was created: Convert the prioID to the scalar/array, access the enabled or active array depending on the current use and compare, write or read from it.

However, the only functionality requiring the u2b-macro is in the join macro to check whether a thread or a number of threads is currently enabled. Rewriting the u2b-macro to transform an integer to an array-based representation of the integer which then can be checked against the enabled-array would be ineffective. Each time n priorities have to be checked for their state, the check would either have to make n comparisons between the enabled-array and the array-representations of the priorities, or merge n priority-arrays together. Both of these comparisons require a lot of unnecessary calculations. Each array-representation of a priority contains only one entry where a bit is set. Thus, the merge or the comparison with the enabled-array compares mostly empty integers. Instead, only the significant entry in the enabled-array needs to be compared. If the priority to be checked is for example 234 with an integer size of 64, the comparison only needs to check the 42nd bit of the fourth entry of the enabled-array. This can be calculated by dividing the priority by the size of the integer. The result + 1 is the entry of the array and the remainder is the bit that has to be checked in this entry. Overall, this check remains linear in the size of the join. Each priority to be joined needs to be checked against the enabled array as described previously and afterwards, all results need to be combined. If any priority is contained somewhere in the enabled-array, the result comes back negative. Otherwise, the result comes back positive.

4.1.5 Java Code Generation

The generation of Java code is based on SJ. Similar to the C code generation, the code is created by translating an SCG node for node. Unlike the original generation, however, SJ

4.1. Priority-Based Compilation

```
1 public ABO() {
2   super(ABOEntry, 3, 3);
3 }
4
5 public final void tick() {
6   setupTick();
7   while (!isTickDone()) {
8     switch (state()) {
9       case ABOEntry:
10        O1 = false;
11        O2 = false;
12        fork(HandleB, 1);
13        gotoB(HandleA);
14        break;
15       case HandleA:
16        if (A){
17          gotoB(_L3);
18        } else {
19          gotoB(_L5);
20        }
21        break;
22       case _L3:
23        B = true;
24        prioB(2, _L4);
25        break;
26       case _L4:
27        O1 = true;
28        termB();
29        break;
30       case _L5:
31        prioB(2, _L6);
32        break;
33       case _L6:
34        prioB(3, _L7);
35        break;
36
37       case _L7:
38        pauseB(_L8);
39        break;
40       case _L8:
41        gotoB(HandleA);
42        break;
43       case HandleB:
44        pauseB(_L0);
45        break;
46       case _L0:
47        if (B){
48          gotoB(_L1);
49        } else {
50          gotoB(HandleB);
51        }
52        break;
53       case _L1:
54        O1 = true;
55        gotoB(_L2);
56        break;
57       case _L2:
58        if (!join(2) || !join(3)) {
59          pauseB(_L2);
60          break;
61        }
62       case _L9:
63        O1 = false;
64        O2 = true;
65        termB();
66        break;
67     }
68 }
```

Listing 4.2. Generated SJ code of the ABO example

already supports models whose prioIDs exceed the length of an integer. Further, due to the adoptions mentioned in Section 2.1.3, most code generation functions analogously. Since Java does not use macros or gotos, each time the scheduler is called the inner switch-structure has to be broken and the surrounding loop has to restart. Thus, the join method needs to be surrounded by a case-label and a break. Each prioB-method, which replace the prio-macros in SCL_p , has to specify the label of its continuation as a parameter, as Java cannot generate and expand them automatically, and must be followed by a break as well to call the dispatcher. Instead of par-macros, threads are terminated using the termB-method in SJ. This also includes the final termination of the root thread. No tickstart or tickreturn macros are required.

Since the Java code generation performs the same depth-first search as the C code generation and only differs due to the translated code, the Java code generation also runs in time linear to the number of nodes and edges in the SCG.

Listing 4.2 depicts the translation of the ABO example from Figure 4.8 to SJ code. The constructor before the tick-method tells the scheduler that the program starts at the label

4. Priority-Based Compilation

ABOEntry with priority 3 and that the highest priority in the program is 3. The tick-method then begins by setting up the program and running into the while-loop. As long as any thread is enabled, the program repeats the following switch-statement. The conditional of this switch-statement contains the scheduler. Depending on the thread with the highest currently active priority, the state()-method returns a label corresponding to that thread. Otherwise, the resulting code functions analogously to the C code. The ABOEntry label first initializes the two output variables and then forks the two threads HandleA and HandleB. HandleA then continues in line 15 by checking for the state of the input A. If A is true, the control-flow continues at label `_L_3` in line 22 by setting B to true and subsequently lowering its priority. As described earlier, this means that a new label is required. `_L_4` continues by setting `01` to true and finally terminating, analogous to the par-statement in the SCL_P code. If A was false earlier, then the control-flow would have continued in line 30 by first lowering its priority, then raising it again and pausing in line 37. In the next tick the thread continues at label `_L_8` before jumping back to the label HandleA. This again happens analogously to the SCL_P code in Listing 4.1, just with more explicit breaks in the program. The thread HandleB is handled in the following lines. It first starts by pausing before continuing in the next tick by checking for the state of B. If B is true, the control-flow continues at label `_L_1` by setting `01` to true before jumping to the join at line 57. If B is false, the code generation recognizes that the following behavior of the thread is identical to the one starting at the label HandleB. Thus, it jumps towards this label. At the join in line 57, the thread probes for the current state of its sibling thread. As long as it is still active, the thread pauses and jumps towards its own label. Otherwise, the thread continues – the unnecessary break was omitted at this place – at the label `_L_9` by setting `01` to false and `02` to true and terminating.

The original implementation of SJ used parent-children relations as well as a polling parent waiting for its children to terminate before executing the join. Due to the additionally required priorities as mentioned in Section 4.1.4, the fork and join macros were adapted for SJ as well such that they work identical to the way they work in the SCL_P approach and the child-parent relation was removed.

4.2 Optimizing the Node Priority Assignment

Changing the priority during the execution of a thread is costly. Every time the priority changes from one node to the next, the dispatcher has to memorize the continuation of the currently executing thread and its corresponding priority, deactivate the old priority, activate the new priority, memorize the state of the current thread, determine the thread with the highest priority, activate it and continue at its continuation point. This is shown in the expansion of the prio macro in line 30 in Listing 2.1. If the number of prio-statements in a program can be reduced, unnecessary overhead can be reduced from the execution. Therefore, optimizations were developed to minimize the amount of required prio-statements in the translated program.

The goal of the first solution was to reduce the amount of unnecessary context switches

4.2. Optimizing the Node Priority Assignment

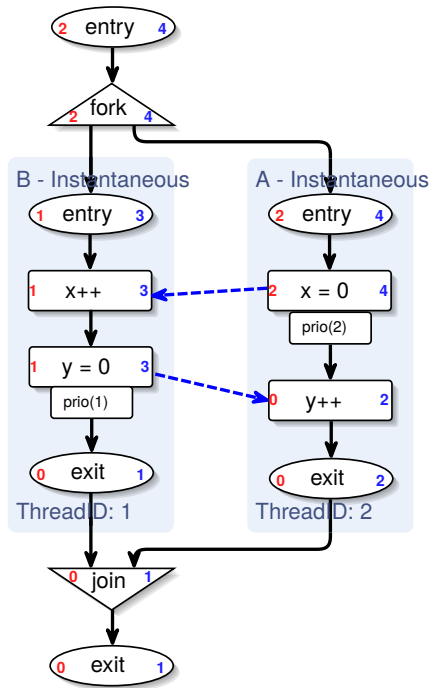


Figure 4.13. SCG model, where the assignment of tsIDs causes an unnecessary context switch from thread B to thread A

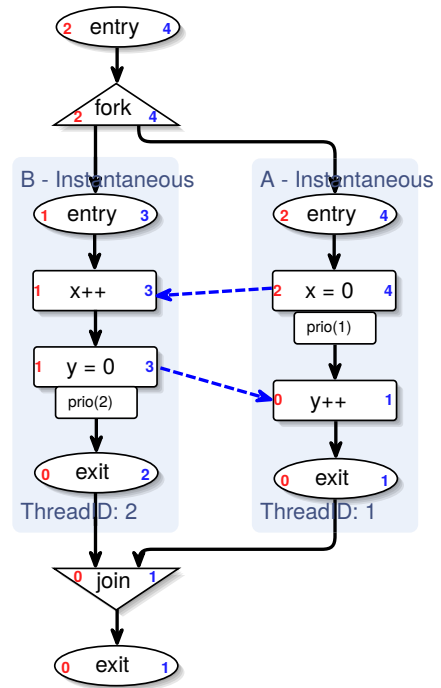


Figure 4.14. The same SCG as in Figure 4.13 but with an assignment of tsIDs where the context stays in B after the prio(2) statement due to the switched tsIDs as seen at the bottom of the threads.

as depicted in Figure 4.13. The SCG representation of the model shows how an adverse assignment of tsIDs causes the prioIDs of thread B to sink from 3 to 1, which causes a context switch from thread B to thread A since thread A has a higher prioID now. This context switch, however, is not necessary since the execution of the exit node in thread B does not depend on any nodes in thread A. Figure 4.14 shows that with a different assignment of tsIDs, the execution does not shift from thread B to thread A after the $y = 0$ assignment. However, this improved assignment of tsIDs will not free the model from any potentially unnecessary prio-statements, as these are dependent on changes in the prioIDs, which themselves are dependent on both the tsID and npr of a node. Figure 4.14 exemplifies that changing the tsIDs in this example does not change anything in the number of changes in priorities. Since there is no difference between executing a context switch from one thread to another due to a change in the prioIDs and continuing a thread with a new prioID, it is not sufficient to find a better assignment for the tsIDs. The only point of this assignment is that – conceptually – the control-flow stays in the same thread, even though this is not the case in the realization.

The second optimization aimed to let the original npr stay the same and only to correct the results afterwards, since the original approach is simple and easy to understand. As can

4. Priority-Based Compilation

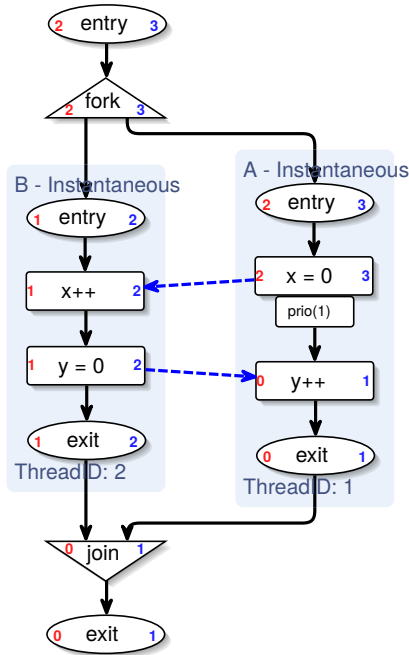


Figure 4.15. Same SCG model as in Figure 4.13 but without the unnecessary prio-statement due to propagation of nprs.

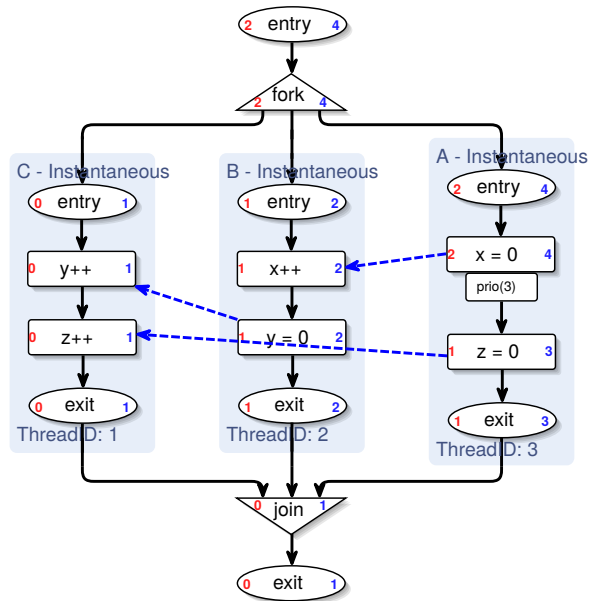


Figure 4.16. Example SCG model where the propagation of nprs cannot reduce all unnecessary context switches.

be seen in Figure 4.13, context switches may happen at the end of threads or before pauses, since all exits of a thread are always assigned the same npr and all surface nodes are always assigned the npr 0. In these situations, the tsID decides which thread continues next. Hence, the final change in priorities to a thread causes a dispatcher call that might not be necessary as all threads except one will terminate at the end of the fork-join block. Thus, the npr of the node before the final prio-statement can be propagated downwards towards an exit or depth node. If the paths of multiple control-flows meet before reaching an exit or depth node, the npr of the thread with the lowest npr is propagated downwards to avoid rising priorities. As can be seen in Figure 4.15, this approach reduces the amount of unnecessary context switches. The prio-statement after the $y = 0$ assignment was eliminated and the npr of the statement propagated down towards the exit node. However, Figure 4.16 shows that this approach still has some shortcomings. It can only remove the final prio-statement of a thread before an exit or surface node. The image depicts that, if a thread has multiple subsequent outgoing dependencies, it can have multiple prio-statements in a row. Since the npr is assigned depending on the longest weighted path originating on a node, a prio-statement has to be inserted after each node, if each subsequent node has a dependency causing a longest weighted path that is higher than their successor node. However, the example shows that the thread could keep its original prioID of 4 without causing any priority inversions.

The third solution was to develop a downwards propagation of nprs that propagates nprs

Algorithm 4 Calculating the maximum npr of a list of SCCs, starting at the SCC containing the final exit node

```

1: function LONGESTWEIGHTEDPATHBACKWARDS(currentSCC)
2:   dependencies  $\leftarrow$  currentSCC.incomingDependencies
3:   neighbors  $\leftarrow$  currentSCC.predecessors
4:   prio  $\leftarrow$  MAX_PRIORITY
5:   for each d in dependencies do
6:     if ! (d.scc.visited) then
7:       d.scc.visited  $\leftarrow$  true
8:       LONGESTWEIGHTEDPATH(d.scc)
9:     end if
10:    prio  $\leftarrow$  min(d.scc.min_npr - 1, prio)
11:  end for
12:  for each n in neighbors do
13:    if ! (n.scc.visited) then
14:      n.scc.visited  $\leftarrow$  true
15:      LONGESTWEIGHTEDPATH(n.scc)
16:    end if
17:    prio  $\leftarrow$  min(n.scc.max_npr, prio)
18:  end for
19:  if (currentSCC.isEntryNode) then
20:    currentSCC.max_npr  $\leftarrow$  currentSCC.min_npr
21:  else
22:    currentSCC.max_npr  $\leftarrow$  prio
23:  end if
24: end function

```

from earlier in the model until it is not possible to propagate the priority any further. This approach to optimizations, however, is intrusive and overhauls the npr assignment distinctively, instead of correcting after the original approach. Thus, the whole model is analyzed to not only reduce unnecessary context switches at the end of threads or before pauses, but also to visibly reduce the number of prio-statements required in all models that contain at least one dependency. First, a minimum npr is calculated for all nodes. This calculation is done as described in Section 4.1.2 using the longest weighted path approach. In the following it will be called the minimum npr . The minimum npr of a node will always be just one higher than the highest minimum priority of all other nodes that are dependent on the execution of this node. Thus, the assigned priority of the thread executing this node must never fall below this minimum npr . Afterwards, a maximum npr is calculated for all nodes analogous to the minimum npr calculation. The algorithm performs a depth-first search analogous to the approach in Section 4.1.2, yet originates at the final exit node and at depth nodes in the model and traverses the control-flow edges backwards. Surface nodes are given the highest minimum npr of the model as their maximum npr and the initial entry node is given its

4. Priority-Based Compilation

minimum npr . Then, all nodes are given the smallest maximum npr of their predecessors. If any incoming dependencies exist, the minimum out of the smallest maximum npr of their predecessors and the smallest minimum npr of the origin nodes of the incoming dependencies - 1 is assigned to the node. Due to this it is guaranteed that the maximum npr of a node is always lower than the minimum npr of all nodes it depends on. If the final assignment of the npr stays within the maximum and minimum npr , the schedule of the execution will be valid. If there are any nodes depending on this node, its assigned npr must be higher than the assigned npr of those nodes, as the assignment of their maximum npr is lower than the minimum npr of the original node. Analogously, if this node depends on the execution of other nodes, its assigned npr must be lower than the maximum npr of the other nodes and thus higher than the assigned npr of those. The algorithm is shown in Algorithm 4. Note the similarity to the original npr assignment in Algorithm 1. The `MAX_PRIORITY` field contains the highest minimum npr . If a node does not have any predecessors as is the case in surface nodes, the `prio`-field will never be overwritten. Thus, the node will receive the highest minimum npr as its maximum npr . The `isEntryNode` property of an SCC will return true, if the SCC only contains an entry node.

After the minimum and maximum npr have been computed, the actual npr has to be assigned to the nodes. The idea is to assign them in a way such that the same npr is applied to as many nodes in sequence as possible. This is done by starting at the initial entry node as well as all depth nodes and running a depth-first search until a surface node or the final exit node is reached. The final node of the depth-first search node is then assigned its maximum npr as their npr . Then all predecessors of reached nodes are recursively given the same npr as their successors, as long as this priority is higher or equal to their minimum priority. The exceptions are all join and exit nodes. After the termination of a thread its priority does not matter anymore with the exception of the thread with the lowest npr . If all exit nodes are assigned their maximum npr as well as the join node, the maximum npr of the join will coincide with the lowest maximum npr of one of the exit nodes. Due to the way the maximum npr are assigned, the join must inherit the priority of one of its predecessors. Thus, exit nodes as well as join nodes can freely be assigned their maximum npr .

Furthermore, if a node has multiple successor nodes, such as conditional nodes and fork nodes, they will receive the npr of the successor node with the highest npr to ensure that the priority does not rise during a tick. If a visited node is encountered during the depth-first search, it is not necessary to visit it again, since all its successor nodes have been visited and given their final npr . Its npr is still returned to the node trying to visit this node to calculate its npr . It is further important to note that forks work as intended. Since the minimum and maximum npr of all entry nodes of a thread are the same, they will always be given the minimum npr as their final npr . During the final npr assignment, the minimum npr of the entry node with the highest minimum npr is then propagated to the fork node. Since the assignment of the minimum npr is identical to the approach delineated in Section 4.1.2, the propagated npr cannot be lower than the minimum npr of the fork node. Figure 4.17 shows the result of the optimizations by reference to the same model as in Figure 4.16. Compared to its

4.2. Optimizing the Node Priority Assignment

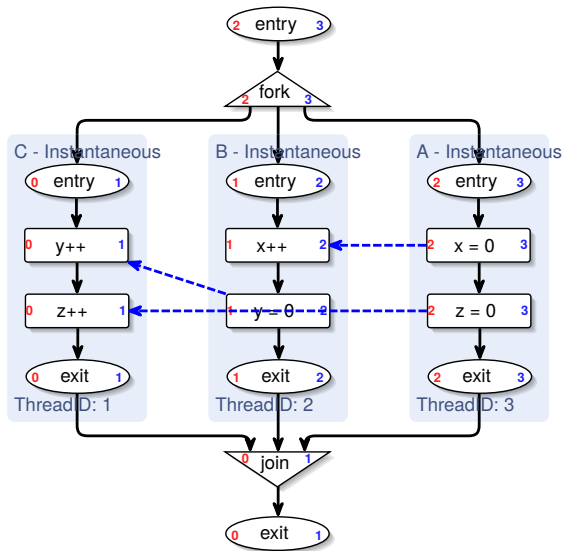


Figure 4.17. Example SCG where all prio-statements could be optimized away

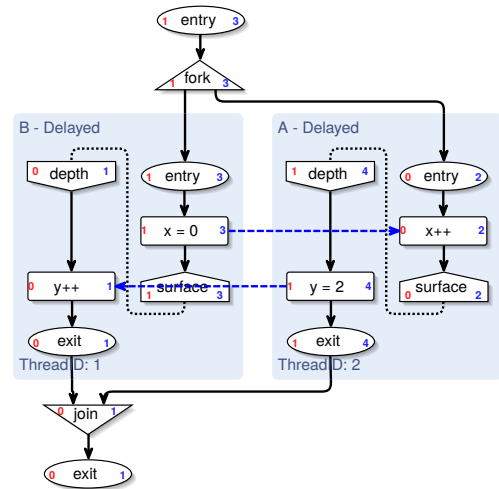


Figure 4.18. Example SCG where the thread with the highest entry npr also has the lowest exit npr

counterpart there are no prio-statements required at all. Another advantage of this approach is that the overall amount of required prioIDs can be reduced. A program that originally had more prioIDs than the length of an integer, but now has less, does not require the array-based approach as introduced in Section 4.1.4. The required memory is thus reduced even more. This optimization furthermore improves the priority assignment and thus both the C code as well as the Java code generation are improved without causing any major changes in the code generation itself.

Figure 4.19 shows the progression of the maximum, minimum, and assigned nprs of the model in Figure 4.17. Across all threads, the optimized npr remains stable, while both the maximum and minimum nprs of most threads vary across the tick. It also illustrates that the optimized npr remains stable between the maximum and minimum npr at all times in this example.

Overall, the complexity of the algorithm does not change. In the original approach, all control-flow edges and all dependencies were passed exactly once. The optimization adds one full pass of all control-flow edges and dependencies for the calculation of the maximum nprs as well as one pass of all control-flow edges for the calculation of the final nprs. Therefore, the complexity of the compilation stays linear in the size of the model. This optimization, however, affected previous assumptions. The forking thread always continued as the child thread with the highest tsID and the joining thread was previously always the thread with the lowest tsID, since all threads were assigned the same npr at their exit nodes. This optimization, however, can cause the exit nprs of all threads to differ. Therefore, the joining thread is now determined by the lowest npr of the exit nodes of all sibling threads. If there are multiple

4. Priority-Based Compilation

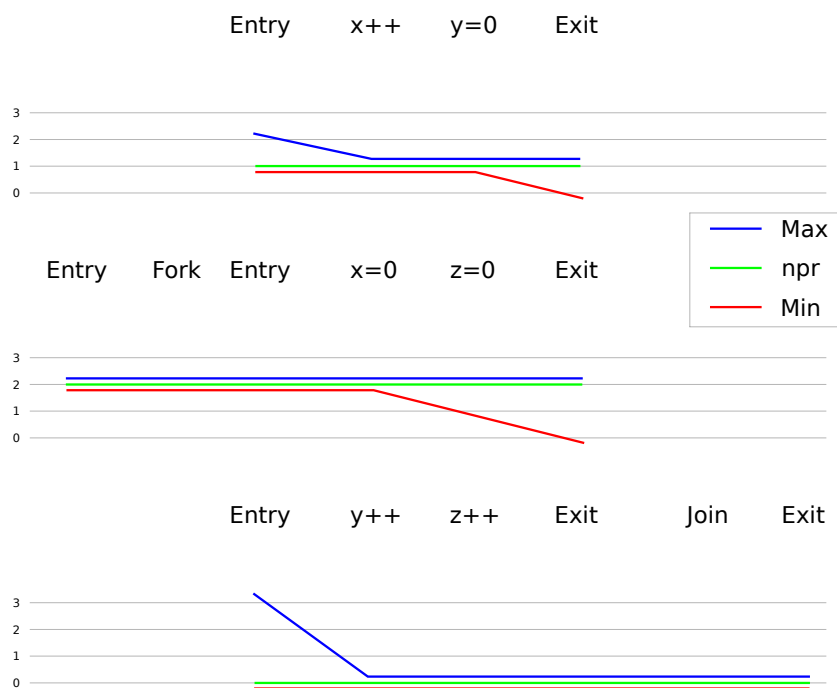


Figure 4.19. Comparison of maximum, minimum, and assigned npr s of threads from Figure 4.17

exit nodes that share the lowest npr , the joining thread will be decided by their $tsID$. The $tsID$ s themselves are also assigned accordingly, as the thread with the lowest exit npr receive the lowest $tsID$ s. If the exit npr of the forking thread is now lower than those of all its siblings, this thread would have to presume the role of the joining thread. This would be impossible though with the original fork- and join-macros. The fork-macro assumes that the forking thread continues as the first thread in the fork-par-join block, while the join-macro assumes that it is the final thread. Figure 4.18 shows a example where such a situation occurs. Due to the dependencies before the pauses of the two threads, thread B has to have a higher npr in its entry node than thread A. Therefore, it is assigned the higher $tsID$. After the pauses the dependency from $y = 2$ towards $y++$ causes thread A to have a higher npr . In the original allocation of npr s, the npr of the exit node would still be 0. In this optimization, however, the npr of the assignment earlier is propagated down towards the exit node. Thus, thread B has the lowest npr at the exit node and will hand down its $tsID$ towards the join. The thread therefore continues after the join. Thus, the same thread that has the highest entry npr also has the lowest exit npr .

One solution to this problem may be to deactivate the propagation of npr s of one randomly selected sibling thread before the final exit node. Then, this thread would have at most the same npr in its exit node as the forking thread. However, this would result in additional context switches and requires some analysis during the npr assignment that is usually done during the allocation of $tsID$ s. To disrupt the readability as little as possible and to be consistent in the macro use, the fork macro was adapted slightly. After forking its siblings, the forking

4.2. Optimizing the Node Priority Assignment

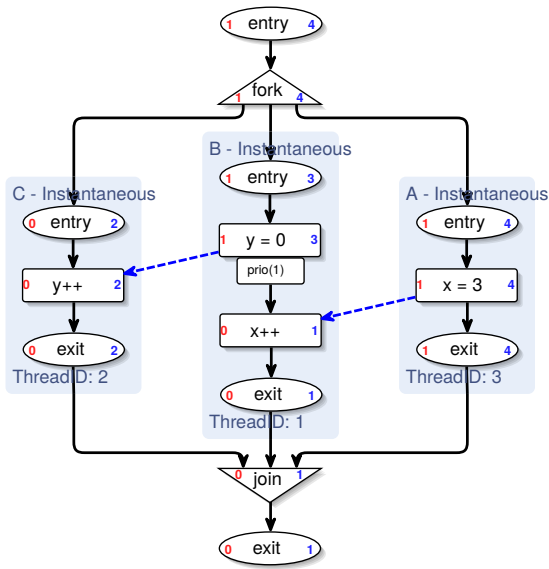


Figure 4.20. Another SCG where this optimization still could not resolve all unnecessary context switches, as the context does not need to switch from B after the `prio(1)` statement towards C.

```

1 // Old definition:
2 #define fork4(label1, p1, label2, p2, label3, p3,
3   label4, p4) \
4   initPC(p1, label1); enable(p1); \
5   initPC(p2, label2); enable(p2); \
6   initPC(p3, label3); enable(p3); \
7   initPC(p4, label4); enable(p4);
8 // New definition:
9 #define fork4(label0, label1, p1, label2, p2, label3,
10  p3, label4, p4) \
11  initPC(_cid, label0); \
12  initPC(p1, label1); enable(p1); \
13  initPC(p2, label2); enable(p2); \
14  initPC(p3, label3); enable(p3); \
15  initPC(p4, label4); enable(p4); \
16  goto label0;

```

Listing 4.3. Old and new implementation of the `fork4` macro

thread is now allowed to jump to a label that was passed as an argument to the `join`-macro. If the `join` macro was to be changed, the `join` macro would sometimes be placed between two parallel threads and signify a jump towards the end of the parallel statements, whereas the changed `fork` macro would not change this order. The changed `join` macro would further mean that the end of a parallel block is marked with a `par`-statement, even though those usually stand between two parallel threads. Listing 4.3 shows the difference between a `fork4`-macro before and after the change. The macro `initPC` takes a priority and a label and assigns the label as the next continuation point of this priority. The `enable` macro naturally enables the priority that was passed to it and sets it to active. Subsequently, the control-flow jumps to the continuation of the forking thread as one of the siblings. The difference between the two macros is twofold: First, the original macro did not need the `label0` argument that is used to determine the continuation of the first thread. In the new macro, it is passed as an argument for another `initPC` macro together with the currently running priority that is saved in the `_cid` field. Second, after all sibling threads have been spawned, the forking thread jumps towards this label, instead of continuing directly after the

4.2.1 Proof of Correctness

It is important to show that these optimizations do not cause any priority inversions of nodes that depend on each other. The maximum `npr` of a node can never be higher than the

4. Priority-Based Compilation

minimum n_{pr} of any node it depends on. This means that if the final assignment of the n_{pr} of a node remains between its maximum and minimum n_{pr} , its n_{pr} must be lower than the assigned n_{pr} of all nodes it depends on and cannot be executed before those.

- ▷ The minimum n_{pr} can only decrease from one node to the next during a tick. Each node is at least assigned the highest n_{pr} of all its succeeding nodes.
- ▷ Analogous to this, the maximum n_{pr} can only decrease during a tick as its assignment is bound to the lowest maximum n_{pr} of its predecessors and the minimum n_{pr} of incoming dependencies.
- ▷ Additionally, the maximum n_{pr} of a node can never be lower than its minimum n_{pr} . Assume that a node n of an SCG has a higher minimum n_{pr} than maximum n_{pr} .
 - ▷ If there are no incoming dependencies on any path following the incoming control-flow edges backwards until they reached a depth node or the initial entry node, then all nodes on the path will be assigned the maximum n_{pr} of their predecessors as their maximum n_{pr} . This is ultimately the minimum n_{pr} of the entry node or the highest possible minimum n_{pr} of the model. However, the minimum n_{pr} of all nodes on the path remains the same across all nodes beginning at the entry or depth node. Thus the maximum n_{pr} of all nodes on the path must be greater or equal to the minimum n_{pr} of all nodes on the path. Since the minimum n_{pr} of n , however, is higher than its maximum n_{pr} , there is a contradiction.
 - ▷ If there is one incoming dependency on any path following the incoming control-flow edges backwards towards a node n_1 , the maximum n_{pr} of this node is the minimum of the minimum n_{pr} of the source of the dependency -1 and the maximum n_{pr} of its predecessors in the control-flow. Since the minimum n_{pr} cannot rise during a tick and there were no further dependencies on the path, the minimum n_{pr} of n must be equal to the minimum n_{pr} of n_1 . As the maximum n_{pr} of n is higher than its minimum n_{pr} and maximum n_{pr} s also cannot rise during a tick, the maximum n_{pr} of n_1 must also be higher than its minimum n_{pr} and thus at least as high as the minimum n_{pr} of the source node of the incoming dependency of n_1 . Since, however, the maximum n_{pr} must be lower than the minimum n_{pr} of any sources of incoming dependencies, there is a contradiction.
- ▷ Thus, since the assigned n_{pr} must remain between the maximum and minimum n_{pr} , and only changes its priority towards the minimum n_{pr} when necessary, it will also only decrease and never increase.

4.2.2 Proof of Improvement

First, it can be proven that as long as there exists at least one concurrent, non-confluent dependency in the SCG, at least one prio-statement can be optimized away. Since there will always be a thread that has an outgoing dependency and no incoming dependency afterwards,


```

1 loop
2   present I then
3     pause
4   end present;
5   emit A;
6   ||
7   pause
8 end loop

```

Listing (4.4) Simple schizophrenic Esterel model

due to the fact that there must never be a cycle induced by dependencies in the SCG, the npr of the nodes *before* the outgoing dependency can securely be propagated down towards the nodes after the outgoing dependency and thus towards the exit.

In the original approach, prio-statements were introduced every time a node had an outgoing dependency and the longest weighted path through the target of the dependency was longer than any longest weighted paths through successor nodes. After the optimizations, prio-statements can only happen when the propagated npr of a succeeding node undercuts the minimum npr of a node. Since this can only happen if the minimum npr is lowered from one node to its succeeding node and the minimum npr of any node is the same as the npr of a node in the original approach, no new prio-statements can be introduced.

4.2.3 Further Improvements

While this solution can reduce the amount of prio-statements as shown earlier, there are models where an improvement can be made. Figure 4.20 shows an SCG where thread B lowers its priority from the $y = 0$ assignment to the $x++$ assignment. Since $x++$ has to be scheduled after the initialization of x in thread A, its maximum npr has to be one lower than the minimum npr of the node in thread A. If all nodes in thread A were assigned the npr 2 instead of 1, thread A would not require a prio-statement. It would further be possible to assign the npr 1 to all nodes in thread B, since then the maximum npr of these nodes would be 1, allowing them to have a higher final npr . Therefore, no prio-statements would be required overall. This problem can, however, only be solved by analyzing the whole model to find out, whether raising the npr of a whole thread helps to reduce the amount of prio-statements and the amount of required prioIDs. This approach may raise the total number of $nprs$, as for example the npr 2 was previously unused, whereas after the optimization, thread A uses it. However, since the code implementations only require the prioID, the npr -space is not significant. Whether this improvement may sometimes increase the total number of prioIDs, is yet to be debated.

4. Priority-Based Compilation

4.3 Schizophrenia

Schizophrenia is a concept pertaining to synchronous languages. It occurs when a statement is executed several times within the same tick. This happens when a loop terminates and, when re-entered, executes parts that were already executed before the first termination of the loop again in the same macrostep. Listing 4.4 shows a simple example of a schizophrenic Esterel program. If the input *I* is present in the first tick, the control-flow stops the upper thread in line 3, while the lower thread stops at line 7. As the second tick starts, emit *A* is executed. Subsequently, the loop body terminates, since both parallel statements terminated, and is re-entered. If *I* is then absent, emit *A* is executed for a second time this tick. For many statements, a re-execution of the statement alone is not problematic. However, this may for example lead to problems when scopes of local variables are exited and re-entered in the same tick. The compiler will have to recognize that the two instances of the local variables in their respective scopes are different and operations on the second instance do not take past operations on the first instance into account. If the model is translated towards software, this can for example be solved using a stack where the local variable is deleted from the stack after exiting the scope. However, if the Esterel program is to be translated to hardware, problems arise. The circuit semantics do not consider signal scoping rules, so both incarnations of a local signal occupy the same physical conduit wires, which may result in unstable circuits. Conventional strategies to solve schizophrenia involve code duplication such as loop unrolling or copying the surface of the loop and introducing a *gotopause* [TS04] as explained in Section 3.2.

In sequentially constructive programs, however, multiple executions of the same statements are no problem, since they can be regarded as sequential statements. Thus, if a local variable is changed, it can simply be regarded as a sequential access to this variable, which is allowed. Section 3.1 talks about the current limitations of schizophrenic models in the data-flow approach.

Conceptually, these schizophrenic models do not cause any problems for the priority based compilation, as can be seen in Figure 4.22. If *I* is absent in the first tick, but present in the second tick of the execution, the conditional check for *I* as well as the *O++* statement happens twice in the second tick, once after leaving the pause and once again after the threads were forked. Listing 4.5 shows the translated code for this model. Again, if *I* is absent in the first tick, thread *A* runs into the pause statement in line 12, while thread *B* pauses in line 6. If *I* is then present during the second tick, the conditional in line 9 holds, *O++* is executed and the thread terminates subsequently. Since thread *B* also terminated, the *join* happens and, due to the *goto* statement in line 17, the threads are forked again. Thread *B* pauses as before and thread *A* checks for *I* and executes *O++* a second time. The behaviors of this model and its resulting code are therefore identical in the priority based compilation.

Problems, however, occur when dependencies exist between the threads that are reincarnated. Figure 4.23 shows such a model. There exists a cycle as highlighted in purple that crosses through the dependency edge into thread *B*. In the original priority based approach, the algorithm would detect the cycle as an SCC and, since dependencies exist within this

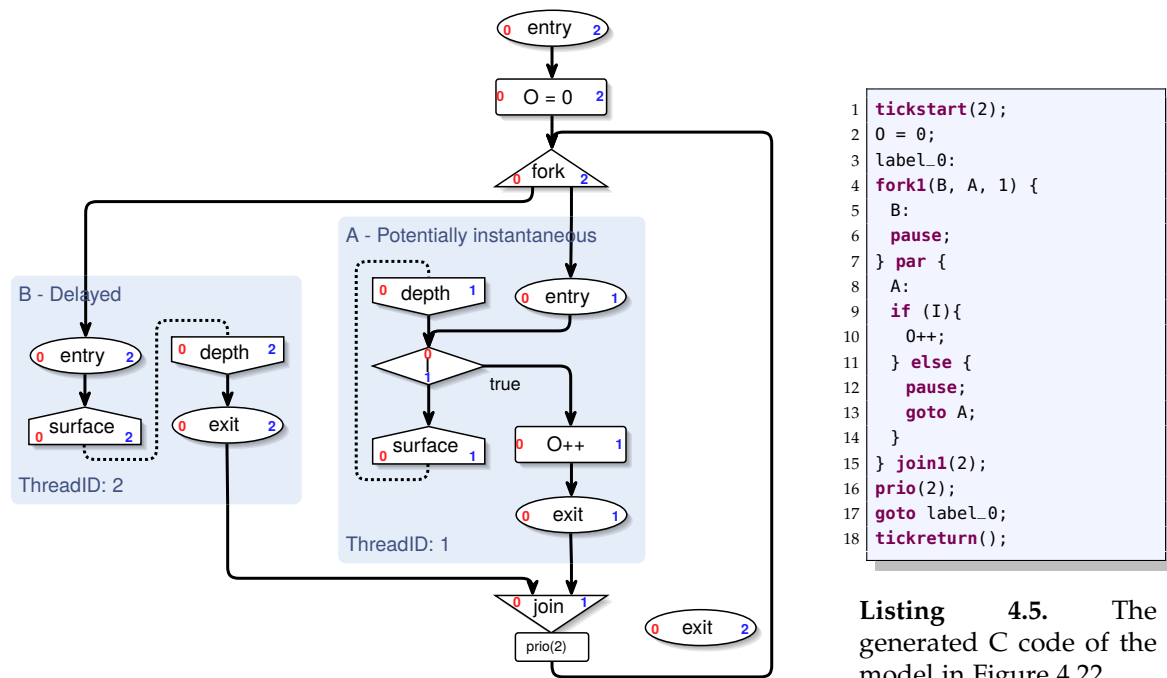


Figure 4.22. The SCG of a simple schizophrenic model with its associated priorities

cycle, reject the model as unschedulable. However, this cycle only exists statically within the model. In any run of the model, the iur property will not be violated since the instance of the conditional $O > 1$ is not runtime concurrent to the second instance of $O++$ in the tick. Their threads have not been instantiated by the same instance of the associated Least Common Ancestor fork. Due to the sequentiality, $O > 1$ is allowed to be executed before the second execution of $O++$. Thus, a way was developed to accept such models and enable them to be schedulable.

4.3.1 Schizophrenia in the Priority Based Compilation

First, it is important to recognize that such a situation only occurs when there exists an SCC that contains a so called *depth-join*. A depth-join is a join where at least one of the joining threads is delayed, while its sibling threads are instantaneous. This would guarantee that it will always take more than one tick for all threads to terminate and the program to resume after the join. All other situations concerning schizophrenia are solved due to the properties of the priority-based approach itself. Once an SCC containing a depth-join has been found, it first has to be searched for dependencies towards a thread that is not one of the threads joined by the join. If that is the case, we cannot guarantee the program to satisfy the iur protocol. Figure 4.24 shows that, while in the first incarnation of the threads A2 and A3 the iur protocol would be upheld, after the execution of the join and the reincarnation

4. Priority-Based Compilation

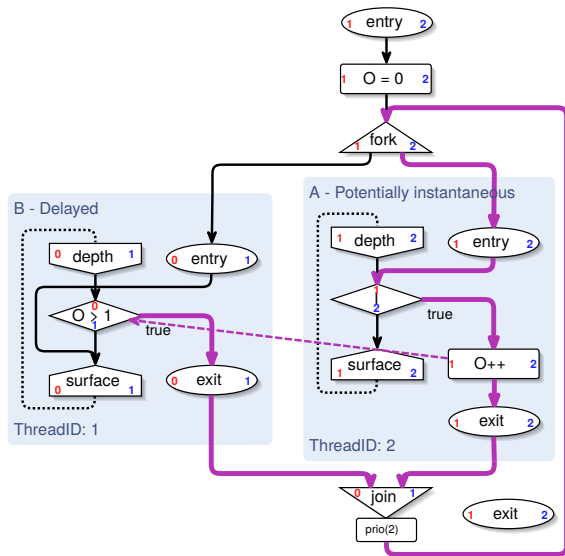


Figure 4.23. The SCG of a schizophrenic model with a dependency induced cycle; The SCC is highlighted in purple.

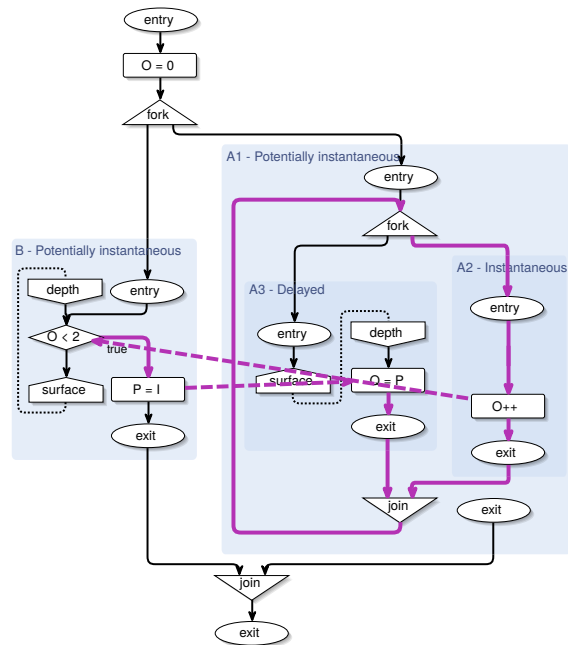


Figure 4.24. The SCG of a schizophrenic model with a dependency induced cycle containing nodes of an external thread; The SCC is highlighted in purple.

of the threads, statement $O++$ in the reincarnation of thread A2 would have to be executed *after* the execution of conditional $O > 1$ in thread B, since statement $P = I$ in thread B had to be performed before statement $O = P$ in the first incarnation of thread A3. According to the definition in Section 2.1 the second instance of $O++$ and the instance of $O > 1$ are runtime concurrent. They appear in the micro ticks of the same macro tick, they belong to the statically concurrent threads A and B and these two threads have been instantiated by the same instance of the associated LCA. Therefore, they violate the iur protocol, as $O > 1$ must be executed before $O++$. The model should be refused by the compiler. Recognizing such a situation can be achieved by analyzing the thread association of nodes in such an SCC. If the SCC traverses into a thread that is statically concurrent to the thread delineated by the depth-join node and its corresponding fork, the program should be rejected.

When such an SCC is found, it is broken up by removing the depth-join. Another search for SCCs is then executed upon the nodes of the former strongly connected component. The original SCC is removed from the list of SCCs and all newly found SCCs are added to this list. If any newly created SCC again contains a depth-join node, they will again be removed recursively, until no more joins are found. The algorithm used for this is highlighted in the next chapter. After all SCCs with depth-joins are broken up, the schedulability analysis is executed as originally intended. If one of the newly found SCCs contains a dependency, the model is

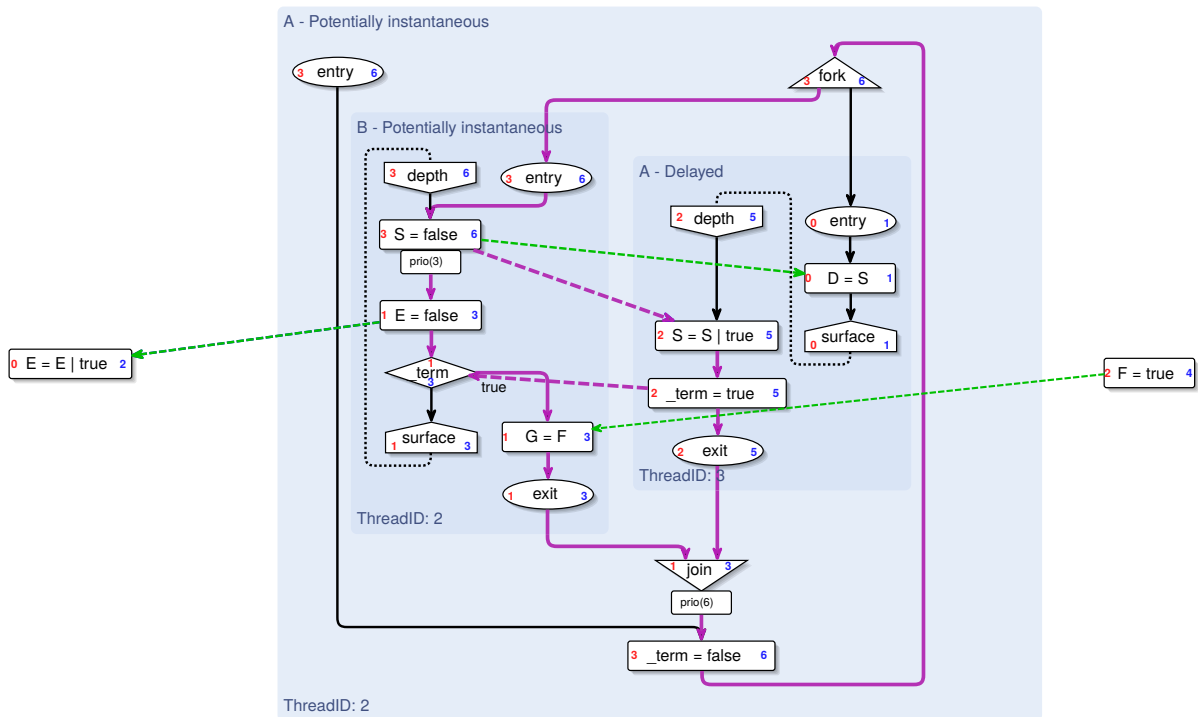


Figure 4.25. The SCG of a schizophrenic model with a dependency induced cycle with further dependencies towards concurrent nodes; The SCC is highlighted in purple.

rejected. This guarantees that the model is still rejected, if there are any dependency cycles within the threads that are joined. Any cycle that has been removed from the schedulability analysis had to cross the join and only existed due to the join node. Thus, any occurring conflicts in these SCCs cannot be runtime concurrent. If they were runtime concurrent, there would still exist an SCC containing these conflicts that does not contain the join.

Afterwards, the node priority assignment has to be revised. Due to the breakup of strongly connected components, the list of SCCs may now be cyclic. An SCC that was broken up now consists of multiple new SCCs that form a cycle, as each node is per definition reachable from each other node in the original SCC via immediate edges. For example, the broken SCC in Figure 4.24 as highlighted in purple now consists of as many SCCs as there are nodes, as each newly calculated SCC now consists of only one node. These SCCs then form a cycle. Thus, the depth-first traversal of the original npr assignment from Section 4.1.2 must now be able to handle a cyclic graph. Further, due to dependencies within these SCCs it may happen that they are assigned different nprs. Therefore, we must allow consecutive nodes to have rising node priorities. The original approach prohibited this, since if the npr between sequential nodes rises during the execution of a thread priority inversion may happen. Thus, all concurrent nodes that are dependent on the execution of any node within the original cycle must be executed after all instances of the sources of their incoming dependencies have been executed. They must be performed after all nodes of the cycle. Therefore, the information about the broken

4. Priority-Based Compilation

cycle is retained and, during the assignment of the minimum node priority as explained in Section 4.2, nodes that are within such a broken cycle also consider all dependencies of nodes within the cycle as long as these dependencies target nodes outside of the cycle. The whole cycle will now be executed before any of the outgoing dependencies of any node in the cycle. On the other hand, if a node has a dependency towards the broken cycle, nothing has to be done. It will always be executed before the first execution of the target of the dependency. If there are any other nodes whose npr is between the priorities before and after the raise of priorities but who do not have any dependencies towards or from the nodes in the cycle, they are confluent towards all nodes within the cycle and the order of their execution does not matter. Figure 4.25 depicts a simplified model of such a situation. As highlighted by the purple edges, a cycle exists containing a dependency. There are further two dependencies from nodes inside the cycle towards nodes in parallel threads, which have been reduced to only two nodes to keep the model simple. It is easy to see that the outgoing dependency from $E = \text{false}$ in thread B has been regarded, since all nodes in the cycle have a higher npr than the $E = \text{true}$ node. This node would therefore always be scheduled after the execution of the $E = \text{false}$ node. The incoming dependency towards the node $G = F$ is also factored in. It is sufficient for the $F = \text{true}$ node to have a higher npr than the $G = F$ node while the other nodes in the cycle are not important, as the node will then always be scheduled before the first execution of $G = F$.

Implementation

The following chapter describes the implementation of the concepts introduced in the previous chapter. The implementation is a part of the KIELER project as introduced in Section 2.2.2. They were written in Xtend, as presented in Section 2.2.1.

First, Section 5.1 describes how the integration into the KIELER project is performed. The section is divided into two parts. In Section 5.1.1 the implementation of the original approach is shown and Section 5.1.2 covers the adaptations required for the optimizations.

5.1 Implementation into KIELER

The KIELER Compiler (KiCo) introduced in Section 2.2.2 provides an interface to easily access transformation chains that translate models or programs from one language to another. One transformation can simply be replaced by another to create a completely new compilation. Due to this adaptability, the KIELER project already contains transformations from and to various languages and intermediate languages as highlighted in Figure 2.7 on page 19. Thus, it is simple to integrate the priority-based compilation into the KiCo toolchain. Figure 5.1 shows the overview of the toolchain with the new translation steps highlighted in red. As displayed in the figure, the SCG annotated with priorities can currently translate to C and Java code. Both results of these transformations, however, require special macros or class extensions to run and cannot run completely standalone.

This abstract view on the transformations hides complexity. Figure 5.2 shows the difference between the modular data-flow compilation steps from SCG to C and Java in Figure 5.2a and the priority-based compilation steps in Figure 5.2b. The approach introduced in this thesis reduces the amount of steps required for the compilation. However, the Priority Calculation step could be divided into the different computation steps introduced in Section 4.1. This was deemed not necessary as the calculation steps are all minimal and a simple property in the KLightD view can individually show and hide the various results of the calculations.

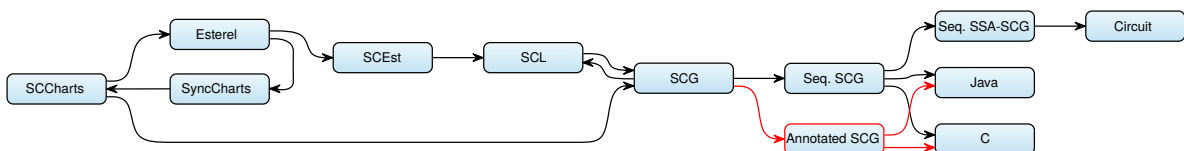


Figure 5.1. Overview of the KIELER compilations enhanced with the priority-based compilation highlighted in red

5. Implementation

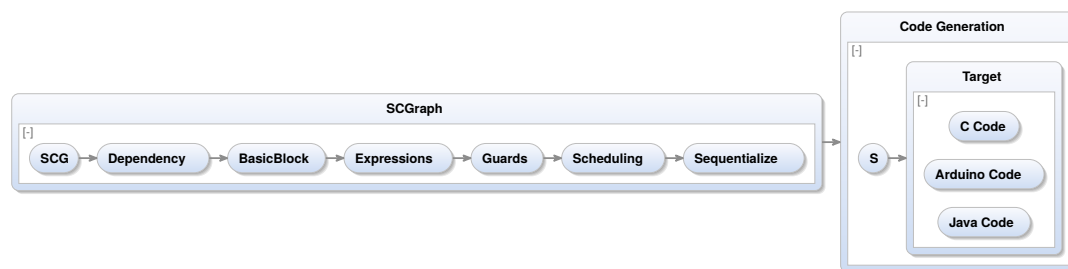


Figure 5.2 (a) Compilation chain of the data-flow compilation

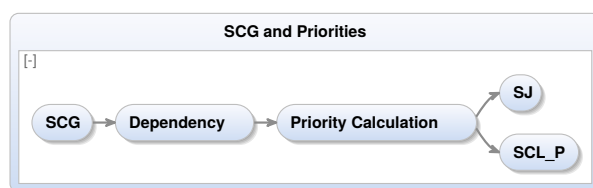


Figure 5.2 (b) Compilation chain of the priority-based compilation

Figure 5.2. Overview of the two low-level compilation approaches

Otherwise, the various steps required for the sequentialization of the model are not required anymore and were therefore discarded. As described in Chapter 4, the transformations to C and Java code are performed on the annotated SCG and no further intermediate representation or transformation is necessary.

Both compilation chains are still adaptable. This way, additional translations into different programming languages are possible. As long as the target language has simple control statements such as gotos or switch-case logic, as well as other statements of imperative programming languages, the translation is possible. Merely the adaptation of the low-level instructions of the priority-based compilation, as implemented via SCL_P or SJ, require more effort.

5.1.1 Original Approach

The original approach to the priority-based compilation implements the concepts introduced in Section 4.1. Therefore, it starts with the calculation of the SCCs. As mentioned in Section 4.1.1, this is done via a modified version of Tarjan's algorithm. Since an SCG is a directed graph with various different kinds of edges, it must be defined over which edges the algorithm may pass first. Thus, shortcuts were created to access only the required edges including immediate control-flow edges and concurrent, but not confluent dependency edges. Listing 5.1 shows the used algorithm. When visiting a node, it first receives an index and a lowlink indicating the node with the lowest index reachable from this node depending on the current count of nodes and is pushed onto the stack, which will later be used to determine the SCC. The data structures used to store the lowlink and index of a node are Java hashmaps. After assigning


```

1 private def void tarjan(Node currentNode) {
2
3     index.put(currentNode, count)
4     lowlink.put(currentNode, count)
5     count++
6     stack.push(currentNode)
7     visited.put(currentNode, true)
8
9     for (nextNode : currentNode.neighborsAndDependencies) {
10        if (isContained.containsKey(nextNode) && isContained.get(nextNode)) {
11            // Next node has not yet been visited
12            if (!visited.containsKey(nextNode) || !visited.get(nextNode)) {
13                tarjan(nextNode)
14                lowlink.replace(currentNode, Math.min(lowlink.get(currentNode), lowlink.get(nextNode)))
15            }
16            // Next node has already been visited, hence in the current Strongly Connected Component
17            else if (index.get(nextNode) < index.get(currentNode)) {
18                if (stack.contains(nextNode)) {
19                    lowlink.replace(currentNode, Math.min(lowlink.get(currentNode), lowlink.get(nextNode)))
20                }
21            }
22        }
23    }
24
25    // Create the Strongly Connected Component
26    if (index.get(currentNode) == lowlink.get(currentNode)) {
27        var scc = <Node>newLinkedList
28        var w = stack.peek;
29        do {
30            w = stack.pop
31            scc.add(w)
32        } while (w != currentNode)
33        sccList.add(scc)
34    }
35 }

```

Listing (5.1) Xtend implementation of Tarjan's algorithm

index and lowlink, all neighbors as calculated with the shortcuts described above will be checked. If they were not visited yet, the algorithm is called recursively on that node. After a node has finished all calculations, its lowlink is checked. If the lowlink of the visited node is lower than the lowlink of the current node, there exists a path to a node whose index is lower than the index (or lowlink) of the present node. Thus, we store the new value. Similarly, if the node was already visited, the lowlink of the current node is updated, if necessary.

After all neighbors of a node are visited, it is checked whether its lowlink is identical to its index. If this is the case, there is no path to a node with a lower index than this node. Therefore, this node is the beginning of a possible cycle with a path ending in this node. All nodes in this cycle will still be on this stack, as their lowlink is dependent on the path to the node with the lowest index. As this is the aforementioned node, their index must not equal their lowlink and they cannot be removed from the stack. Assume that they have a lowlink that is a lower index than the first node. Then the lowlink of the first node must also equal this lowlink, as it is lower than its own lowlink and there exists a path between the two nodes.

5. Implementation

Therefore, a path exists to the node with the index of the lowlink. If a node has the same index and lowlink, its SCC consists of all nodes above it on the stack until another node has the same index and lowlink. Thus, the stack is dismantled in lines 28 to 32 and the SCC is built. This algorithm is repeated on unvisited nodes until all are visited once.

The resulting lists of nodes are memorized as the SCCs and passed on. The list of SCCs is also handed to the SCCharts KLighD synthesis for visualization purposes. If two nodes are inside the same SCC and they have a direct control-flow or dependency edge between them, this edge is thickened and colored purple.

After all SCCs were found, a scheduling analysis is performed. This analysis merely goes through all nodes in all SCCs and finds all their dependencies. If the dependency is concurrent and not confluent and is either a *write-write* dependency or a dependency towards a node inside the same SCC, then the model is rejected.

Subsequently, the Node Priority (*npr*) assignment is performed as depicted in Listing 5.2. The assignment traverses through all SCCs in the model exactly once. If an SCC is visited, the neighbors and dependencies, which do not include any nodes inside this SCC, are calculated in lines 3 and 4. Then, the dependency targets are visited in lines 8 to 19. Since the lists of dependencies and neighbors include nodes and not SCCs, line 9 looks up which SCC the target node of the dependency belongs to. If this SCC has not yet been visited, it is visited and its calculated *npr* used to determine the *npr* of the current SCC in line 13. If the SCC has already been visited, its *npr* is already known and used directly to determine the *npr* of the current SCC in line 16. Since a node has to be executed before the target of its dependency, the calculation of the *npr* takes the *npr* of the target SCC and increments it. The same procedure is repeated for all neighbors of the SCC in lines 22 to 33 without the need to increment the *npr* of targets, as the sequential order already defines the order of execution. Afterwards, the *npr* of all nodes in this SCC is determined and saved into a hashmap.

Afterwards, the Thread Segment IDs (*tsIDs*), Priority IDs (*prioIDs*) and optimized Priority IDs (*prioIDs*) are calculated as described in Section 4.1.3. Since the calculation of the *tsIDs* is similar to the depth-first search already described by the *npr* assignment and the calculations of *prioIDs* and optimized *prioIDs* are simple calculations, they are not described in detail here.

At the end of all these calculations, all important information is stored for later use and for display purposes. All nodes store their *npr*, *tsID* as well as *prioID* as annotations, as these will be shown by the visualization. In the KLighD synthesis, when a node is visited, its *npr* and *prioID* are read and, if desired, visualized as a red or blue number in one of the borders of a node. Each time a region is visualized, only the *tsID* of the very first entry node is visualized. However, the *tsID* may change during the execution of a thread. Thus, this information is unreliable. One could additionally write the *tsID* next to the nodes in each thread, but this could unnecessarily clutter the layout.

In the algorithm, each node is visited exactly once, as it is checked whether a node has previously been visited before the method is called with this node. In each node, all outgoing control-flow edges as well as dependency edges are checked once. Thus, the complexity scales in the number of nodes and immediate edges in the SCG. However, this is only true under the

```

1 private def void longestPath(LinkedList<Node> currentSCC) {
2
3     val neighbors = currentSCC.findNeighborsOfSCC
4     val dependencies = currentSCC.findAllDependenciesOfSCC
5     var int prio = 0
6
7     // Visit all dependencies and calculate their node priority
8     for(dep : dependencies) {
9         val depSCC = sccs.get(sccMap.get(dep))
10        if(!visited.get(depSCC)) {
11            visited.put(depSCC, true)
12            longestPath(depSCC)
13            prio = Math.max(prio, npr.get(depSCC) + 1)
14        } else {
15            if(min.containsKey(depSCC)) {
16                prio = Math.max(prio, npr.get(depSCC) + 1)
17            }
18        }
19    }
20
21    // Visit all neighbors and calculate their node priority
22    for(n : neighbors) {
23        val nSCC = sccs.get(sccMap.get(n))
24        if(!visited.get(nSCC)) {
25            visited.put(nSCC, true)
26            longestPath(nSCC)
27            prio = Math.max(prio, npr.get(nSCC))
28        } else {
29            if(min.containsKey(nSCC)) {
30                prio = Math.max(prio, npr.get(nSCC))
31            }
32        }
33    }
34    npr.put(currentSCC, prio)
35 }

```

Listing (5.2) Xtend implementation of original npr calculation

assumption, that the accesses to hashmaps happen in constant time. If, due to collisions, the complexity of hashmap accesses rise, the complexity of the whole computation rises. Thus, an alternative computation utilizing annotations instead of hashmaps can rectify this.

5.1.2 Optimization

Due to the simplicity of the optimizations introduced in Section 4.2, the adaptations to the code are kept to a minimum. In the npr assignment step, the compilation from above is performed twice: Once from the top and once from the bottom, where the traversal goes backwards through the graph. In order to do this, additional methods were created to find the preceding nodes for incoming control-flow and dependency edges. The forward analysis calculates the minimum npr of SCCs while the backward analysis calculates their maximum npr. When the backwards traversal encounters an entry node, however, it continues like the forward analysis but appoints the minimum npr of the entry node as its maximum npr. After both the minimum and maximum nprs are calculated, the actual npr is calculated as shown

5. Implementation

```
1 private def int calculateNodePrios(LinkedList<Node> currentSCC) {
2   val neighbors = currentSCC.findNeighborsOfSCC
3   var nextPrio = max.get(currentSCC)
4   var succPrio = Integer.MIN_VALUE
5
6   for(n : neighbors) {
7     if(!(n instanceof Join)) {
8       val nextSCC = sccs.get(sccMap.get(n))
9       if(!visited.containsKey(nextSCC) || !visited.get(nextSCC)) {
10        visited.put(nextSCC, true)
11        succPrio = Math.max(calculateNodePrios(nextSCC), succPrio)
12      } else {
13        if(nodePrio.containsKey(n)) {
14          succPrio = Math.max(nodePrio.get(n), succPrio)
15        }
16      }
17    }
18  }
19  if(succPrio >= min.get(currentSCC)) {
20    nextPrio = succPrio
21  }
22  for(node : currentSCC) {
23    if(node instanceof Join) {
24      nodePrio.put(node, max.get(currentSCC))
25    } else {
26      nodePrio.put(node, nextPrio)
27    }
28  }
29  return nextPrio
30 }
```

Listing (5.3) Xtend implementation of the optimized npr assignment

in Listing 5.3. As described in Section 4.2, a depth-first search is performed analogous to the calculations of maximum and minimum npr. Each SCC first finds the highest npr of its successors in lines 6 to 18 and saves it in the succPrio field. If the succeeding node is a join node, this successor will not be regarded, as the end of a thread signifies a break in the priority propagation as explained Section 4.2. If the node has no successors or if the succPrio is lower than the minimum npr of the node, the nextPrio field saving the next npr of this node remains as the maximum npr of the node. Otherwise, the value of succPrio is written into nextPrio. In lines 22 to 28, the saved npr is then written in a hashmap for all nodes in the current SCC. As explained in Section 4.2, joins must always receive the maximum npr as their npr, thus the special case is regarded in line 23.

The algorithm, similar to the npr assignment in Section 5.1.1 visits each node exactly once. Instead of traversing over all immediate edges, it only traverses all control-flow edges. This, the complexity scales in the number of nodes and control-flow edges. Similar to the npr assignment, the optimization relies heavily on hashmaps, which may increase the complexity in the worst case.

```

1 for(scc : sccList) {
2   if (scc.head.hasAnnotation(SCHIZO_ANNOTATION) && scc.length > 1) {
3     if(findSchizoDependencyCycle(scc)) {
4       newSCCs.add(scc)
5     } else {
6       scc.head.removeAllAnnotations(SCHIZO_ANNOTATION)
7       var subSCCCalc = injector.getInstance(StronglyConnectedComponentCalc)
8       val newSCC = subSCCCalc.findSCCs(scc.filter[n | !n.hasAnnotation(DEPTH_JOIN_ANNOTATION)])
9       var LinkedList<Node> join = newLinkedList
10      join.addAll(scc.filter[n | n.hasAnnotation(DEPTH_JOIN_ANNOTATION)])
11      newSCCs.addAll(newSCC)
12      newSCCs.add(join)
13      schizoSccList.add(scc)
14    }
15  } else {
16    newSCCs.add(scc)
17  }
18 }
19 sccList = newSCCs

```

Listing (5.4) Xtend implementation for breaking up schizophrenic SCCs

5.1.3 Schizophrenia

To allow schizophrenic models for the priority-based compilation, multiple adaptations were needed. First, the schedulability analysis must allow such models. Therefore, while scanning for SCCs with Tarjan's algorithm as described earlier, joins were annotated, if they are classified as *depth-joins*. The SCC containing this join is also marked by annotating the head of the list of nodes in the SCC. After all SCCs were created, the method depicted in Listing 5.4 breaks up all annotated SCCs. It iterates over all potentially schizophrenic SCCs. In line 3 it computes whether a schizophrenic SCC has a dependency cycle crossing into an external thread. If such a dependency cycle exists, the model should be rejected as described in Section 4.3.1. Thus, the SCC as a whole is added back into the list of SCCs. As it contains a dependency, the model will be rejected by the schedulability analysis. If the model does not contain any dependency cycles crossing into an external thread, lines 6 to 13 describe how the SCC is broken up. As there may still remain cycles or dependency cycles within this SCC, a new analysis for SCCs has to be started. For this, an injector is used to start a new run of the SCC calculation on a list of nodes containing only the nodes of the SCC without the depth-join. As this new list of nodes is presumably not schizophrenic, the annotation of the initial node has to be removed. During the recursion, additional schizophrenic SCCs might be found. The same procedure will then be called recursively on those as well. After the analysis is performed, the method returns a new list of SCCs. The previously removed depth-join must be added as an SCC back to this list. All these newly created SCCs together with all non-schizophrenic SCCs are then aggregated into a new list of SCCs. It is further important to retain information of the initial schizophrenic SCC due to the special cases described in Section 4.3.1. Therefore, the `schizoSccList` memorizes all schizophrenic but broken up SCCs for later use. The schedulability analysis is then performed as described in Section 5.1.1 without any changes on the new list of SCCs.

5. Implementation

Since schizophrenic regions can be nested indefinitely, the recursive calls can increase the time complexity of the compilation. If there are n nested schizophrenic regions in the model, the recursive call to break up the SCCs is executed n times. As a search for SCCs is executed in each recursive call and the SCC analysis scales linearly in the size of the graph, the complexity of the solution to schizophrenia in the worst case is quadratic. According to Tardieu and de Simone, this is standard for a compiler that can handle schizophrenia [TS04]. Since schizophrenic models occur rarely according to Berry [Ber00a]. Thus, nested schizophrenic models presumably occur even more rarely. On average, the compiler will thus still run in linear time.

During the `npr` calculation, the `schizoSCCList` is then used to determine the minimum `npr`. As mentioned in the special cases in Section 4.3.1, all nodes of the broken up cycle must have a higher `npr` as any node outside the cycle dependent on any node inside the cycle. Thus, an SCC that is a part of a broken up SCC also checks for all dependencies of all other SCCs in the broken up SCC. To reduce redundancy, the first time all dependencies of a broken up SCC are calculated, the dependencies are saved in a hashmap. If the dependencies of this SCC are then required again, the hashmap simply returns them. Otherwise, nothing is changed in the `npr` assignment, because the final `npr` assignment described above checked for the visited state of the next SCC. It was therefore already able to handle cycles in the list of SCCs.

Evaluation

This chapter evaluates the implementation of the priority-based compilation as presented in Chapter 4 and Chapter 5. First, Section 6.1 argues the correctness of the computation via benchmarks used for testing the correctness of the data-flow compilation approach and additional benchmarks used to test the newly compilable models of the priority-based approach. Afterwards, Section 6.2 compares different aspects of the two approaches as the size of the resulting code, the compilation time as well as execution times. It also covers the classes of models that are newly compilable. Section 6.3 highlights the improvement made by optimizing the `nprs` as shown in Section 4.2. Lastly, Section 6.4 shows the differences between different implementations of the priority-based approach. It provides arguments for and against a complete array-based implementation of priorities and delineates the time saved by the optimizations done in this thesis.

6.1 Correctness

The concepts presented in Chapter 4 were implemented in Chapter 5 in the KIELER project. The following section shows an experimental evaluation of this implementation.

6.1.1 Dataset

The SCCharts test data repository at the point of this thesis contained 75 models each with `eso` files containing traces. They are used to test the correctness of different aspects of SCCharts from general functionality over various transformations to simple example models as well as for automatic regression testing on the master. In order to do this, the models can be executed and special simulation software reads the inputs as specified in the traces and compares the resulting outputs. In addition to the test data repository, two models testing explicitly for cyclical models and five schizophrenic models were created for this test. Of those five schizophrenic models, three are supposed to work with the new approach and two are supposed to fail.

All of these models were compiled to C code using the translation described in Section 4.1.4 and subsequently executed and ran against the prepared `eso`-traces. This was performed within a JUnit Test.

6. Evaluation

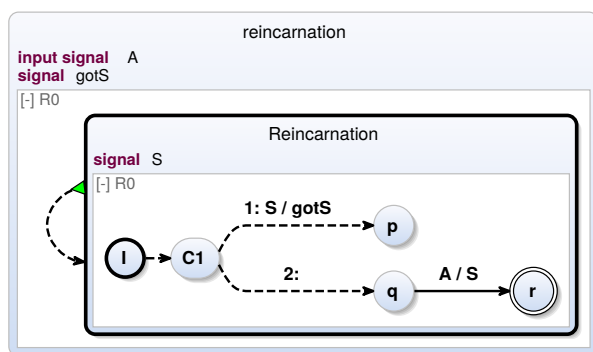


Figure 6.1. SCCharts model that failed during the JUnit test

6.1.2 Results

Of the 75 models from the SCCharts test data repository, 74 worked successfully while the JUnit test of one model failed. This model as shown in Figure 6.1, however, while supposed to fail for the current implementation of the data-flow compilation approach, is allowed in the priority-based compilation. Due to the immediate reincarnation of the signal S when the control-flow reached the final node in the inner Reincarnation state, the data-flow compilation rejects this model. Similar to the examples from Section 4.3, there exists a possible immediate cycle that can be broken up by the join, while one of the threads is always delayed. Due to the adaptations made for schizophrenic models, this model is allowed as any dependencies are superseded by the sequential execution. Therefore, the model should be accepted. As the test checked for a failure during the compilation, the test itself failed, because no failure occurred. This is the expected and correct behavior.

Thus, all tests executed successfully and the correctness of the priority-based compilation approach has been proven for at least the base test cases and the provided traces.

6.2 Comparison to the Data-flow Low-Level Compilation

Next to a compilation approach for SIASC and some IASC programs, the goal of this thesis was also to create an alternative to the already implemented data-flow compilation approach. Thus it is not only important to measure whether the execution of generated models is possible in reasonable time, but also to compare it with the alternate code. The comparison is performed on the generated C code, since C is better suited for timing comparisons than Java due to the worse predictability in Java caused by e. g., garbage collection. An additional reason for performing the benchmarks in C is that the comparison by von Hanxleden et al. [HDM+14] was also based on the compilation to SCL_p. This way, these results can be supported directly.

The following comparison will therefore be between the generated C code of the priority-based low-level compilation and the data-flow low-level compilation. They were performed on a virtual machine running on the the *aeon* server of the Real-Time and Embedded Systems

6.2. Comparison to the Data-flow Low-Level Compilation

Group of the Kiel University, Germany. The machine has two Intel(R) Xeon(R) E5540 processors running at 2.53 GHz with 16 MB cache. The virtual machine is running a 64 bit Ubuntu Linux v. 16.04 with 24 GB RAM.

The benchmarks were performed on Eclipse running the KIELER platform. The timing of transformations is measured automatically. Thus, the compilation time was calculated by simply adding the time needed for each low-level transformation. Certain static properties of a model were calculated from the interim results of the transformations. These properties included the number of states in the normalized SCCharts, the number of nodes in the SCG, the total number of threads as well as the maximum number of parallel threads in the SCG and the number of dependencies. Many of these properties were not used, yet they could be used for further evaluation. One additional property for both compilation approaches was the actual size of the program after the compilation. Since the priority-based compilation approach uses various header files, the size of the program is deceptively small. Initial testing showed that the lines of code produced by the transformations was more than 50% smaller for the priority-based approach than the control-flow approach. However, when expanded with the macros, the programs of the former increased significantly in size. Thus, to test the actual size of the programs, the benchmarks compare the size in bytes of the executable file after the compilation performed using the gcc¹ compiler without only the standard options enabled. To calculate the execution times of a compiled model, the simulation component for generated C code in the KIELER project was adapted. For each model an additional output variable was latched on to the surrounding simulation code that was supposed to save the required time of the execution. In the code that executes the tick function of the generated code, measurements of the time exactly before and after the call of the tick function were performed. The exact time of the execution is then saved in the output variable and read from in the benchmark. The benchmarks further calculated the average jitter of a model. The jitter of a model highlights the deviation of execution times between ticks. Thus, each execution time of a tick of a model was compared to the average execution time. An average over all absolute variances was then computed and saved as the jitter.

All benchmarks measuring time were executed 20 times to correct for measuring uncertainty. Even though the server was reserved only for the benchmarks, due to Eclipse running in the background, it cannot be guaranteed that any execution of a tick was not interrupted by garbage collection or some other background functionality. After measuring the times, the *K-Best Measurement Scheme* as introduced by Bryant and O'Hallaron [BO03] was applied. This measurement scheme takes all results and measures the average of the K most appropriate results, dismissing the remaining results as outliers. Using this scheme, the average over the best 15 results was measured.

¹<https://gcc.gnu.org/>

6. Evaluation

6.2.1 Datasets

For the comparison, 640 different models were used overall. Those not only included the models of the correctness test from Section 6.1, but also further models that are presented in the following.

SCCharts Models Repository

At the time of this thesis, the private SCCharts models repository contained 713 SCCharts models of varying complexity. The models include structural tests for SCCharts, legacy models adapted to the current implementation of SCCharts, the simulation tests from Section 6.1, models of former or current problems in SCCharts as well as various anonymous models extracted from student work. Not all of those models could be used, however. Due to their age, some models were obsolete and could not compile any more. Others were not schedulable by design. Therefore, only 382 models could be used for the comparison. Of those, only 132 models including the test models from Section 6.1.1 had predefined eso traces describing their behavior. Thus only those models could be used for a meaningful comparison of the execution time. These models with traces, however, had at most 72 SCG nodes. Thus, larger executable models were required to compare models of larger size.

Additional Script Models

As shown in Section 1.2, some hypotheses concerning the priority-based compilation ought to be proven. To show that this approach struggles with execution time jitter, a model was designed to highlight this. The model as shown in its SCG form in Figure 6.2 depends on the input x . If x is true, the model only executes the statement `_term = true` and is done. Otherwise, all statements in the middle block are executed. In the priority-based compilation, if x is present, only the statement `_term = true` is executed, and nothing else. However, in the data-flow compilation, the execution still checks for the guards of all states on the right. Thus, the difference between the execution if x is true or false is far smaller. When executing the script, an integer input is taken and for each number between 3 and this input, a model is generated. The model contains as many of the sequential nodes as the current number between 3 and the input. The script further generates eso files for all models. Each file contains two traces, one that initially sets x to true and one that does not.

Random SCCharts Models

Using a generator for SCCharts model files, 59 random models were created. These models each received between one and ten boolean inputs and outputs and have a variable size between 10 and 150 states. Hierarchical and other complex behavior was allowed. Due to limitations of the random generator, very few dependencies are generated, as each additional dependency may induce a dependency cycle. To reduce the risk of the model being non-schedulable, no dependencies were added. Additionally, random traces were created that

6.2. Comparison to the Data-flow Low-Level Compilation

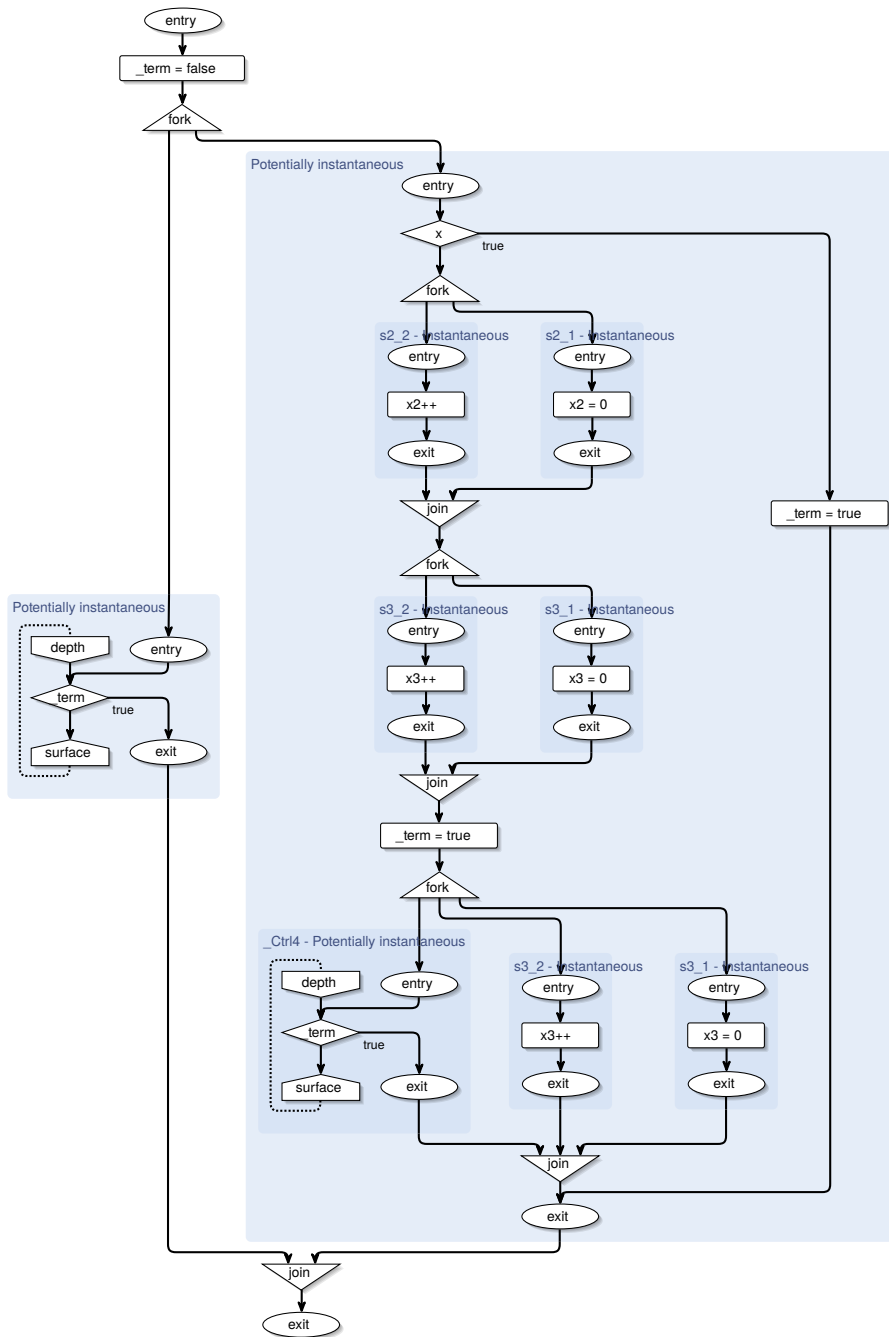


Figure 6.2. Example *JitterProgram* with three sequential concurrent regions

induced random behavior. Each model received between one and four traces with each trace spanning across one to five ticks. As these traces were not tested manually, some may do nothing or be duplicates of others. This approach is, however, still better than simply

6. Evaluation

using no inputs at all, as an execution with no inputs will not always properly describe the temporal behavior of a model. While the random inputs could all produce the same behavior as no inputs, the probability that all generated random traces across all traces result in no descriptive temporal behavior is much smaller than only executing the model once with no inputs.

6.2.2 Results

After executing the benchmark, the results were gathered in scatter plots for comparison purposes. Some of the plots were smoothed using the Locally Weighted Smoothing (LOESS) non-parametric regression method to show a trend, if it exists [CD88]. If a plot does not contain LOESS smoothing, its trend is visible without any additional visualization or no significant trend could be found. The results, divided into compilation results and execution results, are gathered in the following subsection.

Compilation

At first, the compilation results are examined. Since the results are independent from both the traces as well as the origin of the model – as long as it can be compiled by both approaches – all model groups are compared in one benchmark. Figure 6.3 depicts the compilation time of all models. The graph shows that both compilation approaches seem to have a few outliers. However, there does not exist any correlation between these outliers; e. g., one outlier in the top left comes from a randomly generated model while the other outlier in the top left comes from a homework solution. Manual compilation of these models did not yield similar results, even though the compilation duration was computed using the *K-Best Measurement Scheme* as mentioned in Section 6.2. The most probable cause is that the execution time more than five compilation runs deviated far from the average due to background processes. Therefore, at least one outlier could still influence the result heavily. However, their occurrence is important, as they might still signify potential problems. The most likely reason for these outliers is the Java runtime performing background tasks. Otherwise, the graph seems to suggest a linear increase in the compilation time for the data-flow approach while the curve of the priority-based compilation indicates a quadratic increase. This contradicts the assumptions made in Chapter 4 and Chapter 5. One assumption is that the frequency of accesses to hashmaps increases the complexity as the probability of collisions increases. This will require additional research, which could not be performed during the creation of this thesis due to time constraints. The graph further shows that the priority-based compilation is faster for smaller models, which the zoom in Figure 6.4 further supports. Only starting at roughly 800 SCG nodes, the data-flow approach scales better. When ignoring the scripted models, the data-flow compilation already starts to scale better at roughly 400 SCG nodes, yet no polynomial curve is discernible.

Parallel to the creation of this thesis, a new compiler for SCCharts was developed, which seems to speed up the data-flow compilation. Since the priority-based compilation was not

6.2. Comparison to the Data-flow Low-Level Compilation

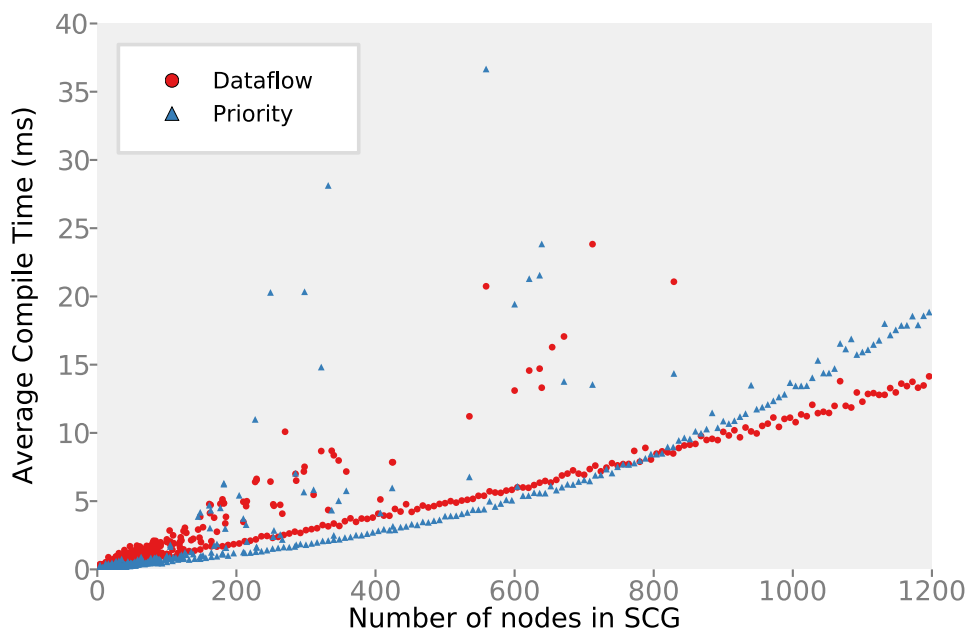


Figure 6.3. Compilation time of all models in ms

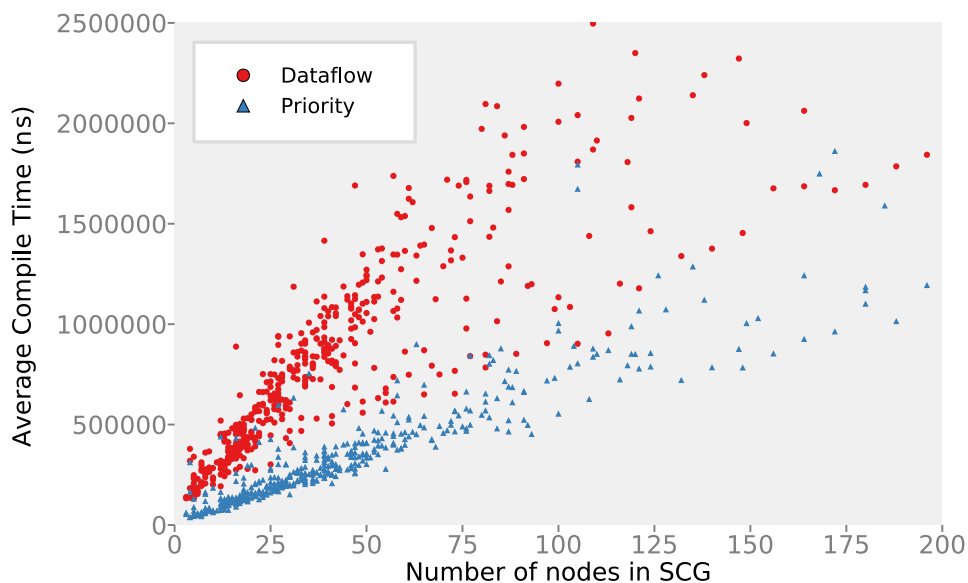


Figure 6.4. Compilation time of all models with less than 200 nodes in the SCG in ns

yet implemented in this new compiler, the benchmark could not be executed on it and the current values are obsolete. A new comparison should be performed on the new compiler as soon as the priority-based approach has been ported over.

Figure 6.5 shows how many variables were created due to the compilation. In the data-flow approach, each scheduling block, as explained in Section 3.1, requires a guard. Each guard

6. Evaluation

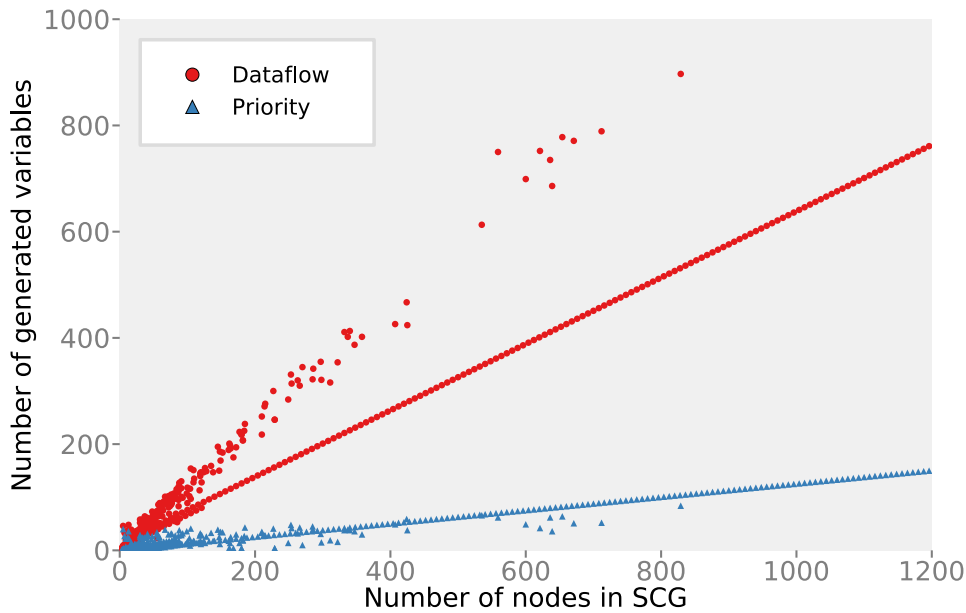


Figure 6.5. Number of generated variables of all models

itself is represented via a variable, thus the number of variables scales with the size of the model. The priority-based compilation on the other hand only requires the input and output variables as well as the variables required by the macros. However, this visualization only presents the generated variables by the compilation. The macros provide up to 50 additional global variables and up to three additional local variables. Thus, Figure 6.6 highlights the threshold, where the priority-based compilation requires fewer variables than the data-flow compilation. It shows that the former already has an advantage over the latter with fairly small models, starting at roughly 100 SCG nodes. This result shows promise for software environments with restricting virtual machines like the Lego Mindstorm NXT systems². The NXT 1.0 systems for example only allow 256 variables. Figure 6.5 shows that the priority-based compilation approach stays much longer below this threshold compared to the data-flow compilation. Even very large models with more than 1000 SCG nodes do not exceed this limit since the number of variables only scales with the number of input variables as specified by the modeler.

Figure 6.7 shows the sizes of the compiled programs in bytes. The graph shows that the data-flow compilation compiles to smaller code for small models up to 400 nodes in the SCG. Due to the overhead produced by the SCL_P macros, the priority-based compilation produces larger code for those smaller models. For larger models, starting at roughly 400 SCG nodes, the priority-based approach starts to scale better. The code generation of the data-flow compilation generates a lot of guards that have to be written for each scheduling block in the model as described in Section 3.1 to describe the control-flow of the program. This is controlled by

²<https://www.rtsys.informatik.uni-kiel.de/en/teaching/lego-mindstorms-modelrailway/lego-mindstorms-1>

6.2. Comparison to the Data-flow Low-Level Compilation

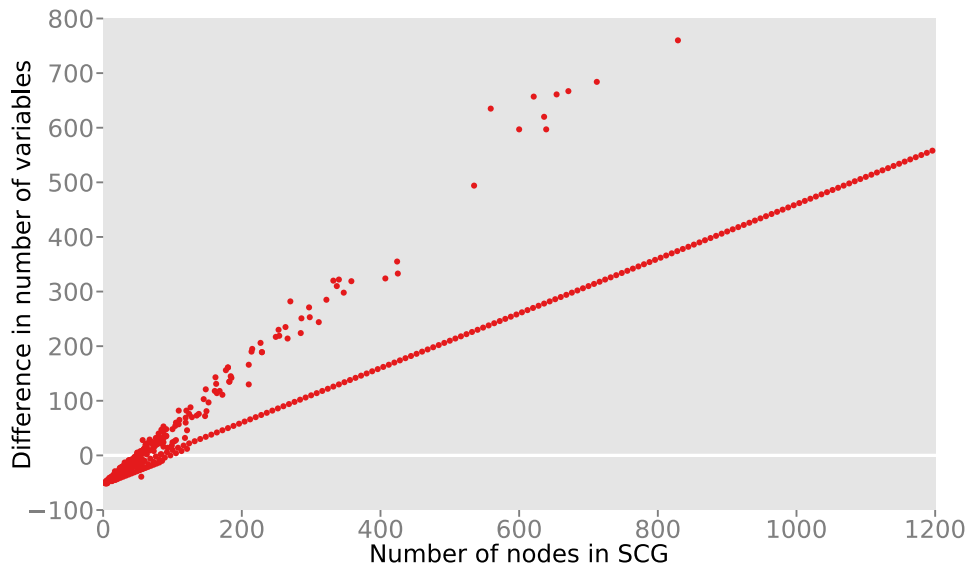


Figure 6.6. Difference in compiled variables between the priority-based and data-flow compilation approaches

the macros in the priority-based compilation. The guards are not only introduced at each incoming dependency of a node, but also at forking control-flow after conditionals. In the priority-based compilation – especially in the optimized variant – prio-statements are only introduced after nodes with outgoing dependencies. This reduces the generated code. The priority-based compilation further requires less variables as already shown in Figure 6.6 saving more memory. The graph shows another phenomenon, where the sizes seem to jump at some points. Presumably this is due to some optimizations made by the gcc compiler, as the compilation to e. g., assembler code does not produce similar results with jumps. Still, the real size of these compiled programs is compared, as only this size is important for a programmer creating code for a system with limited resources.

Execution Times

After testing the results of the compilation, the results of the **SCCharts models repository** as described in Section 6.2.1 are examined for their execution times. Figure 6.8 depicts the resulting execution times of all models **with traces** except the scripted models. The plot shows in general that the priority-based compilation does scale better in the size of the model as predicted by von Hanxleden et al. [HDM+14]. The LOESS curve indicates that the execution time is almost constant and independent from the size until roughly 200 SCG nodes. The priority-based compilation only executes active part of a model that are divided by tick borders. Thus, the execution time of a tick is only dependent on the potential length of a tick in the model. As explained in Chapter 1, a tick is supposed to be short to allow a fast reaction time. Thus, the execution time of such a model remains constant even with rising model

6. Evaluation

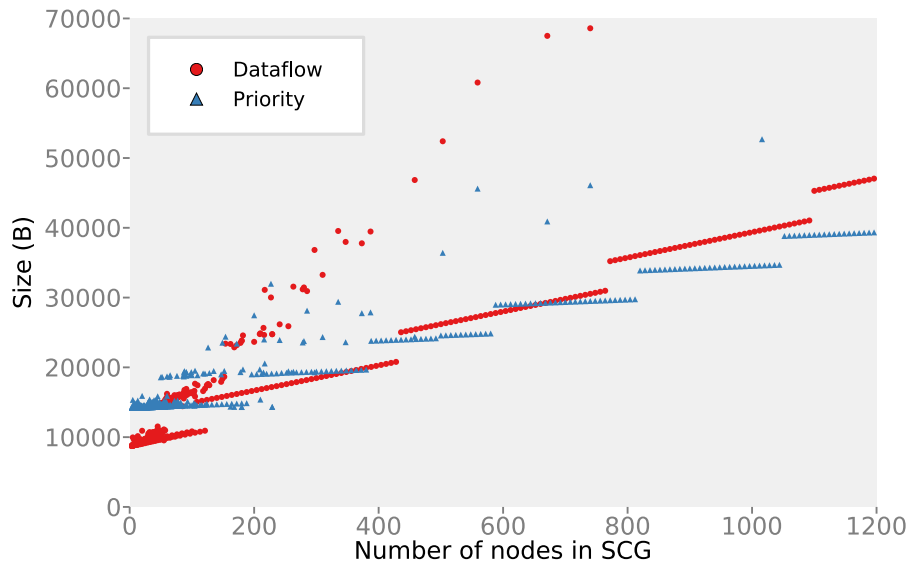


Figure 6.7. Size of the compiled C program in bytes of all models

size. After the 200 nodes, however, the low density of datapoints does not lend to a good approximation. The average execution times remain within 4000 nanoseconds. The zoom into models with less than 200 nodes in the SCG in Figure 6.9 highlights that models below 40 SCG nodes seem to perform worse in the priority-based compilation approach. This is due to the initial overhead introduced by the macros. If the execution has to setup the enabled array as well as the priorities at the beginning of the program, yet the tick function itself is just a few lines of codes long, this overhead remains significant.

The graph in Figure 6.10 then shows the execution time of the *JitterProgram* scripted models. The plot shows that for these specific models the priority-based compilation performs strictly worse than the data-flow compilation and scales worse. This is due to the high amount of forks and joins in these models. As highlighted in Figure 6.2 the models fork and join two threads multiple times sequentially depending on the number of the program. Each fork and join costs time compared to the alternative approach due to the executed macros. Since there are not only many forks and joins in the SCG, but also few other nodes in relation to the number of forks and joins, these macros represent a high amount of the time required of one tick. On the other hand, the data-flow compilation only has to calculate the guards of the join. These example models therefore execute significantly faster for the data-flow compilation on average. However, the graph highlights that the plot of the priority-based approach jumps at roughly 500 SCG nodes. As explained in Section 4.1.4, the priorities are saved in an array instead of a scalar, if the number of priorities exceed the size of an integer. At roughly 500 SCG nodes, the number of priorities in the *JitterProgram* models exceed 64. Since this is the size of an unsigned long integer on the testing system, the priority-based compilation changes to the array-based system. This adds roughly 5000 nanoseconds to the execution time of a tick and causes the execution time to scale even worse.

6.2. Comparison to the Data-flow Low-Level Compilation

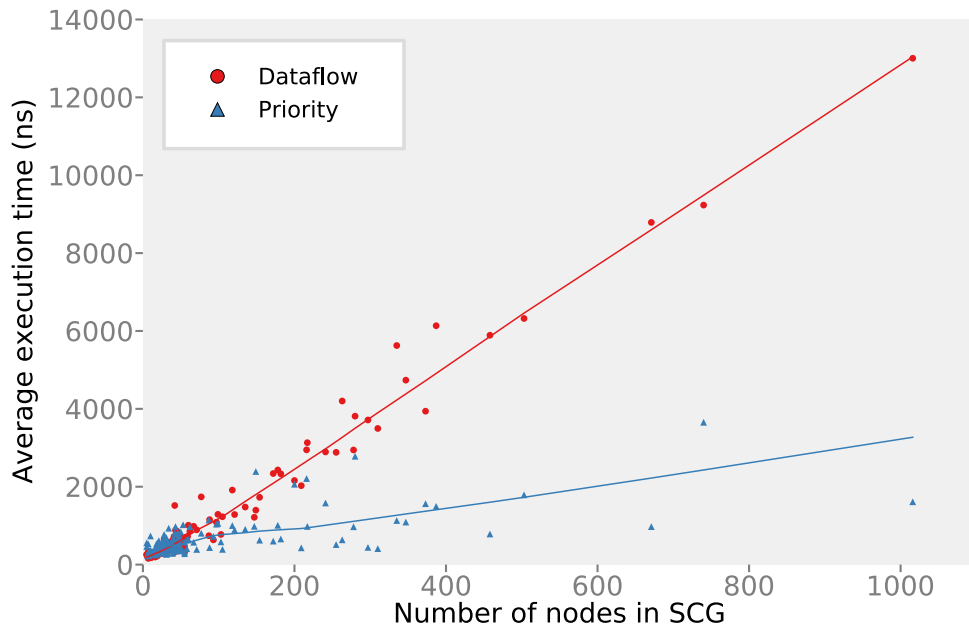


Figure 6.8. Average execution time of all non-scripted models with traces in ns

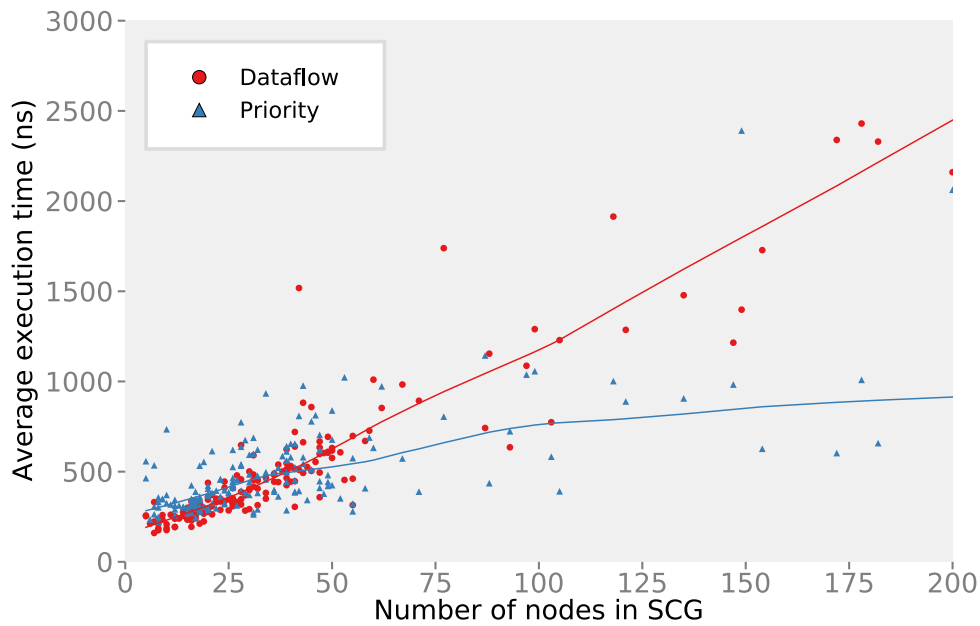


Figure 6.9. Average execution time of all non-scripted models with under 200 nodes in their SCG and traces in ns

Jitter

Next to the pure execution time, the execution time jitter was measured. The results of all models with traces already show that the priority-based compilation has an disadvantage

6. Evaluation

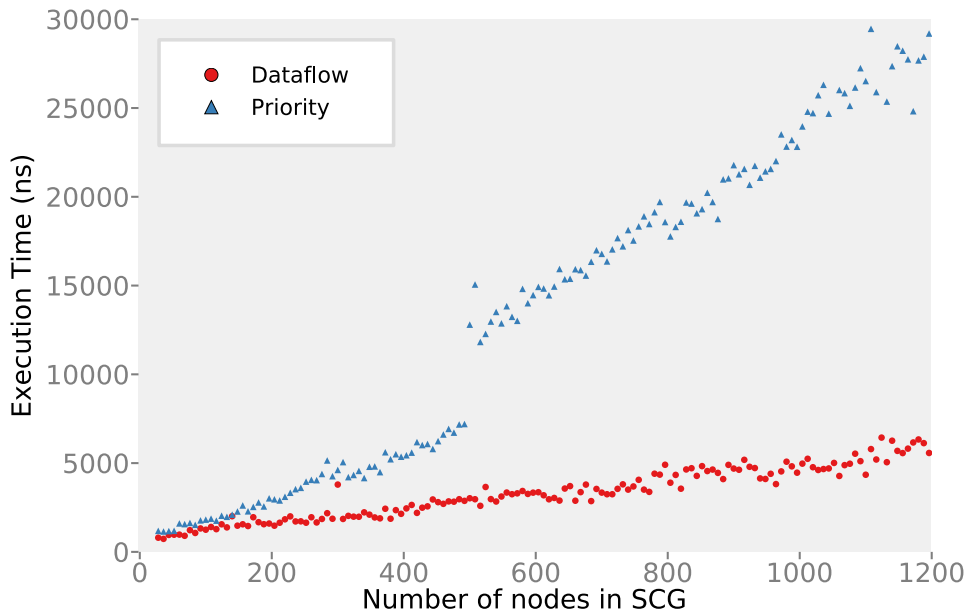


Figure 6.10. Average execution time of all scripted *JitterProgram* models in ns

over the data-flow approach as shown in Figure 6.11. The models include the *JitterProgram* models as described in Section 6.2.1. The graph not only shows that the jitter of larger models increases a lot more in the priority-based approach, but also that the jitter varies more between models. Additionally, the graph shows that for very small models, the priority-based compilation seems to perform better than the alternative. However, due to the very small deviations from the execution times in those models, this can mostly be attributed to randomness.

Figure 6.12 highlights the difference in the jitter between the two approaches. The graph shows execution times based on the model *JitterProgram148.sct* for both compilation approaches. If the input x is true, the program does nothing. If it is false on the other hand, 148 different variables are set to 0 and concurrently incremented. This happens over 148 sequential parallel statements. The blue curve shows the execution times in the priority-based compilation per tick and the red curve the execution times of the data-flow compilation per tick. It shows how the red curve stays relatively constant within 5000 to 7500 ns execution time, whereas the blue curve fluctuates between 17500 and 35000 ns depending on whether the input x was set or not. The higher execution times of the priority-based compilation can again be explained by its behavior. Since it only executes active parts of the program, each other tick performs the 148 sequential states while the remaining ticks execute only a single node as the SCG in Figure 6.2 shows.

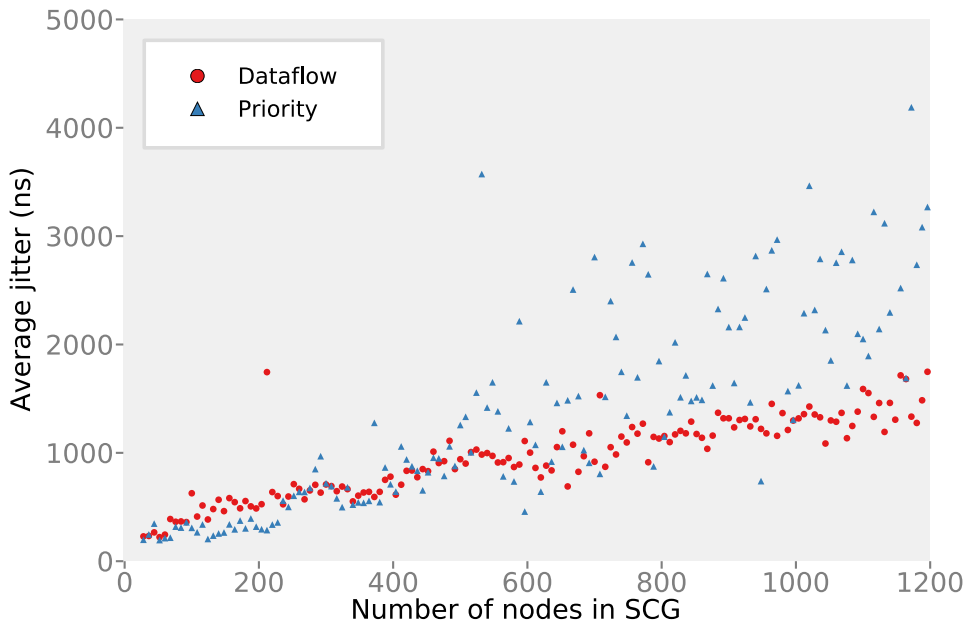


Figure 6.11. Average jitter of all models with traces in ns

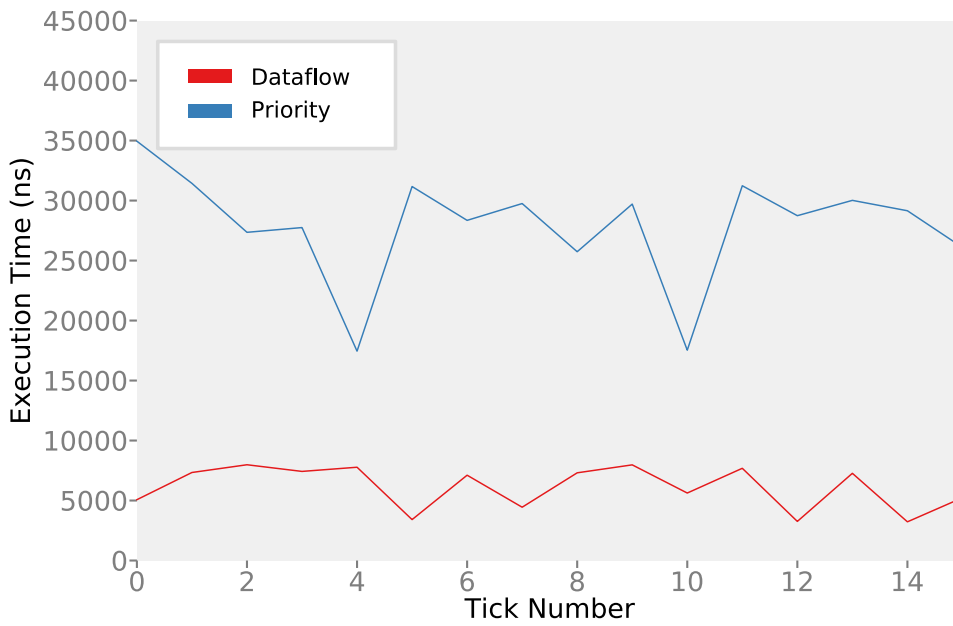


Figure 6.12. Example execution times per tick of the *JitterProgram148.sct* model exemplifying the jitter

6.3 Optimization Comparison

To see whether the optimization of the `npr` assignment has any significant impact on the execution time, the optimizations were turned off and another time measurement was started.

6. Evaluation

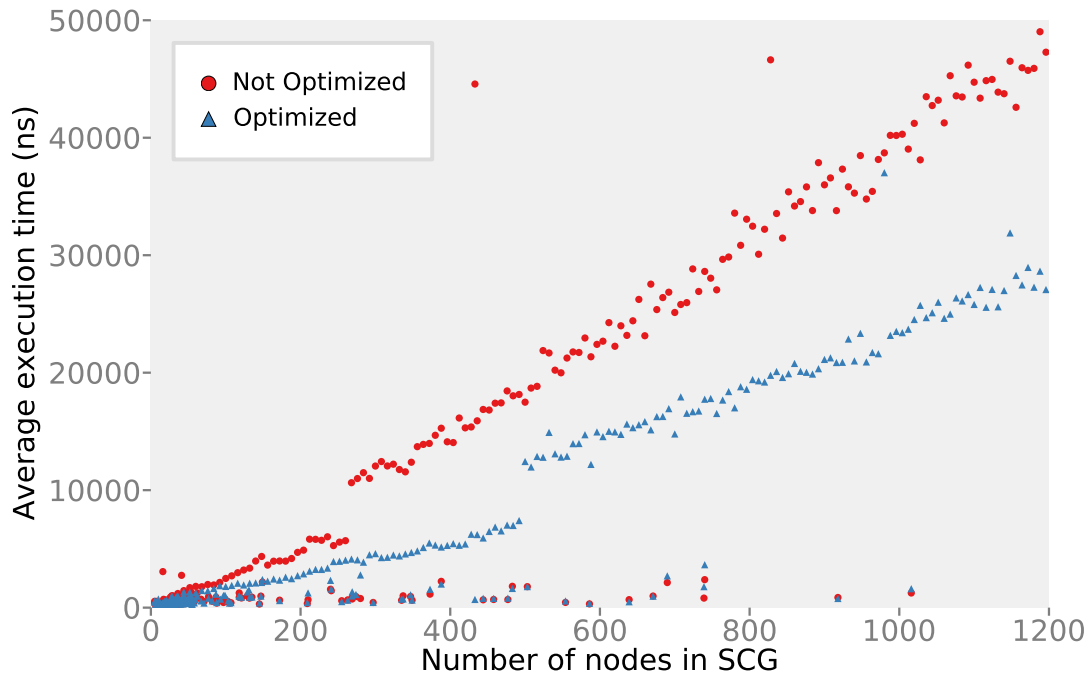


Figure 6.13. Average execution times of the optimized and non-optimized priority-based compilation approach

The results as shown in Figure 6.13 stress that the optimizations increased the performance. On average, the execution time sank by 20%. The graph further shows the effect of the reduced total number of priorities in the model. This means that the non-optimized variant needs to use the array-based implementation a lot earlier than the optimized variant. For the non-optimized variant this means that at roughly 250 SCG nodes the number of priorities exceed 64, while the optimized variant exceeds this limit first at 500 SCG nodes. These datapoints, however, are largely influenced by the *JitterProgram* scripted models. Without these, the difference is not as distinct. Due to the low number of dependencies and more often low *npr* numbers in these models, the effect of the optimizations is low. The outliers at the bottom of the graph show those models that have mostly a low number of dependencies. Even those models that have a high number of dependencies often have low *nprs* as often most dependencies originate in one single node. In those models only a single node requires a higher *npr*.

6.4 Low-Level Implementation Evaluation

An important aspect of the SCL_P macros is the effectiveness of priorities below a certain threshold. Due to the scalar-based check for enabled threads, joins can be performed very fast. However, a scalar cannot be indefinitely large in a computer system. Therefore, the array-based implementation is required, once the number of priorities exceeds the size of an integer

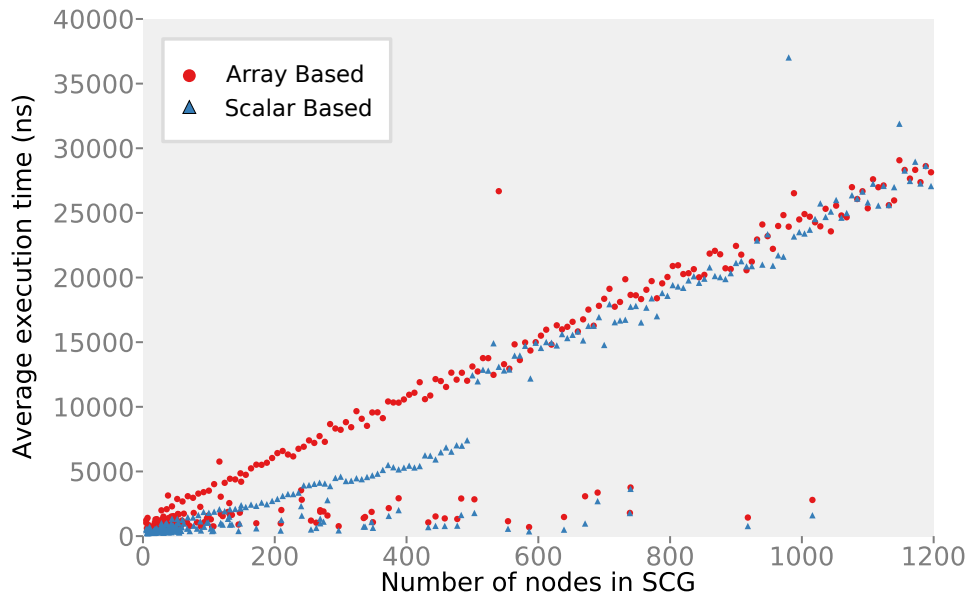


Figure 6.14. Average execution times of the priority-based compilation approach with only array-based implementation compared to scalar-based implementation

on the system. Figure 6.14 illuminates the difference between these two implementations. Both curves show the average execution time of the priority based approach over various models. The blue curve, however, only implements the array-based priorities, while the red only switches to the array-based implementation above 64 priorities. The graph shows that the scalar-based implementation is up to 50% faster than the array-based implementation depending on the model. In all models, where the scalar-based implementation works, it is the faster alternative. Beginning at roughly 500 SCG nodes, the scalar-based implementation stops working as the number of priorities exceed 64 and both implementations achieve the same results.

Conclusion

The following chapter summarizes the presented approach to an alternative low-level compilation of SCCharts. After the summary in Section 7.1, Section 7.2 gives a small outlook to further work to extend this approach.

7.1 Summary

This thesis describes a compilation approach towards SCGs and low-level SCCharts based on priorities and described further optimizations and enhancements to this approach. This approach includes assigning priorities to nodes in an SCG and using this annotated SCG to generate C as well as Java code. A simple schedulability analysis performed during the priority assignment determines whether a program written in SCCharts or an SCG is translatable or not.

At the start, node priorities are assigned to all nodes using a depth-first search over all nodes in the provided SCG. These determine a mandatory order that the nodes have to be executed by. A node that executes concurrently to another node must be executed earlier, if it has a higher node priority. Afterwards, a fixed and deterministic order of execution is calculated based on the node priority and on the associated thread of a node, its `tsID`. The resulting ID, the `prioID` of a node, is then optimized to the optimized `prioID`. Using this number, code is finally generated from the annotated SCG.

Using a newly developed approach towards `npr` assignment, the node priority assignment was optimized. The new assignment reduces unnecessary context switches in almost all concurrent models. Fewer context switches lead to less overhead by the dispatcher as well as better locality for caching and thus to better performance, which is especially important for safety-critical and embedded systems. Additionally, the approach was adapted to allow schizophrenic models. Previously, some schizophrenic models – namely those containing concurrent, but not runtime concurrent models – were not accepted. Now, when preparing for the schedulability analysis and `npr` assignment, schizophrenic statements are detected and any illegal dependency cycles removed, if possible.

Afterwards, both the translation as well as the optimizations were implemented in the context of the KIELER project. This translation provides an alternative way to generate deterministic C and Java code from SCCharts or SCG models. It not only translates all SASC and many ASC programs, but also all SIASC and many IASC programs. The implementation further shows that a translation into a new language is easily possible due to the modularity and

7. Conclusion

flexibility provided by the KIELER compiler.

The evaluation then showed that the new compilation approach is not only comparable in compilation speed with the data-flow approach, but even outperforms it in average execution speed in most models. Even considering the larger execution-time jitter of programs created using the priority-based compilation, the WCET of programs is shorter even for some smaller and medium sized models and especially for large models. These findings support the hypotheses proposed by von Hanxleden et al. in their paper covering high-level and low-level compilation of SCCharts [HDM+14]. Further findings suggest that the priority-based approach in general shows promise for large scale models. Not only does the execution time scale better in the size of the model, but the size of the compiled program and the number of required variables also scale better. This can play an important role when dealing with systems with limited resources.

7.2 Future Work

The main focus of this thesis was to develop and enhance the priority-based compilation for SCCharts in the context of the KIELER project. However, due to the time constraints during this thesis, not all goals and ideas could be achieved and pursued. Thus, this section talks about further ideas that can be pursued after the completion of this thesis.

7.2.1 Further Optimizations

As already mentioned in Section 7.2.1, some unnecessary context switches cannot yet be reduced. Further small optimizations could also be done towards improving the tsID assignment. Currently, the tsIDs of child threads are assigned depending on their npr at the exit node. However, for example due to locality some threads may better be scheduled directly after each other. A better way to assign tsIDs should therefore be first evaluated and then implemented.

Additionally, an alternative implementation of the C macros using a parent-children relation as originally intended in for example the SyncCharts transformation from Section 3.3 could be useful for a comparison. The intrusion into the priority assignment itself would be minimal and confined to the tsID assignment. Only the code assignment ought to be adapted to the new fork- and join-macros.

Lastly, Section 6.4 highlights that the array-based implementation for priorities is slower than the scalar-based implementation. An extension to the macros that uses only integers without the need for arrays could significantly increase the performance of the execution. Simple macros that control which scalar should be checked can be implemented to replace the check for the array. Thus, macros that at compile time create a number of scalars and control structures to address the correct scalar can be created. Alternatively, a study to find how large the priorities realistically get in average models may find that more than 64 priorities occur rarely. Then, such an improvement would be insignificant and only improve border cases.

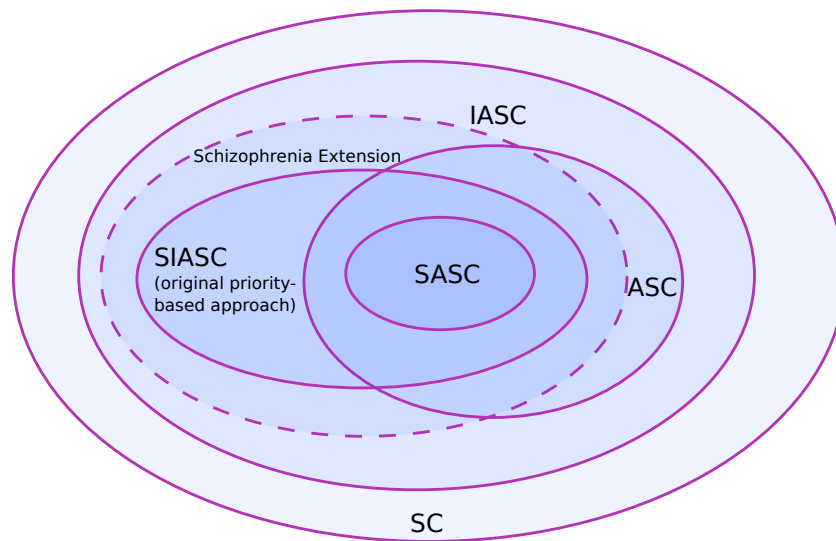


Figure 7.1. The classes of SC, extended by the schizophrenia adaptations of this thesis

Otherwise, the study could find the optimal number of scalars. If, e. g., an implementation using five scalars covers 90% of all models, then the implementation could be enhanced to using five scalars by default.

7.2.2 Classification of Compilable Models – Improvements

The class of compilable programs has been extended beyond only SIASC programs and into IASC programs. Figure 7.1 shows a simplified representation of the compilable programs by the priority based compilation based on the earlier representation in Figure 2.4. While certainly not all IASC programs can be compiled – e. g., models with unreachable dependency cycles –, the expansion allows some other models that were not schizophrenic but IASC schedulable to be compilable. However, some schizophrenic models are still not accepted due to the conservative condition that no dependency cycle may cross through a join as well as into a thread not joined by the join. Figure 7.2 depicts a model that is rejected due to this restriction, but does not violate the iur protocol. Without the feedback loop through the join no cycle would be present. It can also be seen that no second instances of the nodes in B2 are reachable from the entry of the thread, therefore it is impossible for $l = 0$ to preempt or be preempted by any concurrent nodes. The cycle induced by the dependency from $R = \text{true}$ to $O = R \mid \text{true}$ is between nodes whose instances are never runtime concurrent. Thus, the whole cycle should not exist. A solution to this problem could be to eliminate all dependencies between nodes that cannot be runtime concurrent as in the example above. This not only improves problems in these schizophrenic models but also for arbitrary models with cyclic dependencies between nodes that are not concurrent at runtime. Analyzing all dependencies for runtime concurrency may prove difficult however. Alternatively, one might take a detailed look at the source and target of the dependencies. If one of them is in the surface of its thread,

7. Conclusion

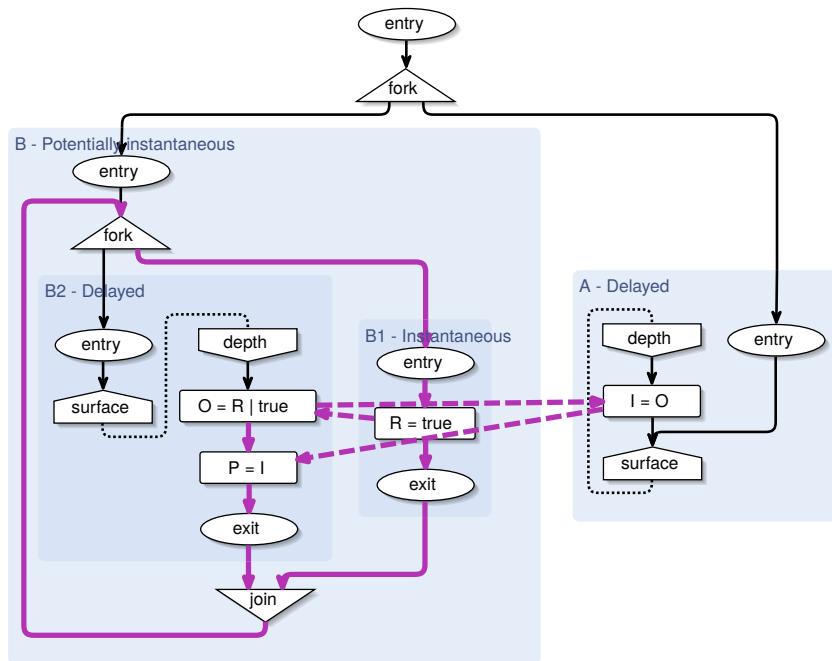


Figure 7.2. A schizophrenic model that is still not schedulable

while the other is in the depth of a delayed thread, the dependency can be eliminated and the model accepted.

Further models that are still not schedulable are those where parts of the model are unreachable as in Figure 4.7 or other models where dependency cycles only occur statically, but never during a run of the model. A schedulability analysis for such models would have to take the contents of assignments and conditionals into account. A static scheduling analysis such as the one introduced in this thesis is not well suited towards such a task.

7.2.3 Further Alternative Compilations

In the design of safety-critical systems, various different standards are used that restrict the use of the language for safety purposes. Some of these standards, e. g., the MISRA C 2004 standard [MIS04], ban the use of the goto statement. Thus, the current implementation of the SCL_P macros would be prohibited for these standards. To comply to these standards, one could use the switch-case based implementation as used in SJ in C as well.

Some additional restrictions apply when implementing the *statemachine pattern*. A general overview over various implementations of the statemachine pattern in modeling tools and other projects is given by Domi et al. [DPR+12]. The generated code of this pattern should be easy to follow, as free from overhead as possible and, if changes to the source model are made, those changes should be minimally invasive to the generated code and only change locally. Typically, statemachines do not implement concurrency. Thus, an initial implementation of

the statemachine pattern could translate all modeled states as they are and introduce a switch-case structure that decides which state to execute next. If a transition between states is taken, the switch-case statement should be left and re-entered, analogous to the implementation in SJ. To improve readability the cases should be named after the states in the model. In KIELER SCCharts, this translation would have to draw on the core SCCharts step of the translation as depicted in the upper halve of Figure 1.1, as the statemachine pattern cannot handle features like strong aborts or history transitions. After the core SCCharts step, the model loses a lot of information about states. Additionally, one has to analyze, if and how hierarchical behavior should be implemented. Since the resulting code should be certifiable and easy to read, nested behavior should be restricted.

If concurrency is to be implemented as well, one could think about using an adaptation of the priority-based implementation. Since the statemachine pattern builds upon the core SCCharts step, the SCG is never created and a priority assignment cannot be performed. One solution would be to adapt both the dependency analysis as well as the priority assignment to work with core SCCharts. One state in the model would then be regarded as an atomic statement and interleaving would have to be prohibited. The priority assignment then creates a schedule of these atomic states. Another solution could be to still perform the dependency analysis and priority assignment on an intermittently created SCG. The result of these analyses would then have to be traced back to the original states, allowing an interleaving of states. It is questionable, however, whether this behavior is desired.

Bibliography

- [And13] Sidharta Andalam. “Predictable platforms for safety-critical embedded systems”. PhD thesis. The University of Auckland New Zealand, 2013.
- [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.
- [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.
- [Ber00a] 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.
- [Ber00b] Gérard Berry. “The foundations of Esterel”. In: *Proof, Language, and Interaction: Essays in Honour of Robin Milner*. Ed. by Gordon Plotkin, Colin Stirling, and Mads Tofte. Cambridge, MA, USA: MIT Press, 2000, pp. 425–454. ISBN: 0-262-16188-5.
- [Ber05] G. Berry. “Esterel v7: from verified formal specification to efficient industrial designs”. In: *Fundamental Approaches to Software Engineering FASE 2005*. Ed. by M. Cerioli. Vol. 3442. LNCS. Springer, 2005, pp. 1–1.
- [BO03] Randal E Bryant and David Richard O’Hallaron. *Computer systems: a programmer’s perspective*. Vol. 2. Prentice Hall Upper Saddle River, 2003.
- [CD88] William S Cleveland and Susan J Devlin. “Locally weighted regression: an approach to regression analysis by local fitting”. In: *Journal of the American statistical association* 83.403 (1988), pp. 596–610.
- [DPR+12] Eladio Domí, Beatriz Pérez, Ángel L Rubio, et al. “A systematic review of code generation proposals from state machine specifications”. In: *Information and Software Technology* 54.10 (2012), pp. 1045–1066.
- [Edw99] Stephen A. Edwards. “Compiling Esterel into Sequential Code”. In: *Proceedings of the 7th International Workshop on Hardware/Software Codesign (CODES 99)*. <http://www1.cs.columbia.edu/~sedwards/papers/edwards1999compiling.pdf>. May 1999.
- [EKH06] Stephen A Edwards, Vimal Kapadia, and Michael Halas. “Compiling esterel into static discrete-event code”. In: *Electronic Notes in Theoretical Computer Science* 153.4 (2006), pp. 117–131.
- [Fuh11] Hauke Fuhrmann. “On the pragmatics of graphical modeling”. Dissertation. Kiel: Christian-Albrechts-Universität zu Kiel, Faculty of Engineering, 2011.

Bibliography

- [Han09a] Reinhard von Hanxleden. *SyncCharts in C*. Technical Report 0910. Christian-Albrechts-Universität zu Kiel, Department of Computer Science, May 2009.
- [Han09b] Reinhard von Hanxleden. “SyncCharts in C—A Proposal for Light-Weight, Deterministic Concurrency”. In: *Proc. Int’l Conference on Embedded Software (EMSOFT’09)*. Grenoble, France: ACM, Oct. 2009, pp. 225–234.
- [Har87] David Harel. “Statecharts: A visual formalism for complex systems”. In: *Science of Computer Programming* 8.3 (June 1987), pp. 231–274.
- [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, and Owen O’Brien. “Sequentially Constructive Concurrency—A conservative extension of the synchronous model of computation”. In: *Proc. Design, Automation and Test in Europe Conference (DATE’13)*. Grenoble, France: IEEE, Mar. 2013, pp. 581–586.
- [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.
- [Lee06] Edward A. Lee. “The problem with threads”. In: *IEEE Computer* 39.5 (2006), pp. 33–42.
- [MHH13] Christian Motika, Reinhard von Hanxleden, and Mirko Heinold. “Programming deterministic 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.
- [MIS04] MISRA. *Guidelines for the use of the c language in critical systems*. 2004. ISBN: 0952415623.
- [Mot17] Christian Motika. *Sccharts—language and interactive incremental implementation*. Kiel Computer Science Series 2017/2. Dissertation, Faculty of Engineering, Christian-Albrechts-Universität zu Kiel. Department of Computer Science, 2017. ISBN: not assigned yet.

- [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.
- [PS04] Dumitru Potop-Butucaru and Robert de Simone. “Optimization for faster execution of Esterel programs”. In: Norwell, MA, USA: Kluwer Academic Publishers, 2004, pp. 285–315. ISBN: 1-4020-8051-4.
- [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.
- [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.
- [Sch16] Alexander Schulz-Rosengarten. “Strict sequential constructiveness”. <http://rtsys.informatik.uni-kiel.de/~biblio/downloads/theses/als-mt.pdf>. Master thesis. Kiel University, Department of Computer Science, Sept. 2016.
- [SMH15] Steven Smyth, Christian Motika, and Reinhard von Hanxleden. “A data-flow approach for compiling the sequentially constructive language (SCL)”. In: *18. Kolloquium Programmiersprachen und Grundlagen der Programmierung (KPS 2015)*. Pörschach, Austria, May 2015.
- [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.
- [SW01] Klaus Schneider and Michael Wenz. “A new method for compiling schizophrenic synchronous programs”. In: *Proceedings of the 2001 international conference on Compilers, architecture, and synthesis for embedded systems*. ACM. 2001, pp. 49–58.
- [TAH10] Claus Traulsen, Torsten Amende, and Reinhard von Hanxleden. *Compiling SyncCharts to Synchronous C*. Technical Report 1006. Kiel, Germany: Christian-Albrechts-Universität zu Kiel, Department of Computer Science, July 2010.
- [TAH11] Claus Traulsen, Torsten Amende, and Reinhard von Hanxleden. “Compiling SyncCharts to Synchronous C”. In: *Proceedings of the Design, Automation and Test in Europe Conference (DATE’11)*. Grenoble, France: IEEE, Mar. 2011, pp. 563–566.
- [Tar72] Robert Tarjan. “Depth-first search and linear graph algorithms”. In: *SIAM journal on computing* 1.2 (1972), pp. 146–160.

Bibliography

- [TS04] Olivier Tardieu and Robert de Simone. “Curing schizophrenia by program rewriting in esterel”. In: *Formal Methods and Models for Co-Design, 2004. MEM-OCODE’04. Proceedings. Second ACM and IEEE International Conference on*. IEEE. 2004, pp. 39–48.