Reactive Processing for Synchronous Languages and its Worst Case Reaction Time Analysis

Dissertation

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

Claus Traulsen

Kiel
2010

1. Gutachter	Reinhard von Hanxleden
2. Gutachter	Michael Mendler
3. Gutachter	Partha Roop
Datum der mündlichen Prüfung	26. Februar 2010

Abstract

Many embedded systems belong to the class of *reactive systems*. These are systems that have to react continuously to the environment at a rate that is determined by the environment. Reactive systems have two specific characteristics: their control flow requires concurrency and preemption, and, since the reactive systems are often safety-critical, we must be able to prove the correctness of the behavior and of the timing. To implement reactive systems, the *synchronous languages* were developed, which have a clear mathematical semantics and allow the expression of concurrency and preemption in a deterministic way. Programs in a synchronous language can be either compiled to software and run on a common processor, they can be synthesized to a hardware description, or a software/hardware co-design approach can be taken. However, the compilation of synchronous hardware into efficient code is not trivial.

To improve the efficiency of the execution and at the same time simplify the compilation, *reactive processors* were introduced, which have an instruction set architecture that is inspired by synchronous languages. In particular, reactive processors have direct support for preemption and concurrency. Furthermore, these processors optimize the worst case reaction time, in contrast to common processors which optimize the average case reaction time. This simplifies the timing analysis, which is necessary to prove that a system meets its timing requirements.

This thesis presents three contributions to reactive systems:

- A formal semantics is given to the Kiel Esterel Processor (KEP), a reactive processor to execute the synchronous language Esterel. Also a compilation scheme from SyncCharts to the KEP assembler is presented, in addition to the existing compilation from Esterel into KEP assembler.

- The Kiel Lustre Processor is introduced, a reactive processor for the synchronous dataflow language Lustre, which allows true parallel execution with multiple processing units.

- Different approaches for the worst case reaction time analysis of KEP programs are presented: a search for the longest execution path in the KEP assembler, a formal modeling of the execution times based on interface algebras. Also an approach to use model checking to analyze the reaction time is applied to the KEP.

Contents

List of Figures

1. Introduction

Reactive systems are control systems that have to react continuously to inputs at a rate which is determined by their physical environment. The control-flow of such systems differs from that of standard computer systems: the systems are inherently concurrent, since they have to deal with a physical environment that itself is concurrent. The control can often switch between different *modes*, hence parts of the systems have to be suspended or aborted. Since these systems are often highly safety-critical, the behavior of the controller should always be deterministic. This is not only true for the functional behavior, but for the timing as well. Hence, for reactive systems we identify the following key issues:

- concurrency,
- preemption,
- determinism, and
- timing predictability.

In order to match these needs, synchronous languages, such as Esterel [Potop-Butucaru et al., 2007], Lustre [Halbwachs et al., 1991a], and Signal [Guernic et al., 1991] were introduced. The execution of these languages is divided into discrete *ticks* or *instants*. Parallelism is supported by logical concurrency, which is usually sequentialized for the execution. While Esterel is an imperative, control-oriented language, Lustre and Signal are dataflow languages. Since the design of reactive systems is often done by engineers who have a background in control theory, rather than in computer-science, a dataflow formalism is a good means to describe these systems. Traditionally, synchronous languages are either synthesized to hardware, or compiled into C code and executed on standard processors. However, this compilation is not trivial, because common processors do not support the reactive control flow, in particular concurrency and preemption.

1.1. Reactive Processing

Reactive processing supports the key issues of reactive systems, in particular concurrency and preemption, directly in hardware, while still allowing the flexible compilation from synchronous languages for this hardware. The timing predictability is achieved, by having a processor design that simplifies the timing analysis. Reactive processors can either be new processor designs from scratch, or a patch to existing processors. The *KEP* [Li, 2007], the EMPEROR [Yoong et al., 2006] and the STARPro [Yuan et al.,

2008] are reactive processors that can directly execute Esterel programs. The *KLP*, presented in this thesis, is a reactive processor developed to execute Lustre programs. While the restriction to an input language limits the class of executable programs, it greatly simplifies both processor design and timing analysis. Reactive processors can be seen as an Application Specific Instruction-set Processor (*ASIP*), or, since they are not designed for a specific application but for the whole application area, namely reactive systems, as an Application Area Specific Instruction-set Processor (*AASIP*).

Beside the reactive processors, there are also other recent approaches to build a processor which allows a tight timing analysis, such as the PReT [Lickly et al., 2008] or the Predator project[1]. The goal of these approaches is to design a general purpose processor with timing analysis in mind, while still allowing the execution of arbitrary C code. For reactive systems, we do not focus on the Worst Case Execution Time (*WCET*) of a program, *i. e.*, the maximal execution time for a given piece of code, but on the Worst Case Reaction Time (*WCRT*), *i. e.*, the maximal time that outputs are generated after new inputs are read. One can express the WCRT as the WCET of the function that computes one reaction. However, the WCRT usually depends on the internal state of the program, hence the WCRT analysis should consider the different modes of the system.

From the reactive processing approach, we expect multiple benefits: deterministic behavior and timing, better resource usage including program size, and dependability. Reactive Processors are also interesting from a theoretical point of view: they give an operational semantics to programs that are not valid in the traditional semantics of their input language. We will now take a closer look at these issues:

Precise Timing

One of the main problems when designing reactive systems is to determine the exact execution time. To determine a tight WCET for the execution on modern systems is a hard problem, since they are optimized for the average case execution time. Synchronous languages, in contrast, optimize for the worst-case. For synchronous languages this problem can be reduced to finding the maximal number of instructions that are executed within one instant, the so called WCRT [Boldt et al., 2008]. The WCRT analysis is simpler, because the design of reactive processors is simpler than that of standard processors. On the other hand, they directly support high level constructs, like preemption, in which execution time can be easily analyzed. To analyze the execution time of the software implementation of the same preemption is much harder.

Dependability

Developing robust and correct software for reactive systems is not trivial. While formal verification (*e. g.* model checking) can prove that the model is correct with respect to its specification, the correctness of the actual system also depends on the (possibly implicit) assumptions that were made on the environment and on the correctness of the target

[1] `www.predator-project.eu`

platform and the compiler. While there exist certified compilers for Esterel and *Scade*, this only assures that the compilers were developed with a specified, robust methodology, but it does not assure that they are actually correct, *i. e.*, always produce code that behaves the same way as the model. In particular, when the models are translated into a subset of C, as it is done for Esterel and Scade, it is hard to assure the correctness of the compilation.

Here reactive processors can help by giving a simple instruction set architecture which is specially designed to support the modeling language. This makes the compilation process easier and should allow for a provably correct compiler. Of course this also implies that the processor itself must be proven to be correct, but this is a simpler task than proving the correctness of software for general-purpose processors.

Another benefit is the increased traceability. The compilation of synchronous programs to traditional instruction sets makes it very hard to determine which statement in the source program led to which assembler instruction. In contrast, the instruction set of a reactive processor allows for a direct mapping between the assembler and the source model. Hence low-level debugging can be performed on the high-level model.

Deterministic behavior

One of the advantages of synchronous programs is their deterministic behavior, which is independent from the scheduling. This is also true in complex situations, *e. g.*, multiple interrupts, without relying on the precise timing as common processors do. Reactive processors enforce determinism in two ways. First, like synchronous languages, they sample their inputs so that all input values are unique in one tick. Second, events occur either simultaneously, or they are separated by at least one tick. Reactive processors avoid race conditions, either by mapping concurrent threads to hardware threads with clearly defined switch points, or by implementing concurrency directly, but enforcing the uniqueness of all values within a tick. To achieve this, either the compiler or the processor itself has to assure that all writes to a specific signal are performed before the signal is read.

Resource Usage

An important issue for embedded controllers is the resource usage. While the power consumption for PCs can often be neglected, this is not true for embedded devices that are sometimes supposed to run for years on a simple battery. Reactive processors might be less efficient than common processors in performing simple tasks, such as multiplying two numbers. They are, however, more efficient in performing high level tasks that are typical for reactive control flow.

Embedded devices have usually a small amount of RAM and ROM. A more direct instruction set leads to a more compact representation of the code size. For Esterel a virtual machine was developed [Plummer et al., 2006], just to achieve smaller object codes. Reducing the code size is already a benefit for itself, but it has even further advantages. *E. g.*, when the program completely fits in the instruction cache, this not

only increases the execution speed, but also simplifies the analysis of the worst case reaction time.

Class of valid programs

In order to allow a static scheduling, synchronous languages impose constraints on the class of accepted programs. While Lustre only allows acyclic programs, without any static cycles, Esterel allows all programs without dynamic cycles, so called constructive program [Berry, 1999]. While cyclic, constructive programs can be transformed into equivalent constructive programs [Lukoschus and von Hanxleden, 2007], detecting whether a cyclic program is constructive is co-NP-complete [Malik, 1994], therefore recent Esterel compilers [Potop-Butucaru et al., 2007] only allow acyclic programs.

However, reactive processors simply execute any program they get. So what shall we consider as a valid program for a reactive processor? Both for the *KEP* and the processor presented here, the compiler assumes so far acyclic programs. But this is just a matter of implementation, since both use existing front-ends, the cec and the lus2ec, respectively, and has no theoretical reason. Since Lustre does not give a reference semantic for cyclic programs, we can simply assume that any cyclic program for which the processor will not deadlock is a valid program.

1.2. Contributions

This works build on existing work on reactive processors, in particular on the *KEP*. I contribute three different aspects:

1. I extend the *KEP* by a formal semantics and by an additional compilation path.

2. I show how reactive processing can be applied to synchronous dataflow, in particular to Lustre, and how real parallel execution can be used, instead of the pure logical concurrency, which is still sequentialized for at run-time, which is used for the *KEP*. I also show how the approach can be used for Scade models, a commercial variant of Lustre, which combines data-flow with automata.

3. I apply different approaches to WCRT analysis for reactive processing on the *KEP*.

I also connected the *KEP* and the *KLP* with a common interface to the Kiel Integrated Environment for Layout for the Eclipse Rich Client Platform[2] (*KIELER*) tool, to allow the compilation and the execution of programs on a reactive processor within on tool. While the *KEP* and the *KLP* are distinct processors, they have some common supporting tools like the Kiel Reactive Processor (*KReP*) evaluation bench. So the *KReP* is the combination of all reactive processors from Kiel, not one processor.

[2] `www.informatik.uni-kiel.de/rtsys/kieler/`

1.3. Related Publications

Parts of this thesis were already published in research papers:

- Falk Starke, Claus Traulsen, and Reinhard von Hanxleden. Executing Safe State Machines on a reactive processor. Technical Report 0907, Christian-Albrechts-Universität Kiel, Department of Computer Science, Kiel, Germany, March 2009

  This technical report describes the code generation from SyncCharts to *KEP* assembler, which is explained in Section 4.5.

- Claus Traulsen and Reinhard von Hanxleden. Reactive parallel processing for synchronous dataflow. In *Proceedings of the 25th Symposium On Applied Computing (SAC'10), Special Track Embedded Systems: Applications, Solutions, and Techniques*, Sierre, Switzerland, March 2010

  The explanation of the *KLP* in Chapter 5 is an extended version of this paper.

- Marian Boldt, Claus Traulsen, and Reinhard von Hanxleden. Worst case reaction time analysis of concurrent reactive programs. *Electronic Notes in Theoretical Computer Science*, 203(4):65–79, June 2008. Proceedings of the International Workshop on Model-Driven High-Level Programming of Embedded Systems (SLA++P'07), March 2007, Braga, Portugal

  The explanation of the WCRT analysis of the *KEP* by finding the longest path in the Concurrent KEP Assembler Graph (*CKAG*) in Section 6.1 is taken directly from this paper.

- Michael Mendler, Reinhard von Hanxleden, and Claus Traulsen. WCRT Algebra and Interfaces for Esterel-Style Synchronous Processing. In *Proceedings of the Design, Automation and Test in Europe (DATE'09)*, Nice, France, April 2009

  This paper details the usage of an interface algebra for the WCRT analysis. Some parts of the paper are used for the introduction in Section 6.2. The author was responsible for the implementation of the approach.

- Partha S. Roop, Sidharta Andalam, Reinhard von Hanxleden, Simon Yuan, and Claus Traulsen. Tight WCRT analysis for synchronous C programs. Technical Report 0912, Christian-Albrechts-Universität Kiel, Department of Computer Science, Kiel, Germany, May 2009a

  This paper presents the model-checking approach to determine the WCRT of PRET-C programs, parts of the papers are used in Section 6.3. In this thesis the approach is for the first time applied to *KEP* assembler.

Also the corresponding parts of the related work in Section 2 are based on the related work sections in these papers.

The work presented in this thesis builds on the following diploma theses, which were supervised by the author:

- Marian Boldt. Worst-case reaction time analysis for the KEP3. Study thesis, Christian-Albrechts-Universität zu Kiel, Department of Computer Science, May 2007a. `http://rtsys.informatik.uni-kiel.de/~biblio/downloads/theses/mabo-st.pdf`

  The topic of this thesis was the implementation of a WCRT analysis directly in the strl2kasm compiler.

- Marian Boldt. A compiler for the Kiel Esterel Processor. Diploma thesis, Christian-Albrechts-Universität zu Kiel, Department of Computer Science, December 2007b. `http://rtsys.informatik.uni-kiel.de/~biblio/downloads/theses/mabo-dt.pdf`

  This thesis contains the implementation of a compiler from Esterel to *KEP* assembler (strl2kasm).

- Malte Tiedje. Beschreibung des Kiel Esterel Prozessors in Esterel. Diploma thesis, Christian-Albrechts-Universität zu Kiel, Department of Computer Science, January 2008. `http://rtsys.informatik.uni-kiel.de/~biblio/downloads/theses/mti-dt.pdf`

  In this thesis, the *KEP* was reimplemented using Esterel as a hardware description language.

- Falk Starke. Executing Safe State Machines with the Kiel Esterel Processor. Diploma thesis, Christian-Albrechts-Universität zu Kiel, January 2009. `http://rtsys.informatik.uni-kiel.de/~biblio/downloads/theses/fast-dt.pdf`

  This thesis explains the direct generation of *KEP* assembler from SyncCharts.

The processor description of the *KEP* and the *KLP* in Esterel, and the related compilers are published open source on: `www.informatik.uni-kiel.de/rtsys/krep`.

1.4. Outline

In the next chapter, we will consider related work and give an overview over synchronous languages in Chapter 3. In Chapter 4 we introduce a formal semantics for the *KEP* assembler and in detail the compilation into *KEP* assembler. In Chapter 5 we introduce the Kiel Lustre Processor (*KLP*) and the compilation from Lustre and Scade. Chapter 6 gives different approaches for WCRT analysis, and Chapter 7 presents the evaluation of the processors using KIELER.

2. Related Work

Contents

There are three different research areas related to the work presented in this thesis:

1. Reactive processors, *i. e.*, processors specially designed to execute synchronous languages for reactive systems. In a broader sense, this includes processors that are specially designed for the execution of embedded real-time systems.

2. Since we are executing Esterel and Lustre, there are strong relationships to other means of the execution of these languages, in particular on the compilation into software, but also to the distribution of synchronous programs.

3. Worst Case Execution Time (*WCET*) analysis: The current research on WCET analysis focus on the detailed but efficient modeling of modern general purpose processors, while we are interested in the WCRT analysis on reactive processors, which have a simpler timing behavior.

2.1. Processor Design

This work is based on the *KEP* by Li [2007], a reactive processor with an Instruction Set Architecture (*ISA*) closely related to Esterel. It supports concurrency by hardware threads with a priority based scheduling. Preemption is implemented by hardware watchers, which will sense the code ranges and suspend or abort the execution when their trigger signal is present. We will give more details on the *KEP* in Section 4.

Other reactive processors for Esterel are the StarPro by Yuan et al. [2008], which also supports concurrency by hardware threads. But in contrast to the *KEP*, abortion is handled by explicit software checks, which allows a simpler hardware design. The

Emperor [Yoong et al., 2006] is a reactive processor that supports concurrency by multiple cores. However, the synchronous semantics of Esterel makes it hard to really use this parallelism at runtime, because concurrent threads in realistic Esterel programs usually have strong data-dependencies and they can exchange signals back and forth instantaneously within one tick.

The PReT [Lickly et al., 2008] approach targets the development of a general purpose processor, which can execute arbitrary C code, while still allowing exact timing analysis. To achieve this, processor parts that might have unpredictable timings are replaced by better analyzable parts, *e. g.*, caches by scratch-pad memories and memory access by a memory wheel. However, to fully utilize its features, the programmer needs to use low-level deadline instructions. The *KEP* and the *KLP* support a domain specific input language. This simplifies the processor, and allows to program in a high level programming language. A similar approach, to build a general purpose processor from scratch, with special emphasis on the timing predictability, is taken by the Predator project[1].

The key idea of reactive processing is to take a language to describe reactive systems which is already established, such as Esterel, and design a special purpose processor to execute these language. A similar approach is taken by the Java Optimized Processor (*JOP*) [Schoeberl, 2008], a processor that is designed for time-predictable execution of Java byte-code, in particular for Real-Time Java. It implements a simple pipeline mechanism and cache system that do not introduce timing uncertainties. Since the JOP uses java-Byte code as an input language, hence they have no particular support for reactive control flow, in particular for deterministic concurrency and abortion.

The *KLP* is also related to general dataflow processors, like the Manchester Machine [Gurd et al., 1985], which allow the parallel execution of programs on multiple function units. Here, available data will trigger the execution of instructions that depend on it. This aims for simple parallel execution to reduce the average execution time. Compared to this, the parallelism on the *KLP* is more coarse grained, since only data that have reached their final value for the current instant can trigger further executions.

The current trend in the area of reactive processing seems to be more light-weight software solutions. PRET-C [Andalam et al., 2009] is a small extension of Esterel to allow the expression of reactive control-flow, similar to Reactive-C [Frederic Boussinot, 1991]. It was directly inspired by the instruction set of reactive processors, and designed to allow easy timing predictability. To achieve timing predictability, the processor only needs to be extended by a small scheduler.

A similar approach is taken by SyncCharts in C [von Hanxleden, 2009], which is an instruction set that allows to express SyncCharts directly in C. Here, no additional hardware is needed, the complete instruction set can be efficiently implemented as C macros. While this approach does not give timing predictability, it is a convenient way to express the constructs of SyncCharts, in particular the deterministic concurrency, directly in C. Similar to PRET-C, the instruction set could be implemented by a processor extension in hardware, in order to allow predictable timings like in PRET-C.

[1] `www.predator-project.eu`

2.2. Execution of Synchronous Programs

2.2.1. Compiling Esterel

In the past, various techniques have been developed to synthesize Esterel into software; see Potop-Butucaru et al. [2007] for an overview. There are three main compilation approaches for Esterel: compilation into automata, compilation into synchronous circuits, which are simulated at run-time, and simulation-based approaches, which try to emulate the control logic of the original Esterel program directly, and generally achieve compact and yet fairly efficient code. These approaches first translate an Esterel program into some specific graph formalism that represents computations and dependencies, and then generate code that schedules computations accordingly. The EC/Synopsys compiler first constructs a *concurrent control flow graph* (CCFG), which it then sequentializes [Edwards, 2002]. Threads are statically interleaved according to signal dependencies, with the potential drawback of superfluous context switches; furthermore, code sections may be duplicated if they are reachable from different control points. The SAXO-RT compiler [Closse et al., 2002] divides the Esterel program into basic blocks, which schedule each other within the current and subsequent logical tick. An advantage relative to the Synopsis compiler is that the SAXO-RT compiler does not perform unnecessary context switches and largely avoids code duplications; however, the scheduler it employs has an overhead proportional to the total number of basic blocks present in the program. The grc2c compiler [Potop-Butucaru and de Simone, 2004] is based on the *graph code* (GRC) format, which preserves the state-structure of the given program and uses static analysis techniques to determine redundancies in the activation patterns. A variant of the GRC has also been used in the *Columbia Esterel Compiler* (CEC) [Edwards and Zeng, 2007], which again follows SAXO-RT's approach of dividing the Esterel program into atomically executed basic blocks. However, their scheduler does not traverse a score board that keeps track of all basic blocks, but instead uses a compact encoding based on linked lists, which has an overhead proportional to just the number of blocks actually executed.

In summary, there is currently not a single Esterel compiler that produces the best code on all benchmarks, and there is certainly still room for improvements. For example, the simulation-based approaches presented so far restrict themselves to interleaved single-pass thread execution, which in the case of repeated computations ("schizophrenia" [Berry, 1999]) requires code replications. We differ from these approaches in that we do not want to compile Esterel to C, but instead want to map it to a concurrent reactive processing ISA.

The *multi-processing* approach is represented by the EMPEROR [Yoong et al., 2006], which uses a cyclic executive to implement concurrency, and allows the arbitrary mapping of threads onto processing nodes. This approach has the potential for execution speed-ups relative to single-processor implementations. However, their execution model potentially requires to replicate parts of the control logic at each processor. The EMPEROR Esterel Compiler 2 (EEC2) [Yoong et al., 2006] is based on a variant of the GRC, and appears to be competitive even for sequential executions on a traditional processor.

However, their synchronization mechanism, which is based on a three-valued signal logic, does not seem able to take compile-time scheduling knowledge into account, and instead repeatedly cycles through all threads until all signal values have been determined.

The *multi-threading* approach has been introduced by the Kiel Esterel Processor family and has subsequently been adapted by the STARPro architecture [Yuan et al., 2008], a successor of the EMPEROR. In some sense, compilation onto KEP assembler is relatively simple, due to the similarities between the Esterel and the *KEP* Assembler. However, we do have to compute priorities for the scheduling mechanism of the *KEP*, and cannot hard-code the scheduling-mechanism into the generated code directly. Incidentally, it is this dynamic, hardware-supported scheduling that contributes to the efficiency of the reactive processing approach.

It has also been proposed to run Esterel programs on a virtual machine (BAL [Plummer et al., 2006]), which allows a very compact byte code representation. In a way, this execution platform can be considered as an intermediate form between traditional software synthesis and reactive processing; it is software running on traditional processors, but uses a more abstract instruction set. The proposal by Plummer et al. [2006] also uses a multi-threaded concurrency model, as in the KEP platform considered here. However, they do not assume the existence of a run-time scheduler, but instead hand control explicitly over between threads. Thus their scheduling problem is related to the scheduling on the *KEP*, but does not involve the need to compute priorities as we have to do here. Instead they have to insert explicit points for context switches. The main difference in both approaches is that the *KEP* only switches to active threads, while the BAL switches to statically defined control points. One could, however, envision a virtual machine that has an ISA that adopts the multi-threading model of the *KEP*, and for which the approach presented here could be applied. A straightforward, albeit inefficient VM is the *KEP* simulator.

The compact representation of a synchronous language was also a motivation for SyncCharts in C [von Hanxleden, 2009], and via an translation from Esterel to SyncCharts [Prochnow et al., 2006], this approach could also be used to generate C code from Esterel. Since the operators defined in SyncCharts in C are similar to the instructions of the *KEP*, the strl2kasm compiler could also be used to generate code for SyncCharts in C. In particular, the model of concurrent threads with a priority based scheduler is the same. However, since it has no equivalent to the watchers of the *KEP*, abortions need to be implemented by explicit checks.

2.2.2. Compiling SyncCharts

While Statecharts are an appealing language to describe reactive behaviors, the generation of efficient code is not trivial. Three different methods of compiling Statecharts can be distinguished: compilation into an object oriented language using the state pattern [Ali and Tanaka, 2000], dynamic simulation [Wasowski, 2003], and flattening into finite state machines. Since flattening can suffer from state explosion, often a combination of flattening and dynamic simulation is used. As Statecharts exist in various different flavors, the optimal code generation scheme depends on the considered Statechart vari-

ant and its semantics. We focus on SyncCharts, a synchronous Statechart variant with a formal semantics, and on the compilation for the *KEP*. Since the *KEP* directly supports concurrency and hierarchy (by means of preemption) the compilation from SyncCharts to *KEP* assembler differs from standard Statechart compilation. However, it can be seen as a simulation based approach as used for general purpose processors, were the simulator is implemented in hardware.

A translation from SyncCharts to Esterel was proposed by André [2003] together with the initial definition of SyncCharts and their semantics. This transformation, with additional unpublished optimizations, is implemented in *Esterel Studio*[2]. Another compilation from SyncCharts to Esterel was developed byYoong et al. [2009] in the context of function blocks, however the considered SyncCharts are flat.

The translation from SyncCharts to *KEP* assembler is most closely related to the extension of Esterel with GOTO by Tardieu and Edwards [2007]. Since they extend the language, they have to consider all possible usages of GOTO, *e. g.*, jumping from one thread into another. The translation from SyncCharts could be directly used to generate efficient extended Esterel (including GOTO), since the structure of the SyncChart will always generate valid GOTOs. The Esterel with GOTO could then be translated into *KEP*-assembler. Instead, the translation generates *KEP* assembler, which already has a GOTO statement, directly from SyncCharts, without generating Esterel. One practical reason is that there is no publicly available compiler for Esterel with GOTO; another is that the *KEP* assembler is close enough to Esterel that this intermediate step would not make much difference in code generation.

The issue of extracting complex signal expressions into simple condition triggers is related to extracting such expressions into external hardware in hardware/software co-design by Gädtke et al. [2007]. In co-design, the motivation is to accelerate computation, and the challenge is to cleanly extract such expressions into the hw/sw interface. For the translation from SyncCharts to *KEP* assembler, the motivation is to provide triggers for the abortion watchers, and the challenge is to ensure that the trigger signals are computed for as long as necessary without blocking progress.

2.2.3. Compiling Lustre/Scade

Compared to the compilation of Esterel, the compilation of Lustre is trivial, in part due to the restriction to acyclic programs. Therefore, there has not been as much scientific work as for the compilation of Esterel. Lustre code can be compiled into automata [Bouajjani et al., 1992], this is primarily done for verification. More efficient is the generation of so called single-loop programs [Halbwachs, 2005]. Here the compiler orders the equation statically by their dependencies and simply calls them in this order. The main difficulty here is to extract common sub-expressions, to generate efficient code. The approach handles Lustre clocks, which define when an expression is evaluated, just like conditionals. Recently, [Biernacki et al., 2008] introduced a more efficient way to compile Lustre, which uses the clocks to generate minimal models. Since we are

[2] `www.esterel-technologies.com`

compiling for specific hardware, we map Lustre clocks directly to the clocks on the *KLP*.

2.2.4. Distributed Executions

Synchronous languages rely on a global clock, therefore a distributed execution of synchronous programs is not trivial. Girault [2005] gives an overview of different approaches. In the proposed ocrep and screp tools [Caspi et al., 1999], the generated code is replicated to all distributed components, and code parts that are not needed on a specific component are removed afterwards. While this approach is feasible, it is not efficient for longer communication latencies. A more efficient solution is the creation of globally asynchronous, locally synchronous systems (GALS) [Balarin et al., 1999]. The theoretical difficulty is the reaction to absent signals. To support this, the notion of endo-and isochrony was introduced by Benveniste et al. [1997].

Esterel was augmented to allow multiple clocks by Berry and Sentovich [2001]. However, these rely still on a common global clock, derived clocks can be used to down-sample modules. This is not used for the generation of distributed software, but for clock-gating in hardware.

2.3. Worst Case Execution/Reaction Time Analysis

Regarding timing analysis, there exist numerous approaches to classical WCET analysis. For surveys see, *e. g.*, Puschner and Burns [2000] or Wilhelm et al. [2008]. These approaches usually consider (subsets) of general purpose languages, such as C, and take information on the processor designs and caches into account. It has long been established that to perform an exact WCET analysis with traditional programming languages on traditional processors is difficult, and in general not even possible for Turing-complete languages. Therefore WCET analysis typically impose fairly strong restrictions on the analyzed code, such as a-priori known upper bounds on loop iteration counts, and even then control flow analysis is often overly conservative [Malik et al., 1997, Burns and Edgar, 2000]. Furthermore, even for a linear sequence of instructions, typical modern architectures make it difficult to predict how much time exactly the execution of these instructions consumes, due to pipelining, out-of-order execution, argument-dependent execution times (*e. g.*, particularly fast multiply-by-zero), and caching of instructions and/or data [Berg et al., 2004]. Finally, if external interrupts are possible or if an operating system is used, it becomes even more difficult to predict how long it really takes for an embedded system to react to its environment. Despite the advances already made in the field of WCET analysis, it appears that most practitioners today still resort to extensive testing plus adding a safety margin to validate timing characteristics. To summarize, performing conservative yet tight WCET analysis appears by no means trivial and is still an active research area.

The WCRT of a synchronous program is the maximal time between sampling the inputs and producing the outputs. Whether WCRT can be formulated as a classical

WCET problem or not depends on the implementation approach. If the implementation is based on sequentialization such that there exist two dedicated points of control at the beginning and the end of each reaction, respectively, then WCRT can be formulated as WCET problem; this is the case, for example, if one "step function" without an internal state is synthesized, which is called during each reaction. If, however, the implementation builds on a concurrent model of execution, where each thread maintains its own state of control across reactions, then WCRT requires not only determining the maximal length of pre-defined instruction sequences, as in WCET, but one also has to analyze the possible control point pairs that delimit these sequences. Thus, WCRT is more elementary than WCET in the sense that it considers single reactions, instead of whole programs, and at the same time WCRT is more general than WCET in that it is not limited to pre-defined control boundaries.

One step to make the timing analysis of reactive applications more feasible is to choose a programming language that provides direct, predictable support for reactive control flow patterns. We argue that synchronous languages, such as Esterel, are generally very suitable candidates for this, even though there has been little systematic treatment of this aspect of synchronous languages so far. One argument is that synchronous languages naturally provide a timing granularity at the application level, the *logical ticks* that correspond to system reactions, and impose clear restriction onto what programs may do within these ticks. For example, Esterel has the rule that there cannot be *instantaneous loops*: within a loop body, each statically feasible path must contain at least one tick-delimiting instruction, and the compiler must be able to verify this. Another argument is that synchronous languages directly express reactive control flow, including concurrency, thus lowering the need for an operating system with unpredictable timing.

Ringler [2000] considers the WCET analysis of C code generated from Esterel. This approach considers the generation of circuit code [Berry, 1999], which generates a synchronous circuit from Esterel, which is than simulated in C.

An approach to arrive at more accurate values is recently proposed by Ju et al. [2008]. Here, Esterel programs are first mapped to C using the CEC compiler and then an integer linear program formulation is developed to eliminate redundant paths in the code, thus yielding more accurate results. However, the objective of this work was not to obtain the most tight value possible.

Li et al. [2005] compute a WCRT of sequential Esterel programs directly on the source code for the *KEP*. However, they did not address concurrency, and their source-level approach could not consider compiler optimizations. We perform the analysis on an intermediate level after the compilation, as a last step before the generation of assembler code. This also allows a finer analysis and decreases the time needed for the analysis.

One important problem that must be solved when performing WCRT analysis for Esterel is to determine whether a code segment is reachable instantaneously, or delayed, or both. This is related to the well-studied property of *surface* and *depth* of an Esterel program, *i. e.*, to determine whether a statement is instantaneously reachable or not, which is also important for schizophrenic Esterel programs [Berry, 1999]. This was addressed in detail by Tardieu and de Simone [2003]. They also point out that an exact

analysis of instantaneous reachability has NP complexity.

2.3.1. Interface Algebra

Interface algebras are an accepted method of modularizing embedded systems programming and specifically synchronous programming. Most interface models in synchronous programming are restricted to causality issues, *i. e.*, dependency analysis without quantitative time. The modules of André et al. [1997] do not permit instantaneous interaction. Such a model is not suitable for WCRT. Hainque et al. [1999] use a topological abstraction of the underlying circuit graphs (or syntactic structure of Boolean equations) to derive a fairly rigid component dependency model with the effect that multi-threaded execution cannot be modeled compositionally. The interface model also does not cover data dependencies and thus cannot deal with dynamic schedules and does not support WCRT, either.

The causality interfaces of Lee et al. [2005] are more flexible. These are functions associating with every pair of input and output ports an element of a *dependency domain*, which expresses if and how an output depends on some input. Causality analysis is then performed by multiplication on the global system matrix. Using an appropriate dioid structure D, one can perform the analyzes of Hainque et al. [1999] as well as restricted forms of WCRT. However, Lee's interfaces cannot express the difference between an output depending on the joint presence of several values as opposed to depending on each input individually. Thus they do not support full AND- and OR-type synchronization dependencies and hence cannot represent neither multi-threading nor multi-processing.

Similar restrictions apply to recent work [Wandeler and Thiele, 2005, Henzinger and Matic, 2006] combining network calculus [Baccelli et al., 1992, Boudec and Thiran, 2001] with real-time interfaces. These works are concerned with the compositional modeling of regular execution patterns rather than stabilization processes inside each execution cycle of a synchronous program. Existing interface theories [Lee et al., 2005, Wandeler and Thiele, 2005, Henzinger and Matic, 2006], which aim at the verification of resource constraints for real-time scheduling, handle timing properties such as task execution latency, arrival rates, resource utilization, throughput, accumulated cost of context switches, and so on. However, the dependency on data and control flow is largely abstracted. For instance, since the task sequences of Henzinger and Matic [2006] are independent of each other, their interfaces do not model concurrent forking and joining of threads. The causality expressible there is even more restricted than that by Lee et al. [2005] in that it permits only one-to-one associations of inputs with outputs. The interfaces of Wandeler and Thiele [2005] for modular performance analysis in real-time calculus are like those of Henzinger and Matic [2006] but without sequential composition of tasks and thus do not model control flow as we do here.

In so far as WCRT analysis aims to obtain exact bounds on the duration of stabilization processes with synchronous feedback, it is related to the timing analysis of combinational circuits (see, e.g., [Benkoski and Strojwas, 1989, Devadas et al., 1991, Silva and Sakallah, 1993, Lam and Brayton, 1994]) which is known to be NP-complete. Although WCRT analysis for single or multi-threaded synchronous processing can mostly be per-

formed in max-plus as opposed to min-max-plus algebra, the inherent data dependency still makes it computationally intractable without sophisticated heuristics. The work presented here fits into a general and expressive interface theory [Mendler, 2000] for stabilization processes which has been developed to provide a semantic foundation for such heuristics. It supports modularization and hierarchical abstraction and systematizes earlier work on combinational timing analysis.

2.3.2. Model Checking

The use of model checking for the analysis of real-time systems is not new. Metzner [2004] illustrates the effectiveness of using model checking for WCET analysis using the notion of a basis block automaton to represent a program. Similarly, in [Gu, 2005] an approach for computing the best and worst case response time of tasks is presented using model checking. Logothetis and Schneider [2003], Logothetis et al. [2003] have employed model checking to perform a precise WCET analysis for the synchronous language Quartz, which is closely related to Esterel. However, their problem formulation was different from the WCRT analysis problem we are addressing. They were interested in computing the number of ticks required to perform a certain computation, such as a primality test, which we would actually consider to be a transformational system rather than a reactive system [Harel and Pnueli, 1985]. We here instead are interested in how long it may take to compute a single tick, which can be considered an orthogonal issue.

3. Synchronous Languages

Contents

Synchronous languages [Benveniste et al., 2003, Halbwachs, 1998] are a family of languages to describe the behavior of reactive systems. These languages come in different flavors: textual or graphical, imperative or declarative. They all share the common *synchrony hypothesis*, which states that outputs occur simultaneously with their inputs, hence the computation itself does not take time. While this view greatly simplifies the semantics of these languages, it seems to be unrealistic for a real implementation. However, from a implementation point of view, the synchronous hypothesis can be read as: "new events from the environment will not occur before the computation of the current events has finished." From a developers point of view, the logical and the temporal behavior are separated. These languages also separate the description of the behavior and the description of data-handling. Most of the languages do not support complex data computations, but allow to call external functions in a host language to perform computations or access complex data-types.

The key issues of synchronous languages are the specialties of control flow in reactive systems: *concurrency* and *preemption*. Both are deterministic, in contrast to other, asynchronous forms of parallelism. The behavior is divided into discrete *ticks*, also called *instants*. Inside each tick no time elapses, hence all concurrent threads see coherent data.

Synchronous languages are high-level modelling languages. From the models, both hardware and software can be synthesized. Synchronous languages have a rigid formal semantics, this allows the formal verification of models.

In the following we will take a closer look at some synchronous languages: the declarative language Lustre and the imperative Esterel, as well as their graphical counterparts, Scade and SyncCharts. To explain the different languages, we will use the simple gate controller that is shown in Figure 3.1 as a running example. The gate shall have the following behavior: per default, the gate is closed, until the user enters a valid ID. Then

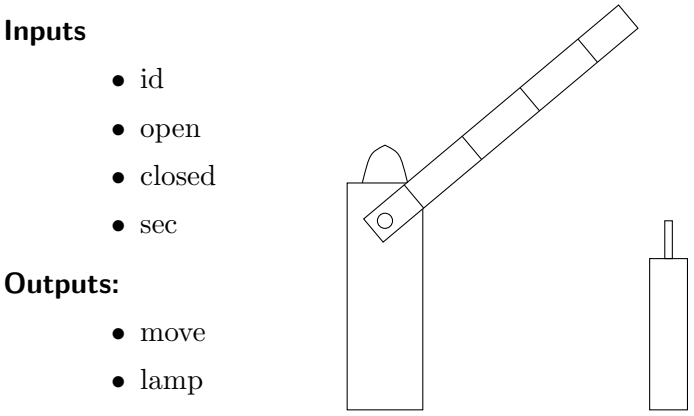


Figure 3.1.: A simple gate, which is used to demonstrate different synchronous languages.

it stays open for 5 seconds. The gate should open whenever an valid ID is entered, even if it is currently closing. As a rough abstraction for the validation of the ID, we consider an id valid if it is a multiple of five. A lamp shall flash with one second frequency whenever the gate is moving. So the inputs are the ID, signals whether the gate is completely opened or closed, and a time signal which is set to true once every second. The outputs are a move signal to actually move the gate and a lamp.

The system consists of three components, 1) check whether the entered ID is valid 2) determine whether we shall open or close the gate and 3) give actual move instructions to the environment, which shall prevent the gate from rapid movements. The first and the third step are data-flow dominated, the second step is pure control flow.

3.1. Lustre

Lustre [Halbwachs and Raymond, 2001] is a synchronous dataflow language. Synchronous dataflow [Lee and Messerschmitt, 1987] is a restriction of the general Kahn dataflow model [Kahn, 1974]. In a Kahn process network, each process communicates with other processes via unbounded buffers. It can produce and consume any amount of data, the only restriction is that reads on an empty buffer are always blocking and a process has no possibility to check whether input is available. While this is a very general model for asynchronous concurrency, these few limitations lead to deterministic behavior, which is independent from the scheduling of the different processes. However, it is hard to determine an upper bound for the buffer-size that is needed for a given scheduling. It is even NP complete to check whether the buffer size is bounded at all [Park et al., 1998]. Consequently, it is hard to come up with a scheduling that has minimal resource usage.

To overcome these issues, synchronous dataflow was introduced. Here, each process will produce and consume a fixed number of data whenever it is executed, independent from the program state or read inputs. This makes the computation of an optimal

```
1  node COUNT(C: bool) returns (X: int);
2  var
3    clock : bool;
4  let
5    clock= true → EDGE(C);
6    X = current((0→ (pre X) + 1) when clock);
7  tel
8
9  node EDGE(I: bool) returns (O: bool);
10 let
11  O = I → I and not pre(I);
12 tel
```

Tick	0	1	2	3	...
C	0	0	1	1	...
clock	1	0	1	0	...
pre X	⊥	0	0	1	...
0→(pre X) +1	0	1	1	2	...
0 →(pre X)+1) when clock	0		1		...
X	0	0	1	1	...

Figure 3.2.: A simple Lustre program and an execution trace

scheduling easy, where the scheduling can either minimize the used buffer space or the program size. While synchronous dataflow is a general concept without a concrete syntax, Lustre is a actual modelling language. It can be seen as a Kahn process network, where each buffer has a size of one.

The basic blocks from which Lustre programs are constructed are called *nodes*. A node consists of a set of *inputs* and *outputs* and a set of *concurrent equations*, which are executed synchronously.

Lustre programs operate on infinite streams of data. Clocks are used to define when a value is computed. In the notion of Lustre, a clock is simply a boolean stream. Beside usual arithmetics, Lustre defines the following *clock operators*:

pre e This accesses the previous defined value of the expression e. This value is undefined (nil) in the first instant the pre is executed. In a correct Lustre program, this value is never accessed. Therefore, pre should only appear on the right hand side of an initialization (see below.).

e1 when e2 Down-sample an expression: the expression e1 is only computed when the boolean expression e2 evaluates to true. In this case, the whole expression evaluates to e1, otherwise it is undefined.

current e Up-sample an expression: the value is the last defined value e.

e1 →e2 Initialize: The result is the value of e1 in the first instant, and the value of e2 in all following instants.

The example in Figure 3.2 shows the behavior of the different clock operators. The node COUNT counts the rising edges of the input signal C. Setting the initial value of clock to true assures that X is correctly initialized as well.

For a valid Lustre program, only expressions that run on the same clock may be combined, since otherwise, it would not be possible to bound the memory usage. Consider the program in Figure 3.3. The program will produce the following values, where X^i is the i-th value of X.

```
1  I = 1 → pre(I) +1;
2  C = true → not (pre C);
3  X = (I when C) + I
```

Tick	0	1	2	3	4
I	1	2	3	4	5
C	T	F	T	F	T
I when C	1		3		5
X	$I^0 + I^0 = 2$	$I^1 + I^2 = 5$	$I^2 + I^4 = 8$	$I^3 + I^6 = 11$	$I^4 + I^8 = 14$

Figure 3.3.: Invalid Lustre program with inconsistent clocks and its execution trace

To compute the i-th value of X, we need the i-th and the $2i$-th value of I. Since we have to store all inputs that we will use later somewhere, i values must be stored for the i-th tick, hence the memory consumption of this program cannot be bounded. To solve this problem Lustre only allows the combination of values that have the same clock. Unfortunately, it is in general not decidable whether two clocks are dynamically the same. Therefore, Lustre only allows clock combinations that are *syntactically* equivalent, *e. g.*, only renamings are considered, but no boolean operations, which are undecidable to be equivalent in general. Figure 3.4 shows an example for *syntactical and semantical clock consistency*. Here, the fact that C1 and C1 and C1 are equivalent are not realized by the compiler.

```
1 node valid(X: int; C1:bool;) returns (O:int);
2 var C2: bool;
3 let
4  C2=C1;
5  O = current ((X when C1) + (1 when C2));
6 tel
```

(a) C1 and C2 are syntactically equivalent. This program is accepted by the Lustre compiler.

```
1 node invalid(X: int; C1:bool;) returns (O:int);
2 var C2: bool;
3 let
4  C2=C1 and C1;
5  O = current ((X when C1) + (1 when C2));
6 tel
```

(b) C1 and C2 are semantically equivalent, but not syntactically. This program is rejected by the Lustre compiler.

Figure 3.4.: Clock equivalence in Lustre programs

The definition of Lustre was clearly influenced by control engineering. Thus one may see Lustre as extending circuits with wires, which can hold not only boolean values but numbers. Since the design of reactive systems is often done by engineers who have a background in control theory, rather than in computer-science, such a dataflow formalism is a good means to describe these systems.

3.1.1. Clock operators

Lustre is a simple language, nevertheless the behavior can be quite tricky. In particular, the handling of clocks can be confusing. Consider the example in Figure 3.5, where we extend the COUNT example by an additional signal X2. On a first glance, X and X2 should behave the same, we only moved the constant initialization outside of the clock operator. But, as the trace shows, this does not properly initialize the expression. So the first time that the when is executed, its body is not defined, due to the access of an undefined pre. In the following instant, the current propagates this undefined value. In

contrast, the definition of X correctly initializes the first execution of the when.

```
1  node COUNT(C:bool) returns (X:int; X2:int);
2  var
3    clock : bool;
4  let
5    clock= true → EDGE(C);
6    X = current((0 → (pre X) + 1) when clock);
7    X2 = 0→current(((pre X2) + 1) when clock);
8  tel
9
10 node EDGE(I:bool) returns (O:bool);
11 let
12  O = I → I and not pre(I);
13 tel
```

C	0	0	1	1	...
clock	1	0	1	0	...
X	0	0	1	1	...
X2	0	⊥	⊥		...
0→(pre(X)+1) when clock	0		1	...	
(pre(X2)+1) when clock	⊥		⊥	...	

Figure 3.5.: A simple Lustre program and an execution trace

3.1.2. Gate Example

Figure 3.6 shows the implementation of the gate example in Lustre. It consists of three nodes: check to compute whether a valid request is coming in, ctrl to implement the control logic that determines whether the gate shall be moved up or down, and smooth to prevent the gate from rapid accelerations. These nodes are connected in the main node gate. In the check node we first compute whether a new id is entered. Only if this is the case, the validation of the id is triggered, *i. e.*, ok is computed. A valid request is entered, if a new id is entered and the validation sets ok to true.

The dataflow equations of Lustre can be naturally represented by dataflow-diagrams. Figure 3.7 shows the same implementation of the gate in a graphical notation, implemented in the KIELER.

3.1.3. Compilation

The first approaches to compile Lustre generated automata code [Halbwachs et al., 1991b]. While this is a very fast implementation and it can be shown that the generated automaton is minimal, this approach turned out to be unfeasible for realistic examples, due to the possible exponential growth of the code size.

Current Lustre compilers implement the program by sequentializing the equations according to the data-dependencies. The when operator is implemented by a conditional, and in each tick, it needs to be checked whether the equation is initialized or not. An important part of the compiler optimization is to extract common subexpressions from different equations and to reduce the amount of memory that is used, *e. g.*, often reordering the equations makes it unnecessary to store the previous value of an equation.

One of the open questions in compiling synchronous languages is modular code generation. Compiling all modules independently is not possible in general, because of instantaneous feed-back loops. The standard Lustre compiler inlines all nodes, and then

```
1  // main node
2  node gate(id: int;
3            open, closed, sec: bool)
4  returns (lamp: bool; move:int);
5  var
6    request, up, down: bool;
7  let
8    request = check(id);
9    (up, down) = ctrl(open, closed, sec, request);
10   (move, lamp) = smooth(up, down, sec);
11
12 tel
13
14 // compute if gate should be opened or closed
15 node ctrl(open, closed, sec, request: bool)
16 returns (up, down: bool);
17 var
18   count : int;
19 let
20   count = 0→ if open and not pre(open)
21                then 5
22                else if pre(count)=0
23                     then 0
24                     else if sec
25                          then pre(count)−1
26                          else pre(count);
27   up = false
28        → request or (not open and pre(up));
29   down = false → not closed and count=0;
30 tel
31
32 // check whetеher a new, valid id was entered
33 node check(id: int) returns(request: bool);
34   var ok, new_id: bool;
35 let
36    request = new_id and ok;
37    new_id = true → id <> pre(id);
38    ok = current ((5 ∗ (id/5) = id) when new_id);
39 tel
40
41 // compute actual actuator output
42 // from logical movements
43 node smooth(up, down, sec: bool)
44 returns (move: int; lamp: bool);
45 var
46   d: int;
47 let
48   d = pre(move)/4;
49   move = 0→ if up
50              then min(max(pre(move)+d+1,
51                           pre(move)−2∗d+1),
52                       100)
53              else if down
54                   then max(pre(move)+d−1,
55                            −100)
56                   else 0;
57   lamp = if move=0
58          then false
59          else current(not pre(lamp) when sec);
60 tel
```

Figure 3.6.: Lustre implementation of the gate example

compiles this monolithic program. Other compilers, such as the reluc compiler, compile all nodes individually, but reject programs which directly use the output of nodes as inputs without any pre operator in between.

3.2. Scade

A commercial variant of Lustre is Scade, which is implemented in the SCADE (Safety-Critical Application Development Environment) tool by Esterel Technologies[1]. A Scade version of the gate example is shown in Figure 3.8.

Beside giving a graphical notation, Scade extends Lustre by some additional features, such as convenient conversion between datatypes, and additional in-build functions. It also extends Lustre by SyncCharts, a synchronous StateChart variant [Colaço et al., 2005]. Data equations and automata can be mixed freely, *i. e.*, states may contain equations and the expression that computes an equation might itself contain an automaton.

[1] `www.esterel-technologies.com`

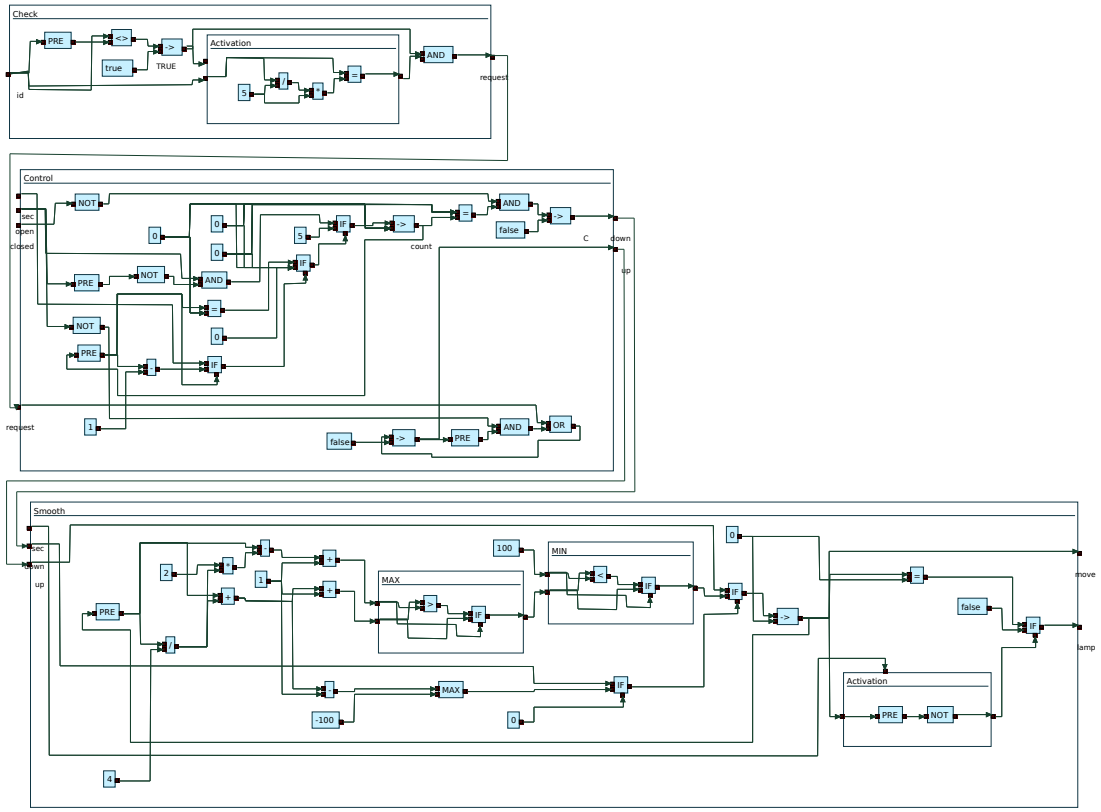


Figure 3.7.: Implementation of the gate with a graphical Lustre variant.

Scade supports three different kinds of transitions:

Weak-delayed abortions allow the execution of the source state in the instant the transition is triggered and activate the target state in the next instant.

Strong abortions immediately abort the execution of the source state when they are triggered and transfer control to the target state.

Synchronized transitions are triggered when the source state itself reaches a state that is flagged as final. The transition triggers can be arbitrary data-expressions or signals, *i. e.*, boolean variables.

Since the arbitrary mixing of clocks can lead to very complex code that is hard to understand, the usage of clocks in Scade is per default restricted. Clocks are replaced by *activation conditions*, which execute a given node only when a boolean value is true, as the CheckID and Toggle nodes in Figure 3.8. Whenever the boolean condition is not true, the output will always be defined, as in Lustre, but the last defined value will be replicated. The activation condition also requires an initial value, hence its output is

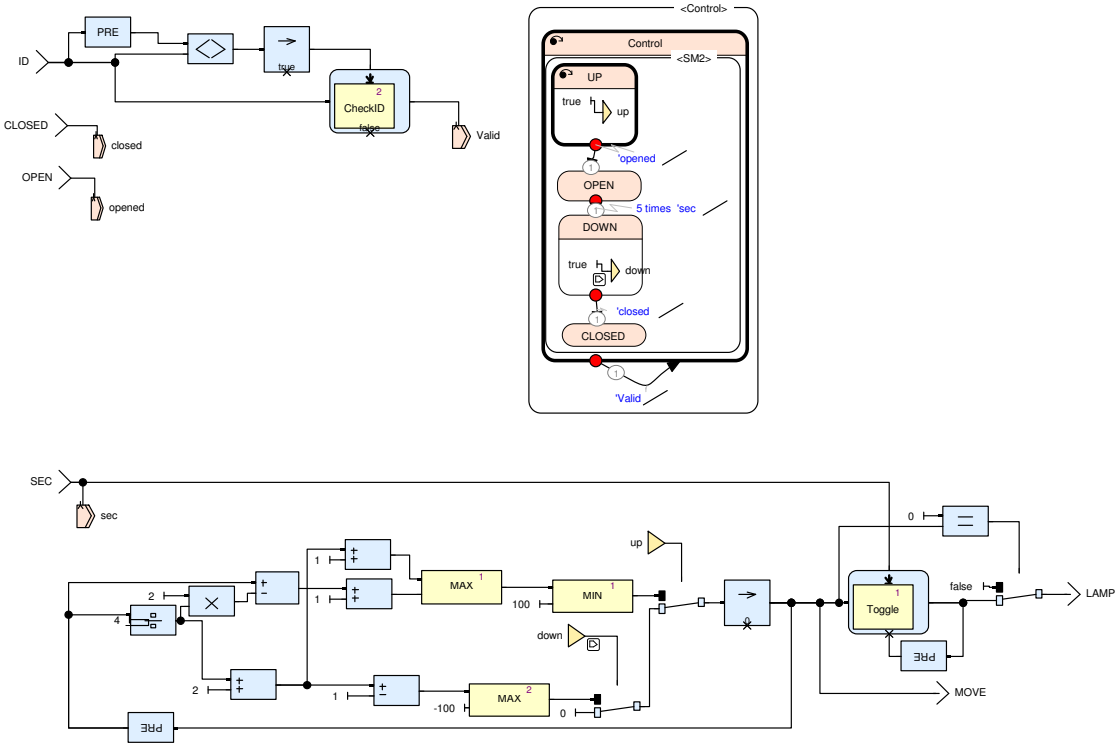


Figure 3.8.: Scade implementation of the gate example

always defined. The usage of the plain current and when is still possible in Scade, but deprecated. Also the Lustre init operator is usually replaced by the fby (followed by) operator, where a fby b is equivalent to a → pre(b).

While Lustre supports boolean clocks, which are either present or absent, clocks in Scade can be arbitrary enumeration, and the when operator is extended to e when (C=V): the flow e is evaluated when the clock C has value V. Enumerated clocks can be directly used to implement SyncCharts, where the active state is implemented as a clock, hence the flow inside a state is only evaluated when the state is active. Another difference between Lustre and Scade is the handling of arrays. In Lustre arrays can be accessed by static indices and slices, while in Scade more flexible map and fold operators are defined.

Parts of the Scade language are no also implement in the new Lustre v6 implementation, in particular enumerated clocks and the handling of arrays by map and fold.

3.3. Esterel

Esterel [Berry and Cosserat, 1984, Potop-Butucaru et al., 2007] is an imperative synchronous language. Esterel programs communicate with the environment and internally via signals, which are either present or absent during one instant. Signals are set present

by the emit statement and tested with the present test. Local signals can be declared using the signal statement. Signals are absent per default: a signal is only present in a tick if it is emitted in this tick. Esterel statements can be either combined in sequence (;) or in parallel (||). The loop statement simply restarts its body when it terminates. All Esterel statements are considered instantaneous, except for the pause statement, which pauses for one instant. The suspend statement suspends its body when a trigger signal is present. Exception handling is done via named exceptions, called traps. The trap statement declares the scope of an exception. When the exception is raised with an exit statement, the control is transferred to the end of this trap scope. If multiple, different exceptions are raised in the same tick, the trap with the outermost scope is taken.

From this small set of *kernel statements* derived statements are declared. This includes simple statements like halt=loop pause, which stops forever, but also the abort and weak abort statements, which terminate their bodies when the trigger signal is present. Weak abortion permits the execution of its body in the instant the trigger signal becomes active, strong abortion does not. Both kinds of abortions can be either immediate or delayed. The immediate version already senses for the trigger signal in the instant its body is entered, while the delayed version ignores it during the first instant in which the abort body is started.

Beside the pure status, a signal can also contain an additional value. This value is persistent over ticks, if the signal is not emitted. If a valued signal is emitted multiple times within a tick, a commutative and associative function must be given to combine the signals. This ensures that the signal value is unique within a tick. Esterel also has a notion for variables, which can have different values within a tick. However, they cannot be read and written in parallel, hence all race conditions are syntactically excluded.

Figure 3.9 shows the implementation of the gate example in Esterel. The control logic can be expressed very naturally (Lines 14–22). However, the validation of the id (Lines 10–12) and in particular the smoothing the output (Lines 24–41) are purely sequential and do not use any special features of Esterel. In particular complex computations can often be better done in the host language, *i. e.*, C.

Esterel program can be compiled into the C code. The main difficulty is to sequentialize the concurrent parts of the Esterel program. This can either be the explicit concurrency expressed by ||, or the concurrency that evolves from the preemption, where both the execution of the preemption body statement and the checking of the trigger signal are semantically concurrent. One compilation approach is to generate automata, but this can lead to exponential growth of the code size, compared to the input program. Therefore, a more compact representation was developed, by generating a net-list from an Esterel program [Berry, 1992]. The generated C code simulates this net-list. While this has the advantage that the code size grows linear with the size of the input program, the execution time is rather slow. During the simulation, many irrelevant computations might be performed, which result will be ignored later. Current Esterel compiler like the Columbia Esterel Compiler (*CEC*) from Stephen Edwards [Edwards and Zeng, 2007] or the grc from Dumitru Potop-Butucaru [Potop-Butucaru et al., 2007] analyze the control flow graph to generate a fine grained static schedule for the sequentilization. However,

```
1  module GATE:
2  input   OPEN, CLOSED;
3  input   SEC;
4  input ID: integer;
5
6  output  LAMP;
7  output  MOVE: integer;
8
9  signal  valid, up, down in
10   every ID do
11     if 5*(?ID/5)=?ID then
12       emit valid
13     end if
14   end every
15 ||
16   every valid do
17     abort
18       sustain up
19     when OPEN;
20     await 5 SEC;
21     abort
22       sustain down
23     when CLOSED
24   end every
25 ||
26   loop
27     pause;
28     present up then
29       if pre(?MOVE)=100 then
30         emit MOVE(100);
31       else
32         emit MOVE(pre(?MOVE) + 1);
33       end if
34     else present down then
35       if pre(?MOVE)=100 then
36         emit MOVE(100);
37       else
38         emit MOVE(pre(?MOVE)  1);
39       end if
40     else emit MOVE(0);
41     end present
42     end present
43   end loop
44 end signal
45
46 end module
```

Figure 3.9.: Esterel implementation of the gate example

this requires that the program does not contain syntactical static cycles, while the compilation to net-list allows to compile all *constructive* programs, *i. e.*, programs where not cyclic dependencies occur at runtime. With the compilation into netlists, it is also possible to synthesize hardware from an Esterel program Berry [1991].

3.3.1. Esterel v7

Originally, Esterel was designed to program reactive systems in software, therefore the Esterel program is compiled into C code. In the last Esterel version, Esterel v7, the focus is put more into the design of hardware [Arditi et al., 2005]. For example, Support for data handling, in particular arrays and bit-vectors was added. To reduce the number of synthesized hardware registers, signals can be declared as valued and/or temp. A valued signal has no status, and the value of a temp signal is not preserved over ticks. Both the Kiel Esterel Processor and the Kiel Lustre Processor were implemented using Esterel v7.

Also, Lustre flows are incorporated into the language: an emit statement cannot only emit a signal, but evaluate a Lustre expression. To iterate over arrays, inside an emit statement it is possible to execute a for loop, which iterates over all signals in parallel or sequentially.

Esterel v7 also introduces multiple clocks [Berry and Sentovich, 2001]. This allows to suspend parts of the hardware, *e. g.*, to save energy. This feature is semantically based on *weak suspension*, first introduced in the synchronous language Quartz [Schneider,

2009], which evaluates all data computations inside its body, but preserves the active pause statements. To make reusing of existing components easier, it is now possible to inherit complete interfaces from other modules, and to combine input and/or output signals into ports.

For verification purpose, input and output assertions can be added to a module, similar to standard pre- and post-conditions. Furthermore, all interface signals can be inherited as outputs, in order to implement observers. Oracles can be used to get non-deterministic inputs for testing.

3.4. SyncCharts

SyncCharts (also called *Safe State Machines*) are a Statechart dialect with a synchronous semantics that strictly conforms to the Esterel semantics. An implementation of the gate example in SyncCharts (using KIELER) can be seen in Figure 3.10. As for the Esterel example, the actual gate controller on the bottom can be implemented very naturally, while the more dataflow oriented checking for the correct id (left) and the smoothening of the output (right) are rather cumbersome to express.

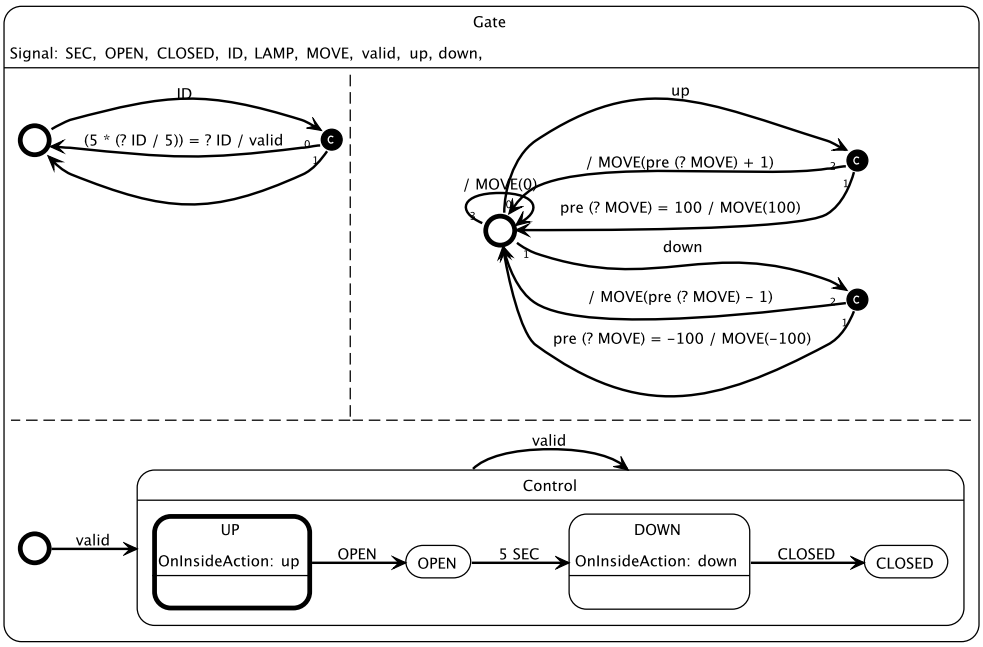


Figure 3.10.: Implementation of the gate as SyncChart

A procedural definition of the semantics of SyncCharts is given by André [2003]. The basic object in SyncCharts is a *reactive cell*, which is a state with its outgoing transitions. Reactive cells are combined to *state-transition graphs*, called *state regions* in other Statechart dialects. These are flat automata with exactly one *initial state*, which

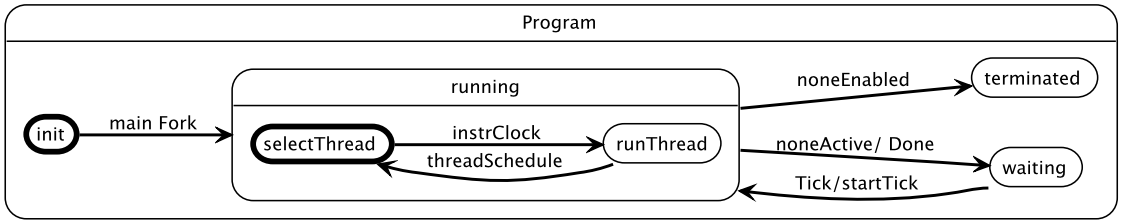
is indicated by a bold border. A *macro-state*, like the control state on the example, consists of one or more state-transition graphs. Additionally, SyncCharts can contain *textual macrostates*, which consist of plain Esterel code. States can also have *internal actions*: on entry, on exit and on inside. An on exit action is executed whenever the state is left, whether this is done via an outgoing transitions or a parent state of this state is left itself. SyncCharts inherit the concept of signals and valued signals from Esterel. Hence a transition trigger can consist of an event, which tests for presence and absence of values, and a conditional, which may compare numerical values. Characteristic for SyncCharts are the different forms of preemption, expressed by different state transition types. *Weak and strong abortion* transitions as well as *suspension* can be applied to macrostates. Strong abortions are indicated by a red dot on the arrow tail, like the transitions that restart the controller for each valid input in the example. Weak abortions are drawn as plain arrows. A variant of weak abortions are *weak-delayed abortions*, which only activate the target state in the next instant. They make sure that states are not transient, what can both simplify the compilation and the understanding of a SyncChart. A macrostate can either be left by an abortion, which has an explicit trigger, or by a *normal termination*, which is taken if the macrostate enters a terminal state. Normal terminations are indicated by a green triangle at the arrow tail. Analogously to Esterel, all transitions can either be immediate or delayed, where a delayed transition is only taken if the source state was already active at the start of an instant. In contrast, immediate transitions may be taken as soon as the state becomes active; this enables the activation and deactivation of a state multiple times within one instant. Delayed transitions can also be *count delayed*, *i. e.*, the trigger must have been evaluated to true for a specific number of times, before the transition is enabled. When a state has more than one outgoing transition, a unique *priority* is assigned to each of them, where lower numbers have higher priority. Weak abortions must have lower priority than strong abortions, and if a normal termination exists, it always has the lowest priority.

4. The Kiel Esterel Processor (KEP)

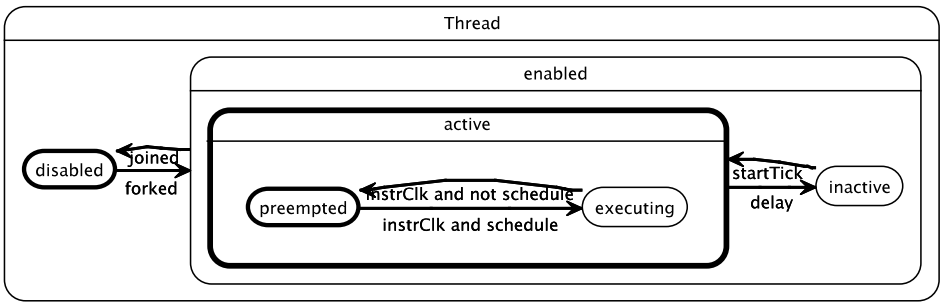
Contents

The *KEP* is a reactive processor, based on the synchronous language Esterel. It has three main features: the implementation of Esterel's preemption by *watchers*, *priority based threads* to implement synchronous concurrency, and a *tick manager* to enforce constant reaction times. The *KEP* was originally designed by Xin Li in VHDL [Li, 2007, Li and von Hanxleden, 2010]. It was later reimplemented by Malte Tiedje in Esterel [Tiedje, 2008], called *KEP* in Esterel (*KEP-e*).

Watchers sense the program counter and a trigger signal to determine whether a preemption takes place. The [w]abort S, A0 statement initializes a watcher with the code range from the current instruction to the label A0 and the trigger signal S. At each clock cycle, the watcher checks whether the current program counter is in the given range and whether S is present. If this is the case, the watcher signals that it is triggered. A watcher-handler determines whether the abortion shall be executed now and, if multiple



(a) Status of the whole program



(b) Execution status of a single thread

Figure 4.1.: Execution model of the KEP.

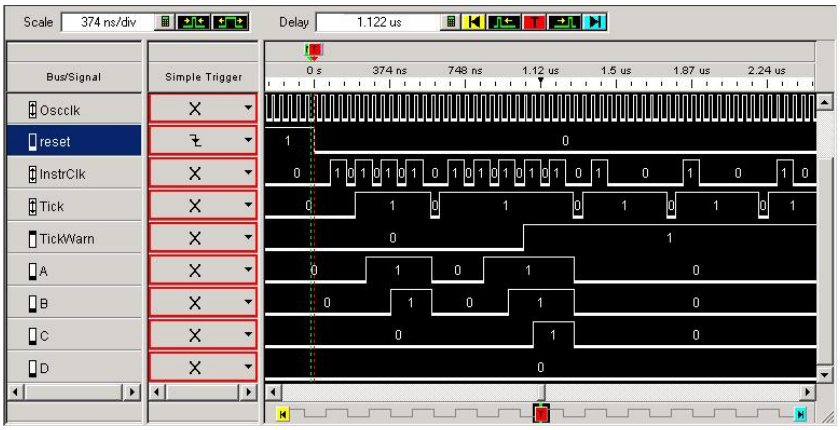
watchers are triggered, which one is taken. A weak abortion, for example, can only be taken if a tick-delimiting instruction is reached. If multiple watchers are active, the watcher with the lowest ID wins.

To implement concurrency, the *KEP* offers hardware threads and a priority-based scheduler. Each thread has an ID, a program counter, and a priority. Additionally, for each thread it is stored whether it is currently active and whether its execution is finished for the current tick. At each instruction cycle, the scheduler determines the thread with the highest priority, and the instruction to which its program counter points gets executed. If two threads have the same priority, the ID is used to determine the unique thread with the highest ID, which will then be executed. The different status of a thread at runtime and the status of the whole program within one tick is shown in Figure 4.1.

The tick manager starts the execution of a new tick and signals to the environment, when the execution for the current thread has finished. It supports two different modes: either the *KEP* runs as fast as possible or with a constant reaction time. If the special register _TICKLENGTH holds a value greater than 0, the tick manager will wait at least this number of instruction cycles before declaring the tick as finished. If the execution takes more clock cycles than the value in _TICKLENGTH, the output TICKWARN is set to true, to indicate a timing violation. The behavior of the tick manager for a small Esterel program with a timing violation is shown in Figure 4.2. Combined with a WCRT analysis of the executed program, the tick manager allows a constant reaction

```
1  INPUT D
2  OUTPUT A,B,C
3
4  EMIT _TICKLEN, #3
5  EMIT A
6  EMIT B
7  PAUSE
8  EMIT A
9  EMIT B
10 EMIT C
11 AWAIT D
```

(a) KEP assembler



(b) Timing diagram of the KEP

Figure 4.2.: Illustration of the tick manager (taken from [Li et al., 2005]). A timing violation occurs in the second tick, the tick takes 5 clock cycles, but TICKLEN has the value 3. The TickWarn output is set three clock cycles after the second tick is started. The execution times for all but the second tick are equal (3 clock cycles), even though in the last ticks only one await is executed.

time without any jitter. If the _TICKLENGTH register holds the value 0, the tick manager sends the information that the tick has finished as soon as all threads have reached a tick-delimiting instruction, *i. e.*, the *KEP* runs as fast as possible.

In the next section, a short overview of the instruction set of the *KEP* is given, including an informal semantics. In Section 4.2 the differences between the *KEP* and the *KEP-e* are described, before giving a formal semantics to a subset of the *KEP* instruction set in Section 4.3. In Section 4.4 and Section 4.5, we take a closer look at the compilation from Esterel and SyncCharts to the instruction set of the *KEP*.

4.1. Instruction Set Architecture

4.1.1. Execution cycle

Standard processors execute in the *von Neumann* cycle: *fetch* an instruction from the ROM, *decode* the instruction, and *execute* it. The *KEP* differs in two aspect from this behavior: 1) Before each fetch, there is a scheduling step to resolve the current program counter of the thread with the highest priority. 2) All other steps in the cycle can be aborted by the watcher, which will jump directly to the scheduling step. While the watcher acts in parallel in the hardware implementation of the processors, the execution cycle can be semantically sequentialized into the following steps:

1. *Schedule*: The scheduler determines the thread with the highest priority

2. *Strong abort*: The watchers for strong abortions and suspension are triggered and indicate whether they are active.

3. *Fetch, Decode, Execute*: The next instruction for the scheduled thread is read from the instruction ROM and executed.

4. *Weak abortion*: The watchers for weak abortions are triggered and indicate whether they are active.

The watchers are the main difference to other approaches for direct execution of Esterel programs. Semantically, it is sufficient to trigger the watchers only when a time delimiting instruction, like a PAUSE is executed: check for strong abortion and suspension when control is resumed at the PAUSE, and check for weak abortion when control reaches a PAUSE. This is done by other approaches to execute Esterel programs, like the EMPEROR [Yoong et al., 2006], the StarPRO [Yuan et al., 2008] or SC [von Hanxleden, 2009], where explicit instructions to check abortions are added before and after each PAUSE instruction. Which approach should be preferred depends on the Esterel program. Checking all preemptions in parallel gives a benefit for deep nestings, both in program size and execution time. However, if only few preemptions need to be checked, the more complex hardware and the resulting slower clock cycle do not pay off.

4.1.2. Instructions

The *KEP* has instructions for all Esterel kernel statements, and some additional instructions for commonly used derived statements, like AWAIT. The instructions can be distinguished in the following classes: time delimiting instructions, preemption instruction, thread instructions, control flow instructions and data-handling instructions. The latter two classes are similar to instructions in general purpose processors. Here we consider a restricted version of the *KEP* by ignoring all data handling except for signal status, similar to restricting full Esterel to pure Esterel. Figure 4.3 gives a short overview of the instructions of the *KEP*.

Time Delimiting

The AWAIT S and AWAITI S instruction wait for the presence of a trigger signal S. If the trigger signal is absent, they terminate the execution of the current thread. Otherwise, they pass the control to the following instruction. The AWAIT declares the thread finished for the current tick when activated, regardless of the signal status. In contrast, the AWAITI checks the signal status immediately. The *KEP* has no PAUSE instruction, but the special signal TICK, which is present in each tick can be used instead: AWAIT TICK waits for exactly one instant.

To implement Esterel's count delays, the LOAD _COUNT instruction can be used to set the _COUNT register. This register is read by the AWAIT instruction: it will not wait for the first occurrence of the trigger signal, but it will wait as many occurrences as the _COUNT register specifies.

Mnemonic, Operands	Esterel Syntax	Cycles	Notes
PAR *prio*$_1$, *startAddr*$_1$, *id*$_1$ ... PAR *prio*$_n$, startAddr$_n$, *id*$_n$ PARE *endAddr*, *prio* *startAddr*$_1$: ... *startAddr*$_2$: ... *startAddr*$_n$: ... *endAddr*: JOIN [*prio*]	[p$_1$ || ⋮ || p$_n$]	$n+1$ 1	For each thread, one PAR is needed to define the start address, thread id and initial priority. The end of a thread is defined by the start address of the next thread, except for the last thread, whose end is defined via PARE. The cycle count of a fork node depends on the count of threads.
PRIO *prio*		1	Set current thread priority to *prio*.
[W]ABORT[I, *n*] *S*, *endAddr* ... *endAddr*:	[**weak**] **abort** ... **when** [**immediate**, *n*] *S*	1	
SUSPEND[I,*n*] *S*, *endAddr* ... *endAddr*:	**suspend** ... **when** [**immediate**, *n*] *S*	1	
startAddr: ... EXIT *exitAddr startAddr* ... *exitAddr:*	**trap** *T* **in** ... **exit** *T* ... **end trap**	1	Exit from a trap, *startAddr*/*exitAddr* specifies trap scope. Unlike GOTO, check for concurrent EXITs and terminate enclosing ||.
PAUSE	**pause**	1	Wait for a signal. AWAIT TICK is equivalent to PAUSE.
AWAIT [I, *n*] *S*	**await** [**immediate**, *n*] *S*	1	
SIGNAL *S*	**signal** *S* **in** ... **end**	1	Initialize a local signal *S*.
EMIT *S* [, {#*data*|*reg*}]	**emit** *S* [(*val*)]	1	Emit (valued) signal *S*.
SUSTAIN *S* [, {#*data*|*reg*}]	**sustain** *S* [(*val*)]	1	Sustain (valued) signal *S*.
PRESENT *S*, *elseAddr*	**present** *S* **then** ... **end**	1	Jump to *elseAddr* if *S* is absent.
NOTHING	**nothing**	1	Do nothing.
HALT	**halt**	1	Halt the program.
addr: ... GOTO *addr*	**loop** ... **end loop**	1	Jump to *addr*.

Figure 4.3.: Overview of the KEP instruction set architecture, and their relation to Esterel and the number of processor cycles for the execution of each instruction.

The HALT instruction marks the execution of the current thread as finished for this tick, without changing the program counter. Hence the HALT instruction is executed in each instant. This is necessary, because the HALT can be aborted by a watcher, and the watcher needs the execution of at least one instruction in its range to be triggered.

Preemption

The ABORT S, endAddr, ID, WABORT S, endAddr, ID, and SUSPEND S, endAddr, ID instructions initialize a watcher with the given ID and trigger signal S and transfer the control to the following instruction. The preemption itself is then performed by the

watchers. The range of the watcher starts at the current address and ends at the given end address. Each of these instructions also has an immediate version, which already activate the watcher in the current tick. The IABORT does not initialize a watcher if the trigger signal is active when it is activated, but instead directly jumps to the end address. There are two way to deactivate a watcher again: control can simply jump out of the range of the watcher, or the watcher can be reinitialized with new values. A watcher can be explicitly deactivated by giving an empty range, *i. e.*, setting an end-address before the start-address. As the AWAIT instruction, the preemption instruction use the _COUNT register to achieve counted abortion, *i. e.*, the abortion does not take place at the first occurrence of the trigger signal, but waits for as many occurrences as specified in the _COUNT register.

Threads

The PAR Prio, StartAddr, ThreadID instruction initializes a thread, with the given priority and thread id. It also sets the end-address of the last initialized thread, if this was not already set by a preceding PARE instruction. The PARE instruction sets the end-address of the last initialized thread and sets the priority of the parent thread.

The JOIN prio instruction checks whether any child thread of the current thread is still active. If this is the case it terminates the current thread, otherwise it transfers the control to the following instructions. It also sets the priority of the current thread.

The PRIO prio instruction sets the priority of the current thread to prio. When the PRIO instruction is executed, the thread already has the highest priority of all active threads. Therefore raising the priority will not alter the scheduling in the current tick, the raising only has an impact on the next tick.

Control Flow and Data Handling

The EMIT S instruction emits S and transfers control to the following instruction, while the SUSTAIN S instruction emits S and stops the execution of the current thread for this tick. The SIGNAL S instruction is dual to the EMIT S, it sets the status of a signal S to absent. It is used to implement local signals in Esterel, which are declared via Esterel's signal keyword.

The PRESENT instruction implements a conditional jump, based on the status of a signal. Note that the jump is performed when the signal is absent; if it is present, control is transferred to the following instruction.

The NOTHING instruction transfers control directly to the following instruction, and the GOTO instruction transfers control to a given address.

4.2. KEP-e

The *KEP-e* [Tiedje, 2008] is a reimplementation of the *KEP*, which uses Esterel as a hardware description language to describe the processor itself. There were two main motivations for the reimplementation: to improve the maintainability of the *KEP* and

to test Esterel as a hardware description language on a project of reasonable size. Even though Esterel is intended to describe control dominated hardware controllers and not complex hardware systems like a processor, this turned out reasonably well [Tiedje and Traulsen, 2008]. However, it also became clear that the original Esterel v5 is not a good choice to describe hardware, but the more expressive Esterel v7 should be used. That version extends Esterel, *e. g.*, by arrays and bit-vectors, which can be easily mapped to hardware. It also extends Esterel by dataflow-like signal emissions, which can be mapped easier to efficient hardware than the usual, imperative style of emission in Esterel. We will come back to the topic of using Esterel for the description of a processor in Section 5.4.

There are two important parts of the *KEP* missing for the *KEP-e*: it lacks an ALU, hence it can only handle pure Esterel, and the *KEP-e* does not support Esterel traps. Since every Esterel program can be translated into an equivalent Esterel program without any traps by replacing a trap by a weak abortion [Schneider, 2009], this deficit can be compensated by the compiler. While replacing traps by weak abortion is difficult for some corner cases, it can be replaced directly by a weak abortion and a local signal in nearly all real programs.

4.2.1. Validation

The implementation of the *KEP-e* also allows formal verifications of (parts of) the *KEP*. This was for example used to show the correctness of the scheduler. A simple, but inefficient specification of the scheduler was written in Esterel. The behavioral equivalence to the actual implementation, which is both more involved and more efficient, was formally shown using the sequential equivalence check of Esterel Studio.

4.2.2. Connection to the "real world"

The *KEP-e*, running on an FPGA board, was successfully connected to external hardware and used to control simple Lego robots. To achieve this, the test-driver of the *KEP-e* was extended to read external inputs as well as inputs coming from the KReP Evalbench and the capability to run in a free mode, starting a new tick as soon as the last one has finished, instead of waiting for the KReP Evalbench to start a new tick. While the controlled systems were simple, this proved that the *KEP* can be easily connected to "the real world."

4.3. Semantics

The semantics of the *KEP* is given in Xin Li's thesis [Li, 2007], however, the behavior is only informally described. In particular, there are some cases where the behavior of the *KEP* and the *KEP-e* differ. The reference semantics for the *KEP* is the formal semantics of Esterel, but this only states that the strl2kasm compiler transform an Esterel program in such a way that its execution on the *KEP* has the same IO-behavior as the original Esterel program. This makes it unnecessarily hard to compile from other languages

than Esterel to the *KEP* Assembler (*KASM*), because the compiler has to mimic the strl2kasm compiler to ensure that the correct behavior is achieved.

```
1     ABORTI S, A0
2     EMIT S
3     EMIT O
4  A0: HALT
```

```
1     ABORT S, A0
2     PAUSE
3     EMIT S
4     EMIT O
5  A0: HALT
```

Figure 4.4.: Non constructive KEP assembler programs.

Consider for example the programs in Figure 4.4. In the program on the left side, the immediate strong abortion (ABORTI) will first check whether the trigger signal is present, and thereafter it will ignore it for the current tick: it is handled like a present not S then abort... when S. Hence, the code on the left will emit both signal S and O, in contrast to the behavior one expects from an immediate strong abort. In the code on the right, the watcher will trigger the abortion directly after the S is emitted and before the EMIT O is executed. While the Esterel program that corresponds to this example is non-constructive, and cannot be compiled for the *KEP*, such *KEP* assembler is also generated by the compiler for constructive and even acyclic programs, as we will see in Section 4.4.

While the behavior of most *KEP* assembler programs can be understood intuitively, at least if one knows Esterel, there are some weird corner-cases:

- When is the join executed? The *KEP-e* assumes that priority of the parent thread is lower than that of all child threads, therefore it is sufficient to execute the join only once after the execution of the child threads. The original *KEP* will execute the join whenever a child thread terminates. While the implementation in the original *KEP* needs less hardware resources, the implementation in the *KEP-e* gives a simpler behavior.

- What happens when the range of a thread is left by a jump or a conditional? The behavior is only defined when a thread is left by normal control flow. But leaving it by a present test would give an efficient implementation for transitions to final states in the translation from SyncCharts to *KEP* assembler.

- What happens when the range of an active watcher is left by a goto? The *KEP* computes the active watchers at each clock cycle based on the current signal values without storing the information that a watcher was active. Hence, leaving the range of a watcher simply deactivates it. On the other hand, what happens if we jump inside a watched range with a goto? This will reactivate the watcher, but not reinitialize it. Hence, it will not reset the flag that indicates whether the watcher is already active: the watcher is immediate.

- Which instructions trigger the watcher? In particular, what happens when the signal of a strong abort is emitted inside its body? In fact, the watchers are checked

in each instruction-cycle, hence any instruction can trigger an abort. From a theoretical point of view, it would be sufficient for constructive programs to trigger the watcher only when control ends or starts at a tick-delimiting instruction. In this case, the emit inside of an abort is temporarily ignored.

While the strl2kasm compiler assumes acyclic Esterel programs, which are hence constructive, it itself generates non-constructive programs in intermediate steps (see Section 4.4).

These examples should motivate that the behavior of the *KEP* is not always trivial and intuitive, so a formal semantics might be useful. However, we should ask ourselves, for what purpose do we need a formal semantics? The purpose for which we want to use the semantics defines how we want to define it. Basically, with the formal semantics we want to tackle three different problems:

1. *Intuitive but precise definition of the behavior.* While the informal explanation gives an introduction to the behavior of the *KEP*, a formal semantics can give a much better understanding. There is no need to follow the intuition of the author of the informal description, but one can check ones own intuition against a precise reference. Here, the main benefit of the *formal* semantics is that for each possible corner-case one gets a precise behavior, while in the informal semantics these cases could be silently ignored. However, the semantics need not to be complete, as long as it is precisely stated for which inputs the behavior is not defined. As a very rough abstraction of the *KEP* semantics the semantics of Esterel itself can be considered.

2. *Reference for the compiler and WCRT analysis.* While we still might abstract details of the *KEP*, the behavior must be captured in more detail, compared to a formal semantics that purpose is to help humans understand the behavior.

3. *Verification of the* KEP. If we want to prove the correctness of (parts of) the *KEP*, we need a very precise semantics of the behavior. In this sense, the description of the *KEP* in Esterel gives a formal semantics to the *KEP*, since Esterel is formally defined itself.

Similar to the restrictions of Esterel to kernel statements in order to define the semantics, we reduce the considered instruction sets to support only the features that are interesting for the semantics, *i. e.*, watchers and threads. All instructions that deal with data as well as traps are omitted. Preemptions have an explicit identifier to indicate the watcher, and we omit counted delays for preemptions: all preemptions react on the first occurrence of their trigger signal. We combine the PAR and PARE instruction into a combined instruction that initializes an arbitrary number of threads. Hence, we only give a semantics to programs where a PAR is directly followed by another PAR or PARE instruction, and all thread identifiers in such a block are distinct. We also require that thread that forks the threads continues its execution at the PARE without changing its priority. Figure 4.5 shows the simplified instruction set. Figure 4.6 shows the assembler

Identical

- JOIN *Prio*
- PRIO *Prio* ,*ThreadID*
- GOTO *addr*
- SIGNAL *S*
- PRESENT *S*,*elseAddr*
- HALT
- NOTHING

Removed

- EXIT *exitAddr startAddr*
- PAUSE

Simplified

KEP Assembler	Simplified
EMIT *S* [, {#*data*|*reg*}] SUSTAIN *S* [, {#*data*|*reg*}]	EMIT *S* SUSTAIN *S*
PAR $prio_1$, $startAddr_1$, id_1 ... PAR $prio_n$, $startAddr_n$, id_n PARE *endAddr*, prio	PAR [T_1, ..., T_n], *endAddr* with T_i=($prio_i$, $startAddr_i$, id_i)
AWAIT [I, *n*] *S* [W]ABORT[I, *n*] *S*, *endAddr* SUSPEND[I,*n*] *S*, *endAddr*	AWAIT[I] *S* [W]ABORT[I] *S*, *endAddr* ,*watcherID* SUSPEND[I] *S*, *endAddr* ,*watcherID*

Figure 4.5.: Simplified kernel instructions of the KEP: all data handling and traps are removed. Preemptions and await always react to the first occurrence of the trigger signal, and related PAR instructions are combined to one. PAUSE is equivalent to AWAIT TICK.

for the Esterel ABRO example as generated by the strl2kasm compiler and in the simplified instruction set. Beside the combination of PAR and PARE into one statement, the labels are replaced by the actual line number.

```
1  loop
2    abort
3      [
4        await A
5      ||
6        await B
7      ];
8      emit O
9    when R
10 end loop
```

(a) Esterel program

```
1  A0: ABORT R, A1
2      PAR 1, A2, 1
3      PAR 1, A3, 2
4      PARE A4, 1
5  A2: AWAIT A
6  A3: AWAIT B
7  A4: JOIN 0
8      EMIT O
9      HALT
10 A1: GOTO A0
```

(b) Generated *KEP* assembler

```
0  A0: ABORT R, 7
1      PAR [(1, 2, 1), (1, 3, 2)], 4)
2  A2: AWAIT A
3  A3: AWAIT B
4  A4: JOIN 0
5      EMIT O
6      HALT
7  A1: GOTO A0
```

(c) Simplified version of the assembler

Figure 4.6.: ABRO example in *KEP* assembler

Gérard Berry states about the semantics of Esterel [Berry, 2000]:

> "To deal with reactive or interactive systems, our first task is to look for an adequate concurrency model. Here, we mean a naive model that one can explain to non-computer scientists, not a 26-tuple of sets and relations."

The reader will notice that the semantics for the *KEP* assembler does not have this benefit.

Configuration A *configuration* of the *KEP* consists of the following parts:

- Since we only consider pure programs, without any data, we only need to know the set of present signals. We do not distinguish between input, output, and local signals.

$$signals: \ S \subseteq \mathbb{S}$$

- To initialize a watcher, we need the *id*, its type and the trigger signal *S*. The range is given by the *start* and *end* address. The start address is included in the range, the end address is excluded. The *active* flag is used to distinguish between immediate and delayed preemption. The *thread id* is needed when the preemption takes place: the program counter of this thread is resumed. One might expect that we do not need the thread id, because we can simply use the current thread when an abortion takes place. However, if there are multiple threads in the range of the watcher, which thread is used to execute the thread handler could depend on the dynamic behavior of the program or even on the input signals. Since the thread id is used for the scheduling, not all threads are equal.

$$\begin{aligned} watcher : &(id, type, sig, start, end, active, thread) \\ &\in \mathbb{N} \times \{\texttt{SA}, \texttt{WA}, \texttt{SUSP}\} \times \mathbb{S} \times \mathbb{N} \times \mathbb{N} \times \mathbb{B} \times \mathbb{N} \end{aligned}$$

- The status of thread contains, beside its *id*, the current program counter *pc* and the priority *prio*. Furthermore, we need to indicate whether the execution of the thread has finished for the current tick, this is done by the *done* flag. In the actual implementation, we need an additional flag, which indicates that a thread is active, *i. e.*, its *done* flag shall be reset at the next tick. In the semantics, we simply remove inactive threads from the list of threads. All threads, except for the main thread, contain a link to their *parent thread*. As long as no active threads are reinitialized by a PAR statement, the parent relation forms a tree of all active threads with the main thread as its root. For the normal termination of threads, each thread needs to know its *end address*. The main thread has the end address ∞; it never terminates. To implement the AWAIT statement, a thread also needs an optional *await* signal.

$$thread: \ \ (id, pc, prio, done, parent, end, await) \in \mathbb{N} \times \mathbb{N} \times \mathbb{N} \times \mathbb{B} \times \mathbb{N}_{bot} \times \mathbb{N} \times \mathbb{S}_{\perp}$$

- The instruction ROM $rom: \ pc \in \mathbb{N} \to instr \in$ Instruction. The ROM is not altered by any instruction.

To denote the replacement of a single entry e to a value v in a thread t or a watcher w, we write $t[e \leftarrow v]$ or $w[e \leftarrow v]$, respectively. We also extend this notation to sets: we denote the setting of a single entry e to value v in one element t of a set T by $T[t.e \leftarrow v]$. For a set S of watchers and thread we denote with $S_{\overline{id}} = S \setminus \{s \in S \mid s.id \neq id\}$ the set excluding the given id, and we also extend this notation to sets of ids.

Execution

We now give the semantics for the normal execution of each instruction. After each normal execution, we need to deactivate all normally terminated threads, *i. e.*, threads whose program counter is behind their end address. This is done by the *normalTerm* function which only returns threads that are not normally terminated:

$$normalTerm(T) = \{t \in T \mid t.pc < t.end\}$$

Note that the watchers are not deactivated. This allows to jump directly into a the range of a watcher, without reinitializing it.

One execution step has the form:

$$(S, W, T) \xrightarrow[at:\texttt{instr}]{} (S', W', T').$$

We take a configuration with a set of set of signals S, a set of watchers W and a set of threads T, and transform it, for a given active thread at and a given instruction *instr* into a new configuration.

Control Flow and Data Handling

- The EMIT sets the status of the signal to true:

$$(S, W, T) \xrightarrow[at:\mathsf{EMIT}\ A]{} (S \cup \{A\}, W, T[at.pc \leftarrow at.pc + 1])$$

- The SUSTAIN sets the signal to true and deactivates the thread:

$$(S, W, T) \xrightarrow[at:\mathsf{SUSTAIN}\ A]{} (S \cup \{A\}, W, T[at.done \leftarrow true])$$

- NOTHING only increases the current program counter:

$$(S, W, T) \xrightarrow[at:\mathsf{NOTHING}]{} (S, W, T[at.pc \leftarrow at.pc + 1])$$

- The SIGNAL instruction declares a new signal, which is initially absent. Hence, if the signal is present in the current configuration, it is removed.

$$(S, W, T) \xrightarrow[at:\mathsf{SIGNAL}\ A]{} (S \setminus \{A\}, W, T[at.pc \leftarrow at.pc + 1])$$

- GOTO jumps to its address:

$$(S, W, T) \xrightarrow[at:\mathsf{GOTO}\ addr]{} (S, W, T[at.pc \leftarrow addr])$$

- The PRESENT jumps to its continuation, if the signal A is not present:

$$\frac{A \notin S}{(S, W, T) \xrightarrow[at:\mathsf{PRESENT}\ A,\ else]{} (S, W, T[at.pc \leftarrow else])}$$

  Otherwise it activates the next instruction:

$$\frac{A \in S}{(S, W, T) \xrightarrow[at:\mathsf{PRESENT}\ A,\ else]{} (S, W, T[at.pc \leftarrow at.pc + 1])}$$

Time Delimiting

- The HALT deactivates the thread:

$$(S, W, T) \xrightarrow[at:\mathsf{HALT}]{} (S, W, T[at.done \leftarrow true])$$

- For the AWAIT, we have to distinguish three cases. In the initial phase, we set the *await* flag of the thread and deactivate the thread. We determine that we are in the initial phase by checking whether the *await* flag is already set.

$$\frac{at.await = \bot}{(S, W, T) \xrightarrow[at:\mathsf{AWAIT}\ A]{} (S, W, T[at.done \leftarrow true, at.await = A])}$$

  In every later step, the behavior depends on the status of the trigger signal A:

$$\frac{at.await = A \qquad A \notin S}{(S, W, T) \xrightarrow[at:\mathsf{AWAIT}\ A]{} (S, W, T[at.done \leftarrow true])}$$

$$\frac{at.await = A \qquad A \in S}{(S, W, T) \xrightarrow[at:\mathsf{AWAIT}\ A]{} (S, W, T[at.pc \leftarrow at.pc + 1, at.await \leftarrow \bot])}$$

- If the signal A is not present, the AWAITI behaves like a HALT:

$$\frac{A \notin S}{(S, W, T) \xrightarrow[at:\mathsf{AWAITI}\ A]{} (S, W, T[at.done \leftarrow true])}$$

  Otherwise is goes to the next instruction:

$$\frac{A \in S}{(S, W, T) \xrightarrow[at:\mathsf{AWAITI}\ A]{} (S, W, T[at.pc \leftarrow at.pc + 1])}$$

Preemption

- The ABORT, WABORT, and SUSPEND initialize the watcher accordingly. The watchers are not activated.

$$(S, W, T) \xrightarrow[at:\mathsf{ABORT}\ A,\ end,\ ID]{} (S, W_{\overline{ID}} \cup \{(ID, \mathtt{SA}, A, at.pc + 1, end, false, at)\}, T[at.pc \leftarrow at.pc + 1])$$

$$(S, W, T) \xrightarrow[at:\mathsf{WABORT}\ A,\ end,\ ID]{} (S, W_{\overline{ID}} \cup \{(ID, \mathtt{WA}, A, at.pc + 1, end, false, at)\}, T[at.pc \leftarrow at.pc + 1])$$

$$(S, W, T) \xrightarrow[at:\mathsf{SUSPEND}\ \mathsf{A},\ end,\ ID]{} (S, W_{\overline{ID}} \cup \{(ID, \mathtt{SUSP}, A, at.pc + 1, end, false, at)\}, T[at.pc \leftarrow at.pc + 1])$$

- The ABORTI checks the current status of the trigger signal A. If it is present, the watcher is not initialized at all.

$$\frac{A \in S}{(S,W,T) \xrightarrow[at:\mathsf{ABORTI}\ A,\ end,\ ID]{} (S,W,T[at.pc \leftarrow end])}$$

  If the trigger signal is absent, the ABORTI behaves like the delayed abortion. Note that the watcher is not activated.

$$\frac{A \notin S}{(S,W,T) \xrightarrow[at:\mathsf{ABORTI}\ A,\ end,\ ID]{} (S, W_{\overline{ID}} \cup \{(ID, \mathtt{SA}, A, at.pc+1, end, false)\}, T[at.pc \leftarrow at.pc+1])}$$

- The WABORTI behaves like the delayed versions, except that the watcher is activated immediately:

$$(S,W,T) \xrightarrow[at:\mathsf{WABORTI}\ A,\ end,\ ID]{} (S, W_{\overline{ID}} \cup \{(ID, \mathtt{WA}, A, at.pc+1, end, true)\}, T[at.pc \leftarrow at.pc+1])$$

- The SUSPENDI marks the current thread as done, if the trigger signal A is present. Note that the program counter is not altered, because we want to achieve the same behavior in the next tick, if the trigger signal is still present. The watcher is not initialized until the trigger signal is absent.

$$\frac{A \in S}{(S,W,T) \xrightarrow[at:\mathsf{SUSPENDI}\ A,\ end,\ ID]{} (S,W,T[at.done \leftarrow true])}$$

  If the trigger signal A is absent, the SUSPENDI initializes the watcher and increments the program counter. In contrast to the delayed suspend, the watcher is immediately active.

$$\frac{A \notin S}{(S,W,T) \xrightarrow[at:\mathsf{SUSPEND}\ A,\ end,\ ID]{} (S, W \cup \{(ID, \mathtt{SUSP}, A, at.pc+1, end, true)\}, T[at.pc \leftarrow at.pc+1])}$$

Threads

- The PAR initializes the new threads:

$$(S,W,T) \xrightarrow[at:\mathsf{PAR}\ [T_0,\ldots,T_n],\ end]{} (S, W, T_{\overline{ids}}[at.pc \leftarrow end] \cup \{(T_0.id, T_0.pc, T_0.prio, false, at.id, T_1.pc, \bot), \ldots, (T_n.id, T_n.pc, T_n.prio, false, at.id, end, \bot)\})$$

  with $T_i = (T_i.prio, T_i.pc, T_i.id)$ and $ids = \{T_0.id, \ldots, T_n.id\}$

- The JOIN checks whether all its child threads have terminated. If this is the case, it increases the program counter.

$$\frac{\forall t \in T : t.parent \neq at}{(S, W, T) \xrightarrow[at:\mathsf{JOIN}\ prio]{} (S, W, T[at.pc \leftarrow at.pc + 1, at.prio \leftarrow prio])}$$

  Otherwise, it marks the current thread as done and updates its priority.

$$\frac{\exists t \in T : t.parent = at}{(S, W, T) \xrightarrow[at:\mathsf{JOIN}\ prio]{} (S, W, T[at.done \leftarrow true, at.prio \leftarrow prio])}$$

Termination

As a last step, we have to check whether the execution has finished for the current tick. Since we directly remove all inactive threads, this simply means that we have to check that the done flag for all threads is set:

$$done(T) = \bigwedge_{t \in T} t.done$$

Strong Preemptions

The function *sabort* computes whether a strong preemption, *i. e.*, a strong abort or a suspend, takes place. It returns either $\perp$, if no preemption is triggered, or a set of new watchers and threads. The inputs of the function are the active thread at, and the current signals S, watchers W and threads T:

$$sabort(at, S, W, T) \rightarrow (W', T')$$

First we need to determine the set of active watchers, these are the watchers for which the currently executed program counter is in the given range and the trigger signal is present.

$$\begin{aligned} activeSWatchers = \{w \in W \mid\ & w.start \leq at.pc < w.end \wedge w.sig \in S \\ & \wedge w.Type \in \{\mathtt{SA}, \mathtt{SUSP}\} \wedge w.active\} \end{aligned}$$

If this set is empty, *sabort* returns $\perp$. The active watcher (aW) is the one from this set with the maximal id.

$$aW = \max\{w.id \mid w \in activeSWatchers\}$$

Based on the type of the active Watcher, we need to determine the new active threads and watchers. For the strong abortion, we terminate all threads that are currently inside the range. Another possibility would be to terminate the active thread and recursively all children threads. For proper nesting of threads and watchers, as it is produced by the strl2kasm compiler, both versions are equivalent. The suspend only stops the execution

of the current thread by marking it as terminated, hence one step in the semantics is needed for each thread inside a triggered suspend.

$$T' = \begin{cases} \{t \in T \mid t.pc < aW.start \vee t.pc > aW.end\} & \\ \cup \{t[pc = aW.end] \mid t \in T \wedge t.id = aW.thread\} & \text{if } aW.type = \texttt{SA} \\ T[at.done = true] & \text{if } aW.type = \texttt{SUSP} \end{cases}$$

We terminate all watchers that are completely inside the active Watcher. This differs from the behavior of the actual implementation of the *KEP*, which uses the parent relation, however, both behaviors are equivalent if watchers are properly nested, *i. e.*, either a watcher is completely nested into another one, or their ranges are disjoint.

$$W' = \begin{cases} \{w \in W \mid w.start < aW.start \vee w.end > aW.end\} & \text{if } aW.type = \texttt{SA} \\ W & \text{if } aW.type = \texttt{SUSP} \end{cases}$$

Weak Abortions

The *wabort* function checks for active weak abortions in a configuration. It returns either new sets of watchers and threads, or $\perp$ if no weak abortion is triggered. First, we have to know all active watchers of the type weak abortion. In contrast to the strong abortion, we have to check that all threads inside the watcher are marked as terminated, because a weak abortion permits the execution of all code inside its range.

$$activeWWatchers = \{w \in W \mid w.sig \in S \wedge w.Type = \texttt{WA} \wedge w.active \wedge \\ \forall_{t \in T}(w.start \leq t.pc \wedge w.end > t.pc) \implies t.done\}$$

If this set is empty, *wabort* returns $\perp$. In contrast to the strong abort, where the active watcher is the one with the highest *id*, we want to execute *all* active weak abortions and therefore execute the watcher with the lowest *id*. Other active watchers with a higher *id* are executed when the handler of the current watcher terminates.

$$aW = \min\{w.id \mid w \in activeWWatchers\}$$

For the new configuration, we remove all threads whose program counter is in the range of the watcher from the set of threads, and we resume the thread that is given by the watcher at the end address of the watcher range.

$$T' = \{t \in T \mid t.pc < aW.start \vee t.pc > aW.end\} \\ \cup \{t[pc = aW.end] \mid t \in T \wedge t.id = aW.thread\}$$

We remove the active Watcher from the watcher set. A weak abortion does not deactivate any other watcher, since a weak abort allows the execution of all statements inside its body.

$$W' = W \setminus aW$$

Scheduler

The *schedule* function determines the active thread with the lowest priority. If more than one thread has the minimal priority, the one with the highest thread ID is chosen. The rationale behind this is that usually parent threads have a lower thread ID than their child threads, hence if all threads have the same priority, child threads are always executed first.

First we determine the set of active threads of a set of threads T. For each configuration that has not terminated (see below), this set is not empty. The selected thread is the one with the smallest priority:

$$schedule(T) = min_{\prec} \{t \in T \mid \neg t.done\}$$

where

$$i \prec j \Leftrightarrow i.prio < j.prio \vee (i.prio = j.prio \wedge i.id > j.id)$$

4.3.1. Microstep

A micro-step corresponds to one instruction cycle of the program. It consists of the following sub-steps:

1. *Schedule*: determine the currently executed thread.
2. *Strong abortion*: determine whether any strong abort or suspension prevents the active thread from executing.
3. *Execution*: actually execute one instruction.
4. *Weak abortion*: check whether any weak abortion takes place.
5. *Update threads*: update thread status for all threads, in particular, remove threads whose program counter has left the range of the thread, running the $normalTerm$ function.

A micro-step transforms a configuration, with present signals S, watchers W, and threads T, into a new one, according to a given instruction ROM:

$$(S, W, T) \xrightarrow[\texttt{ROM}]{done} (S', W', T')$$

The *done* indicates whether the execution has finished for this tick. A micro-step can have one of the following three forms:

- A *normal execution* takes place if neither a weak nor a strong preemption is triggered. In this case, the resulting configuration is determined by the execution of

the current instruction ROM[$at.pc$] of the active thread aT, *i. e.*, the thread that is currently scheduled.

$$\frac{schedule(T) = at \qquad (S,W,T) \xrightarrow[at:\texttt{ROM[}at.pc\texttt{]}]{} (S',W',T') \qquad sabort(at,S,W,T) = \bot \qquad wabort(S',W',T') = \bot}{(S,W,T) \xrightarrow[\texttt{ROM}]{done(T')} (S',W',normalTerm(T'))}$$

- A *strong abortion* takes place if at least one watcher is triggered when executing the scheduled instruction. The threads and watchers are updated according to the triggered watcher, because other watchers or threads could be deactivated. The active instruction is not executed and weak abortions are not checked.

$$\frac{schedule(T) = at \qquad sabort(at,S,W,T) = (W',T')}{(S,W,T) \xrightarrow[\texttt{ROM}]{done(T')} (S,W',T')}$$

- A *weak abortion* abortion takes place, if a weak abortion watcher is triggered after the execution of the active instruction, but no strong abortion was triggered before. The resulting configuration after the execution is then altered according to the triggered weak abortion.

$$\frac{schedule(T) = at \qquad (S,W,T) \xrightarrow[at:\texttt{ROM[at.pc]}]{} (S',W',T') \qquad sabort(at,S,W,T) = \bot \qquad wabort(S',W',T') = (W'',T'')}{(S,W,T) \xrightarrow[\texttt{ROM}]{done(T'')} (S',W'',T'')}$$

4.3.2. Macro-Steps

A macro-step executes the program for a whole step. It consists of a chain of micro-steps, which are executed until the execution of all threads is done, *i. e.*, the *done* function returns true. Furthermore, the configuration needs to be initialized in each tick. This will remove all terminated threads and watchers, set all other watchers active and reset the termination flag of all other threads.

A macro-step has the form

$$(W,T) \xrightarrow[O]{I} (W',T') \Leftrightarrow init(I,W,T) \xrightarrow{false} C_0 \xrightarrow{false} \cdots C_n \xrightarrow{true} (O,W',T')$$

where the *init* function is defined as:

$$init(S,W,T) = (I, \{w \in W \mid w.thread \in T'\}, T' = \{t \in T \mid t.pc < t.end\})$$

4.3.3. Example Execution

We illustrate the semantics by giving the formal execution for the simplified version of the Esterel ABRO program from Figure 4.6.

Figure 4.7 shows the behavior in the first tick. To simplify reading, we abbreviate both watcher and threads inside each step if they are not used, *e. g.*, T_i^j denotes the thread with id i after the j-th microstep. Changed and additional values are marked bold in the final configuration of each microstep. We precede the value of each program counter with L for code-line, to help distinguishing program counters and identifier. Since no weak abortion occurs in this example, we omit this check. Figure 4.8 and Figure 4.9 show the behavior in the second tick, when the signals A and B, or R are present, respectively.

4.3.4. Limitations

Our semantics does still not capture the complete behavior of the *KEP*, in particular, the interaction between different PAR instructions is not matched. Furthermore, the timing behavior of the *KEP* is not matched. The *KEP*, for example, executes each PAUSE instruction twice. One time to terminate the thread, and again in the next instance to check for abortions.

This semantics should be seen as first step towards a formal semantics for the *KEP*. It still remains to compare the semantics carefully to the actual behavior of the *KEP*. Once it has been shown that the actual behavior of the *KEP* matches this semantics, it can be used to validate the compilers from Esterel and SyncCharts into *KEP* assembler.

4.4. Compiling Esterel

Since the instruction set of the *KEP* is closely related to Esterel, the compilation of many Esterel programs is straight-forward. For the compiler, there remain two important things to do:

1. *Dismantling*: Rewrite complex Esterel statements that cannot be directly mapped to *KEP* instructions. This also includes complex signal expressions.

2. *Scheduling*: Detect signal dependencies and order the threads accordingly. Assign thread ids corresponding to this order, and add priority statements if necessary.

The first step is a source to source transformation, while the second step is applied to an intermediate format, the CKAG.

A complete description of the strl2kasm compiler can be found in Marian Boldt's diploma thesis [Boldt, 2007a]. Here, we will only show how the capability of the *KEP* to execute non-constructive programs can be used in the code generation.

Since the *KEP* simply executes its code, it gives a semantics to non-constructive Esterel programs. However, this semantics also depends on the translation from Esterel to *KEP* assembler. For concurrency, the priority assignment of the compiler is crucial for the behavior, and also the exact dismantling of complex statements is part of the

1. ABORT R, L7, 0

$$\frac{schedule(\{T_0^0\}) = 0 \quad sabort(0,\{\},\{\},\{T_0^0\}) = \bot \quad (\{\},\{\},\{(0,L0,0,F,\bot,\infty,\bot)\}) \xrightarrow[\texttt{0:ABORT R,7,0}]{} (\{\},\{(0,\texttt{SA},R,L1,L7,F,0)\},\{(0,L1,0,F,\bot,\infty,\bot)\})}{(\{\},\{\},\{\underbrace{(0,L0,0,F,\bot,\infty,\bot)}_{T_0^0}\}) \xrightarrow{false} (\{\},\{\underbrace{(\mathbf{0},\texttt{SA},\mathbf{R},\mathbf{L1},\mathbf{L7},\mathbf{F},\mathbf{0})}_{W_0^1}\},\{\underbrace{(0,\mathbf{L1},0,F,\bot,\infty,\bot)}_{T_0^1}\})}$$

2. PAR [(1, 2, 1), (1, 3, 2)], 4

$$\frac{schedule(\{T_0^1\}) = 0 \quad sabort(0,\{\},\{W_0^1\},\{T_0^1\}) = \bot \quad (\{\},\{W_0^1\},\{(0,L1,0,F,\bot,\infty,\bot)\}) \xrightarrow[\texttt{PAR [(1,2,1),(1,3,2)],4}]{} (\{\},\{W_0^1\},\{(0,L4,1,F,\bot,\infty,\bot),(1,L2,1,F,0,L3,\bot),(2,L3,1,F,0,L4,\bot)\})}{(\{\},\{\underbrace{(0,\texttt{SA},R,L1,L7,F,0)}_{W_0^1}\},\{\underbrace{(0,L1,0,F,\bot,\infty,\bot)}_{T_0^1}\}) \xrightarrow{false} (\{\},\{\underbrace{(0,\texttt{SA},R,L1,L7,F,0)}_{W_0^2}\},\{\underbrace{(0,\mathbf{L4},1,F,\bot,\infty,\bot)}_{T_0^2},\underbrace{(\mathbf{1},\mathbf{L2},\mathbf{1},\mathbf{F},\mathbf{0},\mathbf{L3},\bot)}_{T_1^2},\underbrace{(\mathbf{2},\mathbf{L3},\mathbf{1},\mathbf{F},\mathbf{0},\mathbf{L4},\bot)}_{T_2^2}\})}$$

3. AWAIT B

$$\frac{schedule(\{T_0^2,T_1^2,T_2^2\}) = 2 \quad sabort(2,\{\},\{W_0^2\},\{T_0^2,T_1^2,T_2^2\}) = \bot \quad \dfrac{T_2^2.await = \bot}{(\{\},\{W_0^2\},\{T_0^2,T_1^2,(2,L3,1,F,0,L4,\bot)\}) \xrightarrow[\texttt{AWAIT B}]{} (\{\},\{W_0^2\},\{T_0^2,T_1^2,(2,L3,1,T,0,L4,B)\})}}{(\{\},\{\underbrace{(0,\texttt{SA},R,L1,L7,F,0)}_{W_0^2}\},\{\underbrace{(0,L4,1,F,\bot,\infty,\bot)}_{T_0^2},\underbrace{(1,L2,1,F,0,L3,\bot)}_{T_1^2},\underbrace{(2,L3,1,F,0,L4,\bot)}_{T_2^2}\}) \xrightarrow{false} (\{\},\{\underbrace{(0,\texttt{SA},R,L1,L7,F,0)}_{W_0^3}\},\{\underbrace{(0,L4,1,F,\bot,\infty,\bot)}_{T_0^3},\underbrace{(1,L2,1,F,0,L3,\bot)}_{T_1^3},\underbrace{(2,L3,1,\mathbf{T},0,L4,\mathbf{B})}_{T_2^3}\})$$

4. AWAIT A

$$\frac{schedule(\{T_0^3,T_1^3,T_2^3\}) = 1 \quad sabort(1,\{\},\{W_0^3\},\{T_0^3,T_1^3,T_2^3\}) = \bot \quad \dfrac{T_1^3.await = \bot}{(\{\},\{W_0^3\},\{T_0^3,(1,L2,1,F,0,L3,\bot),T_2^3\}) \xrightarrow[\texttt{AWAIT A}]{} (\{\},\{W_0^3\},\{T_0^3,(1,L2,1,T,0,L3,A),T_2^3\})}}{(\{\},\{\underbrace{(0,\texttt{SA},R,L1,L7,F,0)}_{W_0^3}\},\{\underbrace{(0,L4,1,F,\bot,\infty,\bot)}_{T_0^3},\underbrace{(1,L2,1,F,0,L3,\bot)}_{T_1^3},\underbrace{(2,L3,1,T,0,L4,B)}_{T_2^3}\}) \xrightarrow{false} (\{\},\{\underbrace{(0,\texttt{SA},R,L1,L7,F,0)}_{W_0^4}\},\{\underbrace{(0,L4,1,F,\bot,\infty,\bot)}_{T_0^4},\underbrace{(1,L2,1,\mathbf{T},0,L3,\mathbf{A})}_{T_1^4},\underbrace{(2,L3,1,T,0,L4,B)}_{T_2^4}\})$$

5. JOIN 1

$$\frac{schedule(\{T_0^4,T_1^4,T_2^4\}) = 0 \quad sabort(0,\{\},\{W_0^4\},\{T_0^4,T_1^4,T_2^4\}) = \bot \quad \dfrac{T_1^4.parent = 0 \quad T_2^4.parent = 0}{(\{\},\{W_0^4\},\{(0,L4,1,F,\bot,\infty,\bot),T_1^4,T_2^4\}) \xrightarrow[\texttt{JOIN 1}]{} (\{\},\{W_0^4\},\{(0,L4,1,T,\bot,\infty,\bot),T_1^4,T_2^4\})}}{(\{\},\{\underbrace{(0,\texttt{SA},R,L1,L7,F,0)}_{W_0^4}\},\{\underbrace{(0,L4,1,F,\bot,\infty,\bot)}_{T_0^4},\underbrace{(1,L2,1,T,0,L3,A)}_{T_1^4},\underbrace{(2,L3,1,T,0,L4,B)}_{T_2^4}\}) \xrightarrow{true} (\{\},\{\underbrace{(0,\texttt{SA},R,L1,L7,F,0)}_{W_0^5}\},\{\underbrace{(0,L4,1,\mathbf{T},\bot,\infty,\bot)}_{T_0^5},\underbrace{(1,L2,1,T,0,L3,A)}_{T_1^5},\underbrace{(2,L3,1,T,0,L4,B)}_{T_2^5}\})$$

Figure 4.7.: Example execution of ABRO in the formal semantics: First Tick

1. AWAIT B, thread 2 is removed by $normaleTerm$

$$\frac{schedule(\{T_0^0, T_1^0, T_2^0\}) = 2 \quad sabort(2, \{A, B\}, W_0^0, \{T_1^0, T_1^0, T_2^0\}) = \bot \quad \frac{B \in \{A, B\} \quad T_2^0.await = B}{(\{A, B\}, \{W_0^0\}, \{T_0^0, T_1^0, (2, L2, 1, T, 0, L4, B)\}) \xrightarrow[\texttt{2: AWAIT B}]{} (\{A, B\}, \{W_0^0\}, \{T_0^0, T_1^0, (2, \mathbf{L3}, 1, T, 0, L4, \bot)\})}}{(\{A, B\}, \{\underbrace{(0, \texttt{SA}, R, L1, L7, T, 0)}_{W_0^0}\}, \{\underbrace{(0, L4, 1, F, \bot, \infty, \bot)}_{T_0^0}, \underbrace{(1, L2, 1, F, 0, L3, A)}_{T_1^0}, \underbrace{(2, L3, 1, F, 0, L4, B)}_{T_2^0}\}) \xrightarrow{false} (\{A, B\}, \{\underbrace{(0, \texttt{SA}, R, L1, L7, T, 0)}_{W_0^1}\}, \{\underbrace{(0, L4, 1, F, \bot, \infty, \bot)}_{T_0^1}, \underbrace{(1, L2, 1, T, L0, 3, A)}_{T_1^1}\})$$

2. AWAIT A, thread 1 is removed by $normaleTerm$

$$\frac{schedule(\{T_0^1, T_1^1\}) = 1 \quad sabort(1, \{A, B\}, W_0^1, \{T_1^1, T_1^1\}) = \bot \quad \frac{A \in \{A, B\} \quad T_1^1.await = A}{(\{A, B\}, \{W_0^1\}, \{T_0^1, (1, L2, 1, T, 0, L3, A)\}) \xrightarrow[\texttt{2: AWAIT A}]{} (\{A, B\}, \{W_0^1\}, \{T_0^1, (1, \mathbf{L3}, 1, T, 0, L3, \bot)\})}}{(\{A, B\}, \{\underbrace{(0, \texttt{SA}, R, L1, L7, T, 0)}_{W_0^1}\}, \{\underbrace{(0, L4, 1, F, \bot, \infty, \bot)}_{T_0^1}, \underbrace{(1, L2, 1, T, 0, L3, A)}_{T_1^1}\}) \xrightarrow{false} (\{A, B\}, \{\underbrace{(0, \texttt{SA}, R, L1, L7, T, 0)}_{W_0^2}\}, \{\underbrace{(0, L4, 1, F, \bot, \infty, \bot)}_{T_0^2}\})$$

3. JOIN

$$\frac{schedule(\{T_0^1\}) = 0 \quad sabort(0, \{A, B\} W_0^1, \{T_1^1\}) = \bot \quad \frac{T_0^2.parent \neq 0}{(\{A, B\}, \{W_0^2\}, \{(0, L4, 1, F, \bot, \infty, \bot)\}) \xrightarrow[\texttt{0:JOIN}]{} (\{A, B\}, \{W_0^3\}, \{(0, L5, 1, F, \bot, \infty, \bot)\})}}{(\{A, B\}, \{\underbrace{(0, \texttt{SA}, R, L1, L7, T, 0)}_{W_0^2}\}, \{\underbrace{(0, L4, 1, F, \bot, \infty, \bot)}_{T_0^2}\}) \xrightarrow{false} (\{A, B\}, \{\underbrace{(0, \texttt{SA}, R, L1, L7, T, 0)}_{W_0^3}\}, \{\underbrace{(0, \mathbf{L5}, 1, F, \bot, \infty, \bot)}_{T_0^3}\})$$

4. EMIT

$$\frac{schedule(\{T_0^1\}) = 0 \quad sabort(0, \{A, B\} W_0^1, \{T_1^1\}) = \bot \quad (\{A, B\}, \{W_0^3\}, \{(0, L5, 1, F, \bot, \infty, \bot)\}) \xrightarrow[\texttt{0: EMIT O}]{} (\{A, B, \mathbf{O}\}, \{W_0^3\}, \{(0, L6, 1, F, \bot, \infty, \bot)\})}{(\{A, B\}, \{\underbrace{(0, \texttt{SA}, R, L1, L7, T, 0)}_{W_0^3}\}, \{\underbrace{(0, L5, 1, F, \bot, \infty, \bot)}_{T_0^3}\}) \xrightarrow{false} (\{A, B, \mathbf{O}\}, \{\underbrace{(0, \texttt{SA}, R, L1, L7, T, 0)}_{W_0^4}\}, \{\underbrace{(0, \mathbf{L6}, 1, F, \bot, \infty, \bot)}_{T_0^4}\})$$

5. HALT

$$\frac{schedule(\{T_0^4\}) = 0 \quad sabort(0, \{\}, \{W_0^4\}, \{T_1^4\}) = \bot \quad \frac{}{(\{A, B, O\}, \{W_0^4\}, \{(0, L6, 1, F, \bot, \infty, \bot)\}) \xrightarrow[\texttt{HALT}]{} (\{A, B, O\}, \{W_0^4\}, \{(0, L6, 1, T, \bot, \infty, \bot)\})}}{(\{A, B, O\}, \{\underbrace{(0, \texttt{SA}, R, L1, L7, T, 0)}_{W_0^4}\}, \{\underbrace{(0, L6, 1, F, \bot, \infty, \bot)}_{T_0^4}\}) \xrightarrow{true} (\{A, B, O\}, \{\underbrace{(0, \texttt{SA}, R, L1, L7, T, 0)}_{W_0^5}\}, \{\underbrace{(0, L6, 1, \mathbf{T}, \bot, \infty, \bot)}_{T_0^5}\})$$

Figure 4.8.: Example execution of ABRO in the formal semantics: Second Tick, A and B present

1. Strong abort

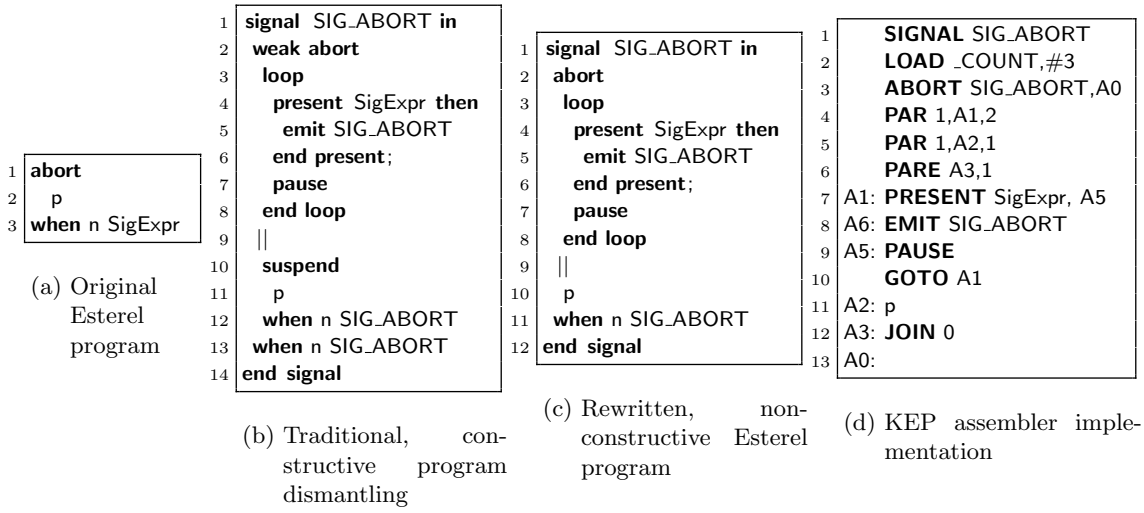
$$\frac{schedule(T_0^0, T_1^0, T_2^0) = 2 \quad sabort(2, \{R\}, \{(0, \texttt{SA}, R, L1, L7, T, 0)\}, \{(0, L4, 1, F, \bot, \infty, \bot), (1, L2, 1, F, 0, L3, A), (2, L3, 1, F, 0, L4, B)\}) = (\{\}, \{(0, L7, 1, F, \bot, \infty, \bot)\})}{(\{R\}, \{\underbrace{(0, \texttt{SA}, R, L1, L7, T, 0)}_{W_0^0}\}, \{\underbrace{(0, L4, 1, F, \bot, \infty, \bot)}_{T_0^0}, \underbrace{(1, L2, 1, F, 0, L3, A)}_{T_1^0}, \underbrace{(2, L3, 1, F, 0, L4, B)}_{T_2^0}\}) \xrightarrow{false} (\{R\}, \{\}, \{\underbrace{(0, 7, 1, F, \bot, \bot)}_{T_0^1}\})}$$

2. GOTO

$$\frac{schedule(T_0^1) = 0 \quad sabort(2, \{R\}\{\}, \{T_0^1\}) = \bot \quad (\{R\}, \{\}, \{(0, 7, 1, F, \bot, \bot)\}) \xrightarrow[\texttt{2: GOTO 0}]{} (\{R\}, \{\}, \{(0, L0, 1, F, \bot, \infty, \bot)\})}{(\{R\}, \{\}, \{\underbrace{(0, 7, 1, F, \bot, \bot)}_{T_0^1}\}) \xrightarrow{false} (\{R\}, \{\}, \{\underbrace{(0, \mathbf{L0}, 1, F, \bot, \infty, \bot))}_{T_0^2}\})}$$

3. From here on, the behavior is equivalent to the first tick

Figure 4.9.: Example execution of ABRO in the formal semantics: Second Tick, R present



(a) Original Esterel program

(b) Traditional, constructive program dismantling

(c) Rewritten, non-constructive Esterel program

(d) KEP assembler implementation

Figure 4.10.: Translating strong abort into KEP assembler

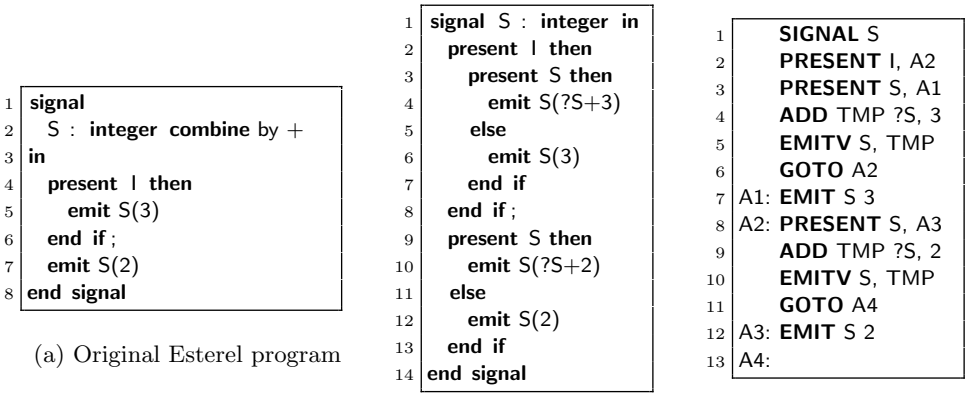
semantics for full Esterel. For simple programs without concurrency and complex statements, there is a one to one mapping into *KEP* assembler. For example, the well known, non-constructive Esterel program abort emit A when immediate A will be translated into the *KEP* assembler:

```
1  IABORT A, A0
2  EMIT A
3 A0:
```

This program has the unique reaction that emits A. Traditional semantics for synchronous programs consider the least fixed-point of the signal status to be the unique reaction. However, for the operational semantics given by the *KEP*, the signal status of the reaction is not a fixed-point. In this sense, the *KEP* assembler itself is not a synchronous language. For the *KEP* all program are deterministic and reactive, the processor will simply execute the program, and produce some output. In the notation of Huizing and Gerth [1992], *KEP* assembler is causal and responsive, but not modular.

4.4.1. Implementing Strong Abort

The general form for a strong abort in full Esterel is abort p when n Sig, where n is an integer and Sig a signal expression. While the *KEP* supports counted delays, the abort trigger needs to be a signal, not a signal expression. To implement this on the *KEP*, we have to evaluate the expression and map the result to a new signal, as shown in Figure 4.10. We could do this globally. This also allows to use extra hardware to evaluate complex signal expression [Gädtke et al., 2007]. However, to do this locally reduces the number of signals needed at runtime, but this requires to put the signal evaluation inside of the abort block. The rewritten Esterel (as well as the *KEP* assembler) is not constructive, because the signal SIG_ABORT is emitted inside the strong abort block



(a) Original Esterel program

(b) Rewritten Esterel

(c) KASM implementation

Figure 4.11.: Implementation of combine

and in Esterel this is only valid for weak abortions. The traditional dismantling of the strong abort replaces the whole abort by a trap, and add a suspend to p to prevent the execution of p when the abort is triggered. Hence, we need to execute an explicit EXIT instruction to detect the abortion, while the abort watcher trigger the abortion without additional overhead. Another possibility is to implement the strong abort with a weak abortion and a suspend, which requires two watchers.

Note that the translation in Figure 4.10a is only correct when p cannot terminate. Otherwise we need an additional signal that is triggered when p terminates and aborts the checking of the signal expression.

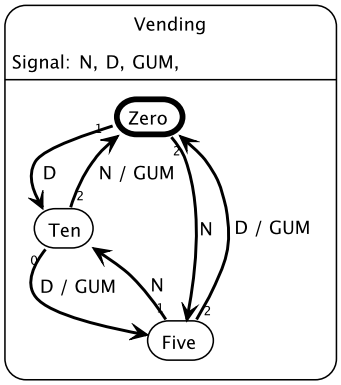
4.4.2. Combine

The combination of multiple emissions of a signal within one tick by a combine function is one of the Esterel features that are so far not implemented by the *KEP* or the strl2kasm. However, the implementation is quite simple, by preceding each emission S(X) with a test for the presence of S, as shown in Figure 4.11.

Note that the rewritten Esterel program is not constructive. However, the *KEP* will simply execute the corresponding code, resulting in the correct behavior. The read write dependencies are handled by the compiler as for all other signals.

4.5. Compiling SyncCharts

The *KEP* was originally designed to execute Esterel programs. However, it can also be used to execute programs written in other synchronous languages. A natural candidate for compilation are SyncCharts, a statechart dialect with a synchronous semantics, which is very similar to Esterel. The usual design flow for SyncCharts is to compile them into Esterel code, which is then compiled further into the target language.



(a) SyncChart

```
1  %           state Zero
2  Zero:       WABORT N, Zero2Five
3              WABORT D, Zero2Ten
4              HALT
5  Zero2Ten:   GOTO Ten
6  Zero2Five:  GOTO Five
7  %           state Five
8  Five:       WABORT D, Five2Zero
9              WABORT N, Five2Ten
10             HALT
11 Five2Ten:   GOTO Ten
12 Five2Zero:  EMIT GUM
13             GOTO Zero
14 %           state Ten
15 Ten:        WABORT N, Ten2Zero
16             WABORT D, Ten2Five
17             HALT
18 Ten2Five:   EMIT GUM
19             GOTO Five
20 Ten2Zero:   EMIT GUM
21             GOTO Zero
22             HALT
```

(b) KEP assembler produced directly from the SyncChart by smakc!

Figure 4.12.: Vending—an example of tightly interconnected SyncChart

Consider the SyncChart in Figure 4.12a. It takes two inputs N (nickel) and D (dime). The state encodes how many cents have been entered: 0 for state Zero, 5 for state Five and 10 for state Ten. A GUM is emitted whenever 15 cents have been entered. We assume that the inputs N and D never occur simultaneously. While such a complete graph is an example for highly non-linear control-flow, it illustrates the problems that can occur when such SyncCharts are transformed into Esterel. Naturally, these difficulties get more pronounced as the number of states and transitions increases. Figure 4.13a shows the Esterel code that was synthesized by E-Studio[1]. Two auxiliary signals, go2Five and go2Ten, are introduced, to indicate that not the initial state Zero, but either state Five or state Ten should be activated.

The problem illustrated here is that a Statechart transition allows arbitrary control changes, akin to a GOTO, and Esterel only allows structured control. More fundamentally, Statecharts are a means to describe reactive behavior [Harel et al., 1990], where it may be perfectly natural to transfer from one system state to an arbitrary other system state. The situation is somewhat different in "classical" computer programs, where a structured control flow is desirable [Dijkstra, 1968]. Actually, it is a common misconception among Statechart novices to treat Statecharts as control flow diagrams, where states

[1] To improve readability of the automatically generated code samples—the Esterel produced by E-Studio as well as the KEP assembler code samples—, comments are added, some labels are changed, and the formatting is polished.

```
1  signal go2Five in
2    loop
3      signal go2Ten in
4        present [go2Five] then
5  %       state Five
6          await
7          case [N] do
8            emit go2Ten
9          case [D] do
10           emit GUM
11         end await
12       else
13 %       state Zero
14         await
15         case [N] do
16           emit go2Five
17         case [D] do
18           emit go2Ten
19         end await
20       end present;
21       present [go2Ten] then
22 %       state Ten
23         await
24         case [N] do
25           emit GUM
26         case [D] do
27           emit GUM;
28           emit go2Five
29         end await
30       end present
31     end signal
32   end loop
33 end signal
```

(a) Esterel code synthesized from the SyncChart by E-Studio

```
1  A0:              SIGNAL go2Ten
2                   PRESENT go2Five, A1
3  %          state Five
4  A3:              PAUSE
5                   PRESENT N, A7
6                   EXIT AC, A3
7  A7:              PRESENT D, A8
8                   EXIT AC_0, A3
9  A8:              GOTO A3
10 AC_0:            EMIT GUM
11                  EXIT AWAIT_CASE, A3
12 AC:              EMIT go2Ten
13                  EXIT AWAIT_CASE, A3
14 AWAIT_CASE: GOTO AWAIT_CASE_0
15 %          state Zero
16 A1:              PAUSE
17                  PRESENT N, A13
18                  EXIT AC_1, A1
19 A13:             PRESENT D, A14
20                  EXIT AC_2, A1
21 A14:             GOTO A1
22 AC_2:            EMIT go2Ten
23                  EXIT AWAIT_CASE_0, A1
24 AC_1:            EMIT go2Five
25                  EXIT AWAIT_CASE_0, A1
26 AWAIT_CASE_0: PRESENT go2Ten, A15
27 %          state Ten
28 A16:             PAUSE
29                  PRESENT N, A20
30                  EXIT AC_3, A16
31 A20:             PRESENT D, A21
32                  EXIT AC_4, A16
33 A21:             GOTO A16
34 AC_4:            EMIT GUM
35                  EMIT go2Five
36                  EXIT A15, A16
37 AC_3:            EMIT GUM
38                  EXIT A15, A16
39 A15:             GOTO A0
```

(b) KEP assembler derived via Esterel

Figure 4.13.: Code generation for the Vending via Esterel

may merely encode the state of the program counter. This "state" is rather short-lived and relates to the *computation* of a behavior, but not the behavior itself. This difference manifests itself also in the synchronous model, where computations are considered to not take any time at all. In contrast, a state of a reactive system should in general be something that the system can reside in for some duration of physical time. Note that one may require a certain structure in Statechart transitions as well. For example, SSMs disallow inter-level transitions, which may simplify compilation and (formal) analysis.

However, within one hierarchy level, transitions are generally unrestricted.

We here consider an alternative path to synthesize code for reactive processors from SyncCharts, which avoids the detour via Esterel, and performs a direct translation instead. This path was investigated for the *KEP* execution platform and a state machine to *KEP* compiler (called smakc!!) was developed, which produces *KEP* assembler directly from SyncCharts. For the Vending example, the code produced by smakc! is shown in Figure 4.12b. As can be seen, this code does not need any additional signals and directly reflects the original structure. Each state is encoded as a HALT, which is enclosed by (weak or strong) ABORTs that trigger outgoing transitions. For example, state Zero corresponds to the HALT in line 4. The transition from Zero to Five, which depends on signal N, is triggered by the WABORT that is initialized in Line 2; if triggered, this transfers control to the label Zero2Five, where a GOTO Five is performed. Note that the nesting order of the ABORTs reflects the transition priorities in the SyncChart. Hence, the generation of *KEP* assembler directly from the SyncChart gives a very simple one-to-one mapping where each assembler instruction can be directly mapped to a state action or a transition. This not only results in compact and efficient code, but also greatly improves code traceability. This supports testing and verification, and in case of safety-critical systems may also aid in certification.

4.5.1. Compiliation Steps

In order to compile a SyncChart into *KEP* assembler, several transformations are applied to the SyncChart, which might alter its structure, or enrich it with additional information. In a last step, the generated SyncChart can be directly written into *KEP* assembler. The compilation is carried out by the following steps:

1. *Complex conditionals extraction:* this transforms complex signal expressions to simple signal tests. This transformation results in an SyncChart with the same behavior, which does not contain conditional expressions.

2. *Dependency analysis:* this detects data and control flow dependencies and adds them to the SyncChart as special transition type.

3. *Cycle detection:* to ensure schedulability, the dependency graph is inspected for cycles.

4. *Scheduling:* the states are scheduled according to the encountered dependencies, the schedule is implemented by assigning appropriate thread priorities.

5. *Code generation:* this generates the target platform's code.

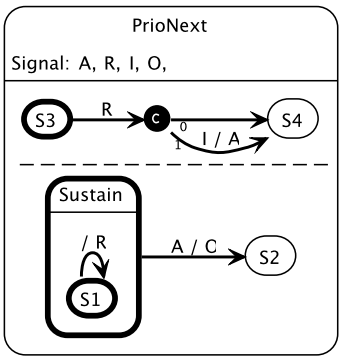
A complete description of the compiler can be found in Falk Starke's diploma thesis [Starke, 2009]. Here we only consider some non-trivial parts of the transformation and compare them to the strl2kasm compiler.

4.5.2. Thread embedding

For the compilation from Esterel to *KEP* assembler, we have to distinguish between prio and prionext: the current priority might be lower than the priority with which we want to restart in the next tick.

```
1  module PrioNext:
2
3  input I;
4  output O;
5
6  signal A, R in
7    weak abort
8      sustain R
9    when immediate A;
10   emit O
11 ||
12   await R;
13   present I then
14     emit A
15   end present
16 end signal
17
18 end module
```

(a) Esterel Source



(b) SyncChart

```
1      PAR 2,A0,1
2      PAR 1,A1,2
3      PARE A2,1
4  A0: WABORTI A,A3
5  A4: EMIT R
6      PRIO 1
7      PRIO 2
8      PAUSE
9      GOTO A4
10 A3: EMIT O
11 A1: AWAIT R
12     PRESENT I,A7
13     EMIT A
14 A7: NOTHING
15 A2: JOIN 0
16     HALT
```

(c) KEP assembler generated from Esterel

```
1      PAR 2, A1, 1
2      PAR 3, A2, 2
3      PARE A6, 0
4  A1: WABORTI A, A5
5      PAR 4, A3, 3
6      PARE A4, 0
7  A3: SUSTAIN R     % S1
8  A4: JOIN 2
9      HALT
10 A5: EMIT O
11 A2: AWAIT R       % S3
12     PRESENT I, A7
13     EMIT A
14 A7: NOTHING
15 A6: JOIN 4
16     HALT
```

(d) KEP assembler generated by smakc!

Figure 4.14.: Handling of prionext by strl2kasm and smakc!

See for example the small Esterel program in Figure 4.14. The logical control flow is that we first have to emit R in the first thread. We check for R in the second thread and emit A when I is present. Now we have to execute the first thread again to check for the abortion. In the *KEP* assembler code generated by strl2kasm, this is realized by

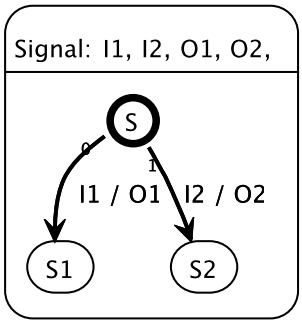
executing the EMIT R with priority two. The PAUSE that is generated from the sustain is executed with the lower priority one. In between, the second thread is executed. The code generation from SyncChart takes a different approach: the thread inside the SUSTAIN state is itself embedded in an extra thread. The abortion is now triggered by the execution of the JOIN statement in line 8. While this requires the usage of one extra thread and more instructions for the setup, the number of instructions that are executed within one tick is reduced. In particular, we can use the *KEP* instruction SUSTAIN, because there is no need to change the priority within the sustain. For the *KEP* assembler generated from Esterel, the following 11 instructions are executed in the second tick, if I is absent: L8, L9, L5, L6, L11, L12, L14, L7, L8, L15, L16. For the code generated by smakc!, only the following 9 instructions are executed: L7, L12, L13, L15, L8, L10, L11, L16, L17. Unfortunately, the compiler does not yet produce this optimal code for the example, therefore it was adjusted by hand, *i. e.*, to detect the SUSTAIN or to remove some unnecessary jumps. This can also be implemented in the compiler.

4.5.3. PRIO instructions

The insertion of PRIO instructions differs from the strl2kasm approach. There a priority for each instruction is computed, for SyncCharts, we can compute one priority for each state, simplifying the compilation. The parent state has higher priority than its child states: outgoing strong abortions are triggered by the execution of the JOIN of the parent state, while outgoing weak abortions are triggered by the execution of time delimiting instructions in the child states. Note that this is not possible for other reactive processors, like the StarPro or for SyncCharts in C, because there explicit checks for abortions must inserted, while on the *KEP* the watchers are checked in parallel, and only need the execution of an arbitrary instructions inside the watched range to trigger the watcher. But having one priority per state leads to the question, where to put the priorities. They can either be put in the start-up code, or they have to be added to each incoming transition from coming from a state with a (potentially) different priority. The former will lead to fewer instructions, but might lead to longer executions, since the PRIO instruction is also executed when we enter the state from another state with the same priority.

4.5.4. Weak abortion

Consider a state with two outgoing weak transitions. A naive way to implement this in *KEP* assembler is shown in Figure 4.15. The semantics of weak abortion assures that the inner part may be executed until a pause is reached, hence for the compilation from Esterel, the weak abortion with the highest priority is mapped to the watcher with the highest priority. However, in contrast to an abortion in Esterel, a transition in a SyncChart can jump outside of the scope of an outer transition. Hence in the implementation on the *KEP*, the order of the watchers for weak abortions needs to be inverted: the transition with the highest priority is mapped to the watcher with the lowest priority, which is executed first. For this to work, it is important that the



(a) A simple SyncCharts

```
1     WABORT I1, A1
2     WABORT I2, A2
3     HALT
4  A2: EMIT O2
5     GOTO S2
6  A1: EMIT O1
7     GOTO S1
```

(b) Naive, wrong implementation

```
1     WABORT I2, A2
2     WABORT I1, A1
3     HALT
4  A2: EMIT O2
5     GOTO S2
6  A1: EMIT O1
7     GOTO S1
```

(c) Correct implementation

Figure 4.15.: Implementing weak abortion

KEP recomputes the fact that an abortion is triggered in each instruction cycle. For the execution of *KEP* assembler derived from Esterel, this does not matter, since the nesting of abortions assures that control will always reach the handler of the outer watcher, either by abortion or by normal control flow.

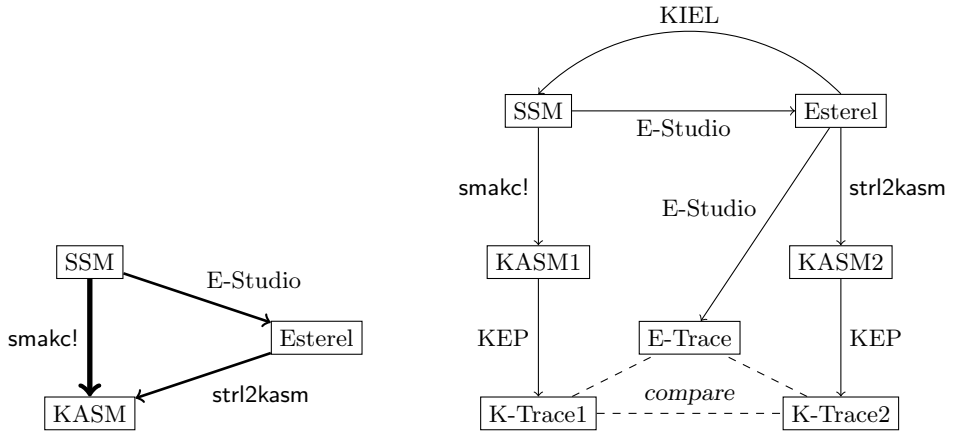
Another possibility is to explicitly check for active transitions with higher priority, what is possible, when executing on a virtual machine. This approach must also be used if a normal termination and a weak abortion can happen simultaneously. However, in this case we only have to insert explicit code to check for outer transitions for *one* normal termination. This is also done for the compilation from SyncCharts to Esterel with Esterel Studio.

4.5.5. Experimental Results

The KReP Evalbench (see Chapter 7) to verify the *KEP* assembler produced by smakc! and to measure its reaction times. Experiments were carried out on several SyncChart examples. The SyncCharts were created in E-Studio, which was also used to compile them into Esterel code and to generate traces (E-traces) that covered all SyncChart transitions. The SyncChart and the generated Esterel program were compiled into *KEP* assembler. The two *KEP* programs could then be directly compared to each other in terms of size and reaction times, see Figure 4.16b.

For the comparison, the programs were executed on the *KEP* with the E-traces as inputs, generating output traces (K-Traces). The KReP Evalbench compared the generated outputs to the outputs in the E-traces, thereby confirming the correctness of the compilation. It also recorded the minimum, maximum and average reaction times of the generated code, which were compared to the reaction times for the compilation via Esterel. Hence, Esterel Studio was used as a reference point both for validation (correctness of smakc!) and for benchmarking (competitiveness of smakc! with code synthesized via Esterel Studio and strl2kasm).

The object code size, measured in 40-bit instruction words, for compilation by the two



(a) Different compilation schemes from SSMs to KEP assembler (KASM). The classical synthesis path involves Esterel as intermediate step

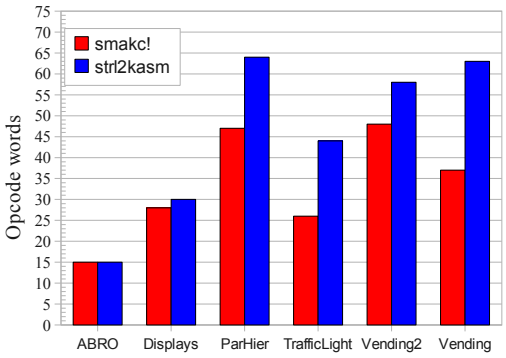
(b) Each model is compiled either directly or via Esterel. To validate the compilation, the executions on the *KEP* are compared to reference traces generated by E-Studio. The reaction times of both executions are compared to each other.

Figure 4.16.: Compilation paths and validation of the compiler

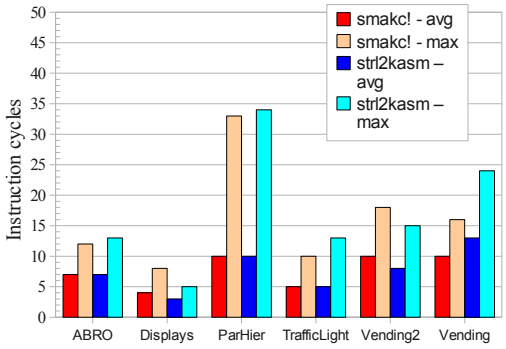
compilers is listed in Figure 4.17a. The smakc! compiler generates slightly smaller programs than strl2kasm. Still the code could be further optimized by removing superfluous thread embeddings and GOTOs, this requires a slightly more sophisticated dependency and control flow analysis. The smakc! compiler is particular at an advantage when the levels of nested hierarchy and transition count per state grow. A possible explanation is that the transformation to Esterel described by André and implemented in E-Studio has difficulties to efficiently encode exactly these two cases.

Average and maximum reaction times of the code are shown in Figure 4.17b. Again, the results are nearly identical, but usually slightly better for the direct compilation, in particular in the automata-like examples. A difficult encoding using a lot of Esterel instructions also accounts for more *KEP* assembler instructions, increasing the reaction times of the resulting code. The reactions of the code generated by smakc! can sometimes be slower, in particular due to the thread embedding, where additional JOIN statements are executed in each tick.

The strl2kasm compiler uses a more involved dependency analysis than smakc!. Incorporating this into the smakc! should help to avoid many unnecessary embeddings of threads in macrostates, and hence improve both code size and reaction times.



(a) Comparison of code size.



(b) Comparison of reaction times, per logical tick.

Figure 4.17.: Comparison between smakc! and strl2kasm.

5. The Kiel Lustre Processor (KLP)

Contents

The *KLP* is a reactive processor based on the synchronous language Lustre. Since the control structures of Lustre are much simpler than the one of Esterel, also the generation of good code, which is both compact and fast, is much simpler. Hence Lustre code can be efficiently executed on standard processors. Therefore the need for a reactive processor for Lustre is not as obvious as for Esterel. However, the implementation of Lustre clocks by conditionals seems to be unsatisfactory, and recent approaches on the compilation [Biernacki et al., 2008] show that there is still room for improvements. Furthermore the augmentation of the language by automata adds to additional complexity of efficient code generation. The main aim of the *KLP* is to examine whether a special hardware for executing Lustre and Scade leads to significant performance improvements over the execution on a standard processor.

The main idea of the *KLP* is to have a direct representation of the Lustre equations in hardware. This leads to four differences to common processor design:

1. *Direct support for clocks*: The Lustre clocks are used directly to determine whether a flow should be computed or not. No further computational overhead is needed.

2. *Multiple PCs*: Since each register has one PC, we always have multiple program counters available, which leads to a very fine grained thread model.

3. *Easy access to previous values*: In Lustre programs, one often references to the previous value of a variable. Therefore, it is directly stored in hardware. Another possibility would be to compute the value only for that flows, whose previous value is actually used.

4. *Parallel execution*: The inherent parallelism of Lustre is explored by the dynamic decision which registers shall be executed. This allows to execute instructions that do not depend on each other in parallel.

Due to the parallel execution, there are some similarities to the distributed execution of Lustre. There are basically three different reasons why one might want to execute a Lustre program in a distributed fashion.

1. *Redundancy*: The platform on which we want to execute the Lustre program might be distributed due to safety reasons. This can mean that the same program is executed on multiple processors. But we can also assume that each hardware unit executes its own program, while we want to be sure that at most one program crashes due to a single hardware failure.

2. *Distributed IO*: If we have a platform with multiple IOs, connected to different controllers, it is useful to run the part of the system that directly uses the IO on the controller that is connected to it.

3. *Performance*: Another reason is pure performance to execute complex control system on slow hardware. Since Lustre programs often consist of multiple, only loosely coupled parts, it might be better, both in terms of costs and resource usage, to execute the programs on multiple, slow processors, than to use one fast processor.

One might wonder, why we do not specify multiple Lustre programs in the first place. The reason is simple: maintainability and flexibility. A monolithic Lustre program frees the designer from thinking about low level details of the distribution and it allows the execution on multiple, different platforms.

Girault [2005] gives an overview of different methods in distributing synchronous dataflow programs. Here, the main method is to duplicate the code for each hardware component, and thereafter to remove on each component the parts that are not necessary for this particular component. Of course, the benefit of the distribution is limited by the message passing time. Since the *KLP* is still a single processor, the benefit of the parallel execution is purely performance. Still, the performance should be achieved in a way that does not sacrifice predictability.

A first approach of the *KLP* [Traulsen, 2007] was to execute Lustre in a pipelined fashion. To achieve this, each Lustre program was split into multiple nodes, with minimal data-dependencies; each node is executed on its own core. Since data computed on a

different core will be visible with one tick delay, we need to store inputs locally to ensure that all inputs are valid in the same tick. One advantage of this approach is that it can run on any multi-core architecture with sufficiently fast communication, we do not need a special reactive processor. However, this compilation makes no use of the special features of Lustre, such as clocks.

The current version of the *KLP* directly reflects the dataflow nature of Lustre programs. Each register of the *KLP* corresponds to one flow in Lustre; it contains all values that are necessary to define one flow: a current and a previous value, its clock and a program counter. The *KLP* executes independent equations in parallel and uses the clocks in Lustre, to detect which parts of a program need to be executed in one tick.

In the next section we take a closer look on the architecture of the *KLP*, before we consider the compilation from Lustre and Scade in Section 5.2. In Section 5.3 we show experimental results for the resource usage and the execution times. In Section 5.4 the experiences with the hardware design of the *KLP* using Esterel v7 is evaluated, before comparing the *KEP* and the *KLP* in Section 5.5

5.1. Architecture

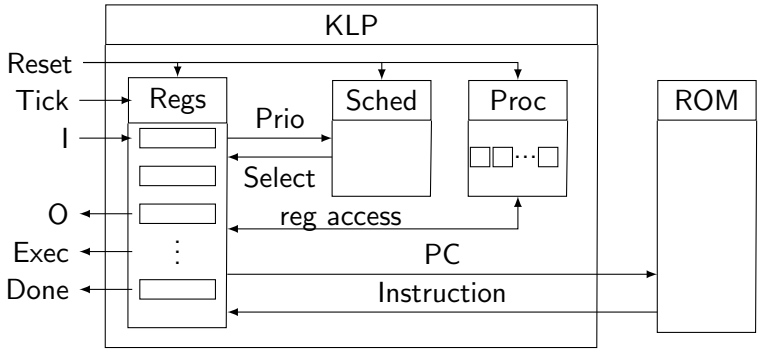


Figure 5.1.: Overview of the KLP

The architecture consists of three main blocks (see Figure 5.1): the *register file* (Regs) stores the relevant information for each equation. The *processing unit*(Proc) executes arithmetical and logical operations, and the *scheduler*(Sched) decides which equations are to be executed next.

The external interface consists of the following inputs: a Tick signal which triggers the execution of a global tick, a set of integer input signals I and instructions coming from the instruction ROM. The outputs are the Done signal, to indicate that a tick has finished, a set of integer output signals O, and program counter values PC to request instructions from the ROM. The Exec output indicates which instructions were executed in the current tick.

The *KLP* has two different options for the scheduling: it is either performed dynamically, or via statically introduced priorities. Depending on this option, the register block,

the scheduler, and the interface between them slightly differs. Also the direct support for clocks can be deactivated. The *KLP* is further scalable by setting the number of registers (N_{REG}), the number of IO ports (N_{IO}), and the number of processing units (N_{ALU}).

5.1.1. Building blocks

Register Files

The *register file* stores all runtime information. For each equation this is:

- the current value,
- the previous value,
- the address of the next instructions which is needed to compute this register,
- a register id of another register, which holds the clock of the expression, and
- a *done* flag to indicate whether the execution has terminated for the current tick.

For the priority based scheduling, each register also holds a priority. Each register that is not done for the current tick sends its priority to the scheduler. Additionally, each register has some basic control logic to detect whether it is done, because the register that holds its clock is done and its value is zero.

The register file also contains some control logic: whenever the input Tick is set to true, it reads all inputs, overwrites the previous value with the current value for all registers, and resets the done-flags of all active registers, *i. e.*, registers with a program counter different to zero. Additional Tick inputs to trigger new computations are ignored until the computation of the current tick is finished.

Whenever there is a register whose done-flag is false, this information is send to the scheduler. From the scheduler it receives the information which register shall be executed. Now it reads the instruction for this register from the instruction ROM and sends the instruction to the processing unit and waits for the results. When all registers are done, the register file signals this to the environment via the Done output and writes the output values.

For the boot process, register 0 is set to ready. This triggers the execution of instruction 0, where the boot code to initialize the other registers is expected. At the end of the initialization, the program counter of register 0 is reset to 0, it will not be executed from that on. Hence register 0 can not be used as a normal register, but it can be connected to some input.

The register file communicates with the environment, *e. g.*, by reading the inputs or instructions and by writing outputs and program counters to the ROM. It sends the active registers to the scheduler and gets for each processing unit the mapped register. The communication with the processing units is more complex: The processing unit sends an identifier to the register file, which sends back the corresponding value. The

processing units can also write register values, where the behavior is undefined if multiple processing units write to the same register. The processing units can also write all other register values, like the clock, the program counter or the done flag.

Dynamic Scheduling For the dynamic scheduling, each register detects itself whether it is ready to execute. This is based on the next instruction and the done flag of all other registers. It is similar to traditional out-of-order execution, but due to the fixed steps in Lustre, solving data dependencies is much simpler. For each register we know precisely whether the computation of its value has finished for the current tick. On the other hand, this will always check all registers, as for a Lustre program there is no order of instructions; therefore it will utilize the maximal degree of parallelism in each instruction cycle. This might still not be the maximal possible parallelism for the whole tick. If the program contains a chain of dependent register and some independent ones, we should avoid to first execute the independent one, before starting to execute the sequence.

To detect on which other register a register depends, all registers need to prefetch their next instruction. Note that each register only needs to know the done flag of the other registers. Due to the regular instruction set of the *KLP*, it is very simple to extract the dependencies from an instruction.

Each ready register sets a ready flag that it sends to the scheduler. Since we require the dependency graph to be acyclic, there is at least one register ready whenever there is a register that is not done. This is even true for all *constructive* Lustre programs, *e. g.*, all programs without cyclic data dependencies at runtime, like the following one:

```
X = if I then Y else 0;
Y = if I then 1 else X;
```

The cyclic dependency will always be resolved at runtime by the input I, hence the execution on the *KLP* will never deadlock. Traditional Lustre compilers reject programs that have static cyclic data dependencies, even if the cycle is always resolved at runtime.

Priority Based Scheduling For the priority based scheduling, the priority for each register is stored. The priorities are set by the PRIO instruction. Two registers may be executed in parallel when they have the same priority. The priorities are computed by the compiler, based on the syntactic dependencies. While this approach may miss potential parallel execution, it needs less control-hardware. Another benefit is that it allows non-local dependencies, *i. e.*, dependencies that are not completely determined by the next instruction. Such dependencies are introduced by Scade automata (see Section 5.2.3), where each outgoing transition adds a dependency to each computation inside a state. Each register, which is not done, sends its current priority to the scheduler.

Processing Units

The processing unit is responsible for the execution of one instruction. For the *KLP* instruction set, this normally means to decode the current instruction into opcode and

Opcode		Arg_0	Arg_1	Arg_2
Deps	Fct			
0	X	I/T	I	I
1	X	T	V	V
2	X	I/T	V	I
3	X	T	C	C
4	X	I/T	C	I
5	X	T	V	I
0 3	4 7	8 15	16 23	24 31

Figure 5.2.: Structure of an 32 Bit *KLP* instruction. I is an immediate value and T a target register. V and C the ids of read valued and clock registers, on which the computation depends: the dependecy is encoded by the first 4 bits.

arguments and send the read values to the contained ALU. The execution of all instructions takes one clock cycle. The arithmetic operations are synthesized from Esterel except for division, for which no automatic synthesis is allowed, therefore a simple implementation of hardware integer division was implemented manually in Esterel.

Scheduler

At run-time, the scheduler maps ready registers at each tick to free processing units. For the dynamic scheduling, the scheduler will simply take the first available registers. Therefore, the actual scheduling and also the utilization are affected by the ordering of the equations. So the compiler should order the equations by the number of equations that depend on them.

For the priority based scheduling, the scheduler first computes the maximal priority of the currently active registers and only maps registers with this priority to the processing units.

5.1.2. Instruction Set

Beside the usual arithmetical and logical instructions, the *KLP* contains a SETCLK and SETPC instruction to initialize a register by setting the clock and the initial program counter. The INPUT and OUTPUT instructions map the register to inputs and outputs. At the start of a tick, the inputs are copied to input registers and at the end of the tick, the values of the output registers are copied to the output. For each reference to a register, the first bit indicates whether the current or the previous value is used. The DONE instruction marks the current register as finished for the current tick. It has the program counter from which the register shall start in the next tick as an argument. This argument can be omitted in the assembler, the control is transfered to the next instruction in this case. The DONE instruction is similar to the PAUSE instruction in *SyncCharts in C* [von Hanxleden, 2009] or the gotopause in *Esterel + goto* [Tardieu and Edwards, 2007].

The *KLP* has a regular instruction set: each instruction has 32 bits, where the first byte contains the opcode, as shown in Figure 5.1.2. The first 4 bits of the opcode encode the data-dependencies of the instructions, this is used for the automatic scheduling. The second 4 bits encode the ALU-function, in case the instruction uses the ALU. An overview of the instructions is shown in Figure 5.3. The instruction set distinguishes between clock and value registers, where clock registers only contain a boolean value, while value register contain an 32 bit integer. Therefore there exist 4 instructions for register to register movements. The current implementation of the *KLP* only contains valued registers and maps clock registers to valued ones.

5.2. Compilation

The compilation from Lustre into *KLP* assembler consists of three steps (see Figure 5.4). First, the lus2ec tool from the Verimag Lustre compiler[1] is used to generate expanded code (ec), *i. e.*, all nodes and tuples are expanded and pre operators are propagated to variables. This also checks that the programs are well-formed, *i. e.*, every variable is defined exactly once, there are no cycles in the dependency graph, and only variables that run on the same clock are combined.

In the second step, the Lustre program is further simplified by restricting the use of clock operators. This results in Clocked Equations (CEQ), which are mapped to the *KLP* instruction set (Klp asm) in the third step. The first two steps are source to source transformations that still yield valid Lustre code. This allows easy validation of the correctness by using existing Lustre tools to compare the Lustre source file with the generated code. For the straight-forward compilation, one register is needed for each clocked equation. However, by introducing static schedules the number of registers can be reduced, similar to the standard compilation of Lustre.

Scade models can be directly translated into clocked equations with automata. In Scade, the usual way to use clocks is via activation conditions, which are simpler to handle than the arbitrary combination of when and current in Lustre. And in Scade each operator defines a new flow, hence the compiler must combine flows to reduce the number of used registers, instead of splitting complex flows, as it needs to be done for Lustre. Consequently compiling the dataflow-part of Scade is simpler than compiling Lustre.

5.2.1. Clocked Equations

Clocked equations are Lustre programs where clock operators may only occur at some special positions.

All equations have the form: x=current((i → e) when C), where i and e are arbitrary expressions that do not contain any clock operators except for pre. The C is either the name of a boolean variable, or true, *e. g.*, the expression is running on the base clock. e and i may not contain nested pre operators, therefore we might have to introduce

[1] http://www-verimag.imag.fr/SYNCHRONE/

Instruction	Opcode	Meaning
SETCLK *reg*, *clock*	0x01	initialize clock register
SETPC *reg*, *pc*	0x02	initialize valued register
DONE [*PC*]	0x03	set done flag and pc for next tick
INPUT *id*, *reg*	0x04	map register to input id
OUTPUT *id*, *reg*	0x07	map register to output id
LOCAL *id*, *reg*		declare local register, only used by the assembler
PRIO *reg*, *p*	0x09	set priority for register *reg* to *p*
ADD *regR*, *reg*1, *reg*2	0x10	$regR \leftarrow Reg1 + reg2$
SUB *regR*, *reg*1, *reg*2	0x11	$regR \leftarrow Reg1 - reg2$
MUL *regR*, *reg*1, *reg*2	0x12	$regR \leftarrow Reg1 * reg2$
DIV *regR*, *reg*1, *reg*2	0x13	$regR \leftarrow Reg1/reg2$
IADD *regR*, *reg*, *val*	0x20	$regR \leftarrow reg + val$
ISUB *regR*, *reg*, *val*	0x21	$regR \leftarrow reg - val$
IMUL *regR*, *reg*, *val*	0x22	$regR \leftarrow reg * val$
IDIV *regR*, *reg*, *val*	0x23	$regR \leftarrow reg/val$
AND *regR*, *reg*1, *reg*	0x3A	logical and
OR *regR*, *reg*1, *reg*	0x3A	logical or
XOR *regR*, *reg*1, *reg*	0x3A	logical exclusive or
IAND *regR*, *reg*, *val*	0x3A	logical imediate and
IOR *regR*, *reg*, *val*	0x3A	logical immediate or
IXOR *regR*, *reg*, *val*	0x3A	logical immediate exclusive or
LT *regR*, *reg*1, *reg*2	0x14	$regR \leftarrow Reg1 < reg2$
LE *regR*, *reg*1, *reg*2	0x15	$regR \leftarrow Reg1 \leq reg2$
EQ *regR*, *reg*1, *reg*2	0x16	$regR \leftarrow Reg1 = reg2$
GE *regR*, *reg*1, *reg*2	0x17	$regR \leftarrow Reg1 \geq reg2$
GT *regR*, *reg*1, *reg*2	0x18	$regR \leftarrow Reg1 > reg2$
NEQ *regR*, *reg*1, *reg*2	0x19	$regR \leftarrow Reg1 <> reg2$
ILT *regR*, *reg*, *val*	0x24	$regR \leftarrow reg1 < val$
ILE *regR*, *reg*, *val*	0x25	$regR \leftarrow reg1 \leq val$
IEQ *regR*, *reg*, *val*	0x26	$regR \leftarrow reg1 = val$
IGE *regR*, *reg*, *val*	0x27	$regR \leftarrow reg1 \geq val$
IGT *regR*, *reg*, *val*	0x28	$regR \leftarrow reg1 > val$
INEQ *regR*, *reg*, *val*	0x29	$regR \leftarrow reg <> val$
JMP *pc*	0x0A	unconditional jump
JT *reg*, *pc*	0x44	jump when true
JF *reg*, *pc*	0x45	jump when false
JZ *reg*, *pc*	0x2A	jump when zero
JNZ *reg*, *pc*	0x2B	jump when not zero
INT *regT*, *regS*	0x50	$regT \leftarrow regS$
BOOL *regT*, *regS*	0x2C	$regT \leftarrow regS$
VVMOV *regT*, *regS*	0x2D	$regT \leftarrow regS$
IVMOV *regT*, *val*	0x0B	$regT \leftarrow val$
CCMOV *regT*, *regS*	0x41	$regT \leftarrow regS$
ICMOV *regT*, *val*	0x0C	$regT \leftarrow val$

Figure 5.3.: Overview of *KLP* instructions. All arguments like registers (*reg*), immediate values (*val*), clocks (*clock*), io identifieres (*id*), priorititeres are 1 Byte long, only program counters (*pc*) are 2 Byte long.

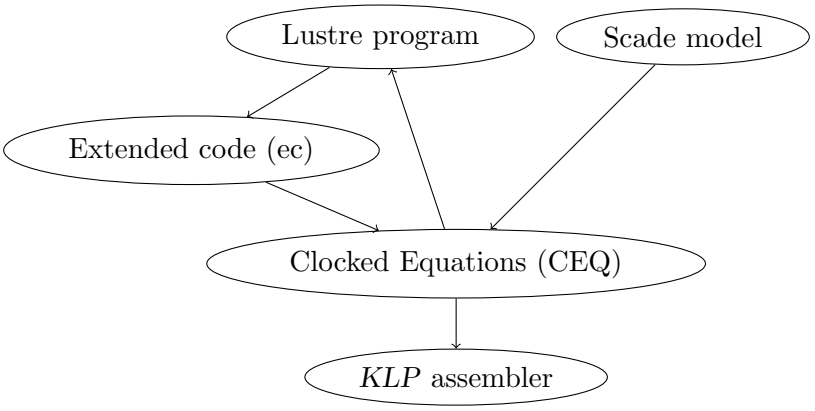


Figure 5.4.: Compilation paths to the KLP assembler

```
1 X = I when C;
2 A = current (X→ (pre(X) + X));
3 B = I→ (pre(current(X)) + I);
```

(a) Combinations of pre and current

```
1 X = current((I) when C);
2 A = current((X→(pre(X) + X)) when C);
3 B = I→(pre(pre_1) + I);
4 pre_1 = current((X) when C);
```

(b) Translation into clocked equations

I	0	1	2	3	4	5	6	7
C	1	1	0	1	0	1	0	1
X	0	1		3		5		7
current(X)	0	1	1	3	3	5	5	7
pre(X)	⊥	0		1		3		5
current(pre(X))	⊥	0	0	1	1	3	3	5
pre(current(x))	⊥	0	1	1	3	3	5	5
A	0	1	1	4	4	8	8	12
B	0	1	2	4	7	8	11	12

(c) Execution trace

Figure 5.5.: Access to previous value with and without current.

additional variables. The initialization is optional, if it is missing, the evaluation of e starts in the first tick. Also the clock is optional, it can be omitted if the equation runs on the base clock.

Note that the equation is initialized when the clock is true for the first time. This is different to x=i → current e when C, where the stream is always defined in the first instance, but is then undefined until the clock is true for the first time.

Figure 5.5 shows an example for the correspondence between Lustre and clocked equations. The main differences are the introduction of one additional register for the previous value and that all variables are running on the base clock. Another option would haven been to take x=((i → e) when C) as the base form for an equation, and use current as an operator inside e. For the access of an actual value, this does not make any change, since the current operator will not affect it. Only the access of previous values is changed (see Figure 5.5c); we need an extra register to store the value of pre(X). If we choose current as a regular operator, we need an extra register to store pre(current(X)). As Figure 5.5, shows, pre(current(X)) and current(pre(X)) are not equivalent. The current(pre(X)) was

chosen as the default, because this seems to be more common in Lustre programs. In particular, this restriction of clocked equations is similar to the activation condition in Scade, except that the activation condition takes an additional default value when the clock is absent. For clocked equation this default value is fixed to the previous value of the output.

The behavior of a clocked equation directly corresponds to the execution of the *KLP*: in each tick, all previous values are overwritten. This corresponds to the fact that the current operator was chosen as a basic instruction in clocked equations, requiring all variables to run on the base clock. If we would only overwrite the previous value of registers that were active in the last tick, this would correspond to handling current as a normal operator.

Automata

For the compiler from Scade to *KLP*-assembler, we extended the clocked equations by automata. The main program contains both equations and states, where each state may contain further equations and states. Hence the nesting is restricted with respect to Scade, where state can also be contained in equations. The compiler only distinguishes weak and strong abortions, normal terminations are not taken into account. Currently the compiler does not support the arbitrary nesting of automata and dataflow that is allowed in Scade. Automata may contain other automata and equations, but equations may not contain automata. This could be done by a preprocessing step, where the automata are extracted from the equations and executed in parallel.

5.2.2. Compiling Lustre

Compiling Lustre into Clocked Equations

We assume that all tuples and nodes are expanded and propagate all pre operators, so that they directly access variables, and not arbitrary expressions. The extended code generated by the lus2ec tool fulfills these requirements. The compilation consists of the following steps:

1. *Infer clocks*: assign to each sub-expression the clock on which it is executed. This also checks whether only valid clock operations are performed, *e. g.*, only variables that run on the same clock are combined, and there is no current operation on a variable that already runs on the base-clock.

2. *Lift clocks*: In order to reduce the number of clock-occurrences, clocks are lifted to the top if possible, *e. g.*, a when C op b when C is combined to (a op b) when C.

3. *Declock*: introduce auxiliary variables for nested clock operations.

4. *Propagate constants*: in Lustre, even constants run on a clock. Therefore, the declock operation might introduce some auxiliary equations, which simply compute the value of constant on a given clock. These are removed again in this step.

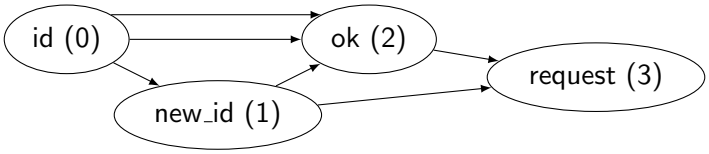


Figure 5.6.: Dependency graph for the check node. The nodes are annotated with the assigned priorities.

5. *Priority assignment*: set the priorities depending on the data dependencies. This is done by performing a topological sort on the dependency graph.

Priorities The priorities are simply computed due to the data-dependencies per equation, this is done by a simple traversal of the dependency graph, Figure 5.6 shows the dependency graph with the assigned priorities for the check node in the gate example from Figure 3.6 on Page 32, as generated by the compiler. Here the compiler assigns the highest priority to the equations that compute new_id, the next priority are assigned to ok, because it has new_id as its clock. The lowest priority is assigned to request, because it depends on both other equations. This is reflected in the generated *KLP* assembler in Figure 5.8 by setting priorities 1, 2, and 3, respectively. Note that the lowest number indicates the highest priority; hence priority 0 is always the maximal priority.

Compiling Clocked Equations into KLP assembler

Each *KLP* program starts with a setup code, which initializes the registers, by setting clocks and program counters. This code starts at address 0 and is executed when the *KLP* is powered on. In the setup phase, only register 0 is active. The real code is not executed before the next tick, when the initial values are written according to the initializations in the Lustre program.

Translating clocked equations to *KLP* assembler is straight-forward. The compiler can generate code that uses the direct support for clocks in the *KLP*, or the clocks can be checked explicitly in the code. The implementation for an equation x=current((i →e when C)) is shown in Figure 5.7.

In the setup phase (Lines 1+2), the program counter is set to the code that executes i and the clock is set to the register for C. For the priority based scheduling, also the priority of the register is set (Line 3). After execution the code for i (Line 6), control jumps to the code for e for the next tick (Line 7). After the execution of e (Line 9), it is restarted in the next tick (Line 10).

Figure 5.8 shows the *KLP* assembler for the check node. The registers are initialized in Line 1 to Line 16. In Line 9 the clock for ok is set to new_id. This is the only clock that is set, because all other flows in the Lustre program run on the base clock. The code for the new_id flow is in Line 20–25. Line 21 performs the initialization of the flow in the first tick, for the next tick, the control is set to Line 24 by the DONE statement in Line 21. From that on, the new_id is computed in Line 24, where the DONE in Line 25

```
1  SETPC reg_X L_x_init
2  SETCLK reg_X reg_C
3  PRIO reg_X p
4  ...
5  L_X_init :
6       ...      // Code for i
7       DONE L_X_run
8  L_X_run:
9       ...      // Code for e
10      DONE L_X_run
```

(a) Using HW support for clocks

```
1  SETPC reg_X L_X
2  PRIO reg_X p
3  ...
4  L_X:
5       JT C L_X_init
6       DONE L_X
7  L_X_init :
8       ...      // Code for i
9       DONE L_X_run
10 L_X_run:
11      JF C L_X_done
12      ...      // Code for e
13 L_X_done:
14      DONE L_X_run
```

(b) Check clocks in SW

Figure 5.7.: Translating a clocked equation x=current((i →e when C)) into *KLP* assembler

executes this Line in each tick.

5.2.3. Compiling Scade

Automata are an important feature of the Scade language. Having good support for automata should give a significant improvement of the performance, even though automata can always be compiled to standard dataflow. Instead of transversing all outgoing transitions within a tick, they are declared at initialization time, all checked in parallel when the state is active. In contrast to the handling of SyncCharts on the *KEP*, it is not easy to extend the *KLP* by watchers to implement the preemption, because watchers, which watch a specific code range and can abort multiple threads, are hard to combine with the multiple processing units of the *KLP*. As mentioned before, Scade can be compiled into dataflow equations similar to Lustre, which then can be mapped to clocked equations, but a a direct compilation, should result in more compact code. One problem with the execution of Scade programs on the *KLP* are non-local dependencies, which are introduced by automata. Before executing the current flow, we have to check all strong abortions that might kill it. After the execution took place, all weak abortions must be executed. Three different approaches exist for the parallel execution of hierarchical automata:

1. The necessary control can be replicated as it is done by the distributed execution of Lustre programs [Girault and Nicollin, 2003]. Of course, this implies that the same code is executed multiple times.

2. Another possibility is to insert instructions into each parallel branch that explicitly request information of the global state from some master branch, which is for example done by the Emperor [Yoong et al., 2006].

```
1  INPUT     id
2
3  LOCAL     new_id
4  SETPC     new_id   L_new_id
5  PRIO      new_id   1
6
7  LOCAL     ok
8  SETPC     ok       L_ok
9  SETCLK    ok       new_id
10 PRIO      ok       2
11
12 OUTPUT    request
13 SETPC     request  L_request
14 PRIO      request  3
15
16 DONE
17
18
19
```

```
20 L_new_id:
21   ICMOV new_id 1
22   DONE L_new_id_run
23 L_new_id_run:
24   NEQ new_id   id       pre(id)
25   DONE L_new_id_run
26
27 L_ok:
28 L_ok_run:
29   IVMOV ok_0  5
30   IDIV  ok_1  id     5
31   MUL   ok    ok_0   ok_1
32   EQ    ok    ok     id
33   DONE L_ok_run
34
35 L_request:
36 L_request_run:
37   AND  request new_id  ok
38   DONE L_request_run
```

Figure 5.8.: KLP assembler for the check node

3. The third possibility is to let the master branch execute the control parts, and only distribute the data-parts that can be easily parallelized. This is the approach we consider here. For each automaton or macrostate one control thread is generated. Per default it runs with higher priority than the substates. At each tick it first checks for strong abortions, then lowers its own priority to let the inner equations execute. Thereafter, it raises its priority again to assure that strong abortions are checked with high priority in the next thread, and checks for weak abortions. If a transition is triggered, it executes code to reconfigure all equations that are defined in the source and target state. Due to the semantics of SyncCharts in Scade, each state is executed at most once in each tick, and the only case when a state is entered and left in the same tick is if it is activated by a strong abortion and left by a weak abortion. If a weak abortion is taken, a DONE statement is executed to stop the controller after the abortion. For the strong abortion, the weak abortions of the target state are still to be checked.

Figure 5.9 shows a simple automaton in Scade. It takes two inputs: an integer I and a boolean X. Its only output is the integer O. The local integer variable c is initialized to 0. On the left side, X is converted into a signal. The variable c is incremented each tick in the initial state A and decremented in state B. Control is transferred from A to B when c equals 10, and from B to A when c is less or equal to 0. These transitions are weak-delayed, indicated by the dot at the arrowhead. Therefore, the equation inside the state is executed one last time in the tick where the transition is triggered, and the execution of the target state starts in the next tick. Control can also be transfered from A to B by the input X. This triggers a strong abortion (dot at the arrow-tail), which will immediately transfer control to state B without executing state A first.

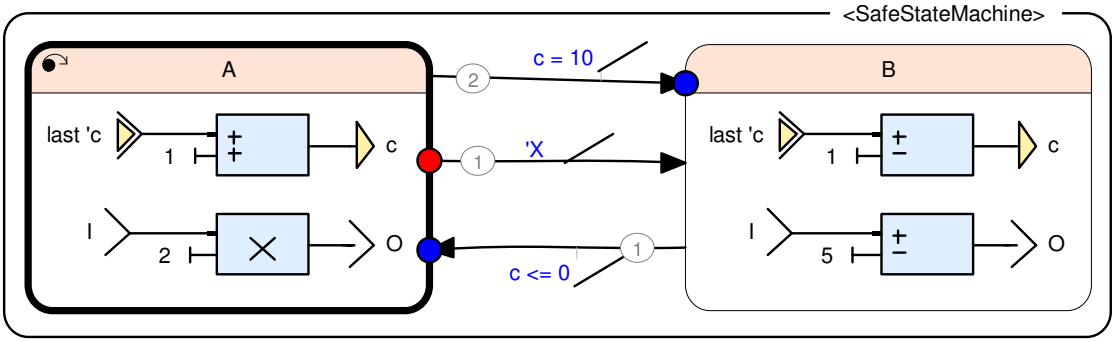


Figure 5.9.: A simple Scade automaton

```
1       INPUT I
2       INPUT X
3       LOCAL CTRL
4       SETPC CTRL L_CTRL_A
5       PRIO CTRL 1
6       OUTPUT C
7       SETPC C L_C_A
8       PRIO C 2
9       OUTPUT O
10      SETPC O L_O_A
11      PRIO O 2
12
13  // Controller for state A
14  L_CTRL_A:
15      JT X A_2_B_S    // Check strong abort
16  L_CTRL_AW:
17      PRIO CTRL 3    // Execute state
18      PRIO CTRL 1
19      IEQ T C 10      // Check weak abort
20      JT T A_2_B_W
21      DONE L_CTRL_A // Resume in next tick
22  // strong abort from A to B
23  A_2_B_S:
24      SETPC C L_C_B
25      SETPC O L_O_B
26      GOTO L_CTRL_B // Check weak
27                    //abortion in B
28  // weak abort from A to B
29  A_2_B_W:
30      SETPC C L_C_B
31      SETPC O L_O_B
32      DONE L_CTRL_B // Resume in B
33                    // in the next tick
34  // Controller for state B
35  L_CTRL_B:
36      PRIO CTRL 3
37      PRIO CTRL 1
38      JLE T C 0
39      JT T B_2_A
40      DONE L_CTRL_B
41  // weak abort from B to A
42  B_2_A:
43      SETPC C L_C_A
44      SETPC O L_O_A
45      DONE L_CTRL_A // Resume in A
46                    //in the next tick
47
48  // Equations inside A
49  L_C_A:
50      IADD C pre(C) 1
51      DONE L_C_A
52  L_O_A:
53      IMUL O I 2
54      DONE L_O_A
55
56  // Equations inside B
57  L_C_B:
58      ISUB C pre(C) 1
59      DONE L_C_B
60  L_O_B:
61      ISUB O I 5
62      DONE L_O_B
```

Figure 5.10.: KLP assembler for the automaton from Figure 5.9

The *KLP*-assembler for this program (Figure 5.10) consists of the following parts. We assign one register to the control of the complete automaton. (For more complex

programs, we need one automaton per hierarchy.) This control part runs with higher priority than the contained equations. In Line 14 the code to check for the execution of state A starts. First, we check whether the trigger X of the strong abortion is true, in this case, we jump directly to A_2_B_S, which sets the program counter of C and O according to state B and also checks for weak abortions. Otherwise, we set the priority to 3, which indicates lower priority than the equations for C and O, which are now executed. Then we raise the priority of the controller back to 1. Thereafter, we check for the weak abortions by comparing C to 0.

In this example, we do not have to alter the priorities of C and O, because they do not have any data dependencies: otherwise, this would have been part of the transition code. We also do not use the value of the register that is assigned to the controller. However, it could be used to store information on the current state, in order to implement the history operator, which restarts an automaton in the last active state, and not in its initial one.

The translation from Scade automata to *KLP* assembler is similar to the translation of SyncCharts to *KEP* assembler as described in Section 4.5. The main difference is that the *KEP* has a single point of control. Therefore, a solution with watchers which monitor the unique program counter and reset it when a transition occurs, is feasible. In that aproach a state is implicitly declared active, when the program-counter is currently inside the scope of the state, while for approach on the *KLP* a state is active when the program counter of the controller is in the corresponding handler.

5.3. Experimental Results

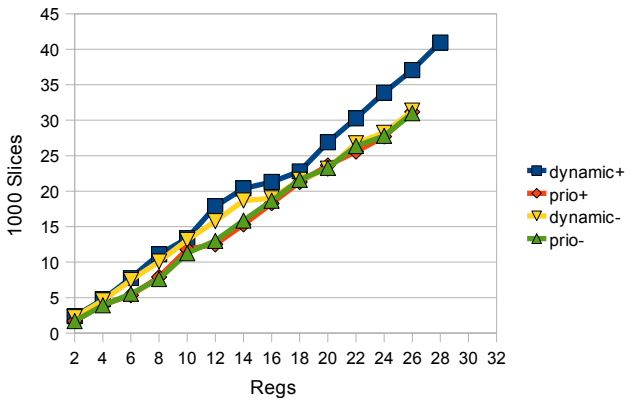
5.3.1. Evaluation

The *KLP* is developed in Esterel v7 with Esterel-Studio, from which either a software emulation in C or an hardware description in VHDL is generated. For evaluation purposes, we extend it by a test-driver that can communicate via a simple protocol to set inputs, read outputs, load programs, and get information on the current execution, such as the execution trace, or the reaction time, as detailed in Chapter 7.

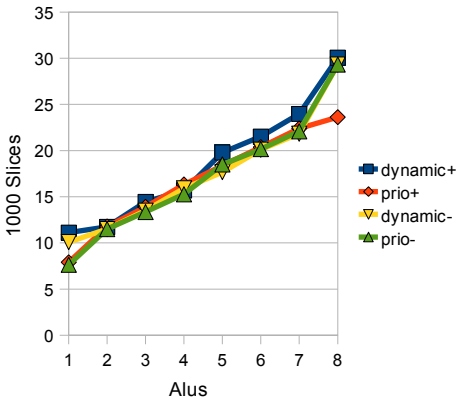
For the validation of the *KLP*, we use the tool lurette Jahier et al. [2006] to generate random traces for the benchmarks and then compile the benchmarks with the Lustre v4 compiler to get the correct output for the test traces. These outputs are compared automatically by the KReP Evalbench to the output of the *KLP*, when the same benchmarks are executed, to validate the correctness of the compiler and the processor. At the same time, the reaction times are measured.

5.3.2. Resource Usage

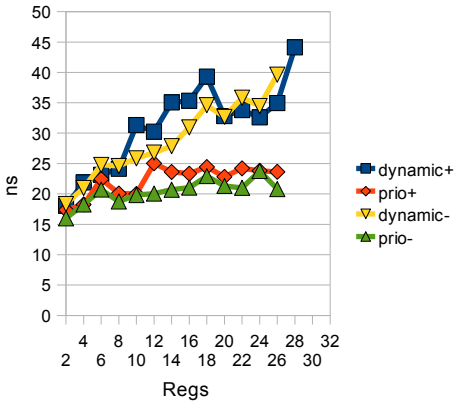
We compare the resource usage of four difference variants of the *KLP*. The dynamic and priority based scheduling (dynamic, prio) can be combined with direct support for hardware clocks (+, -). For the measurement, the *KLP* was synthesized for a Virtex 4 FPGA with Xilinx ISE 11, using the standard settings for different numbers of registers



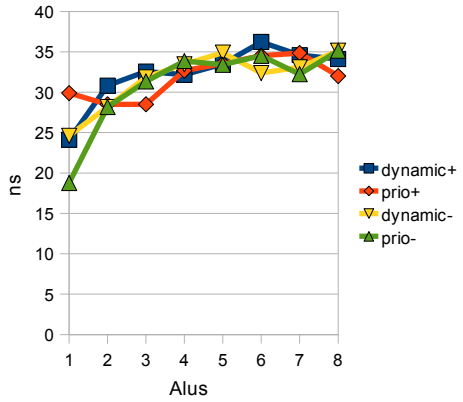
(a) Hardware usage of the KLP per register for one ALU



(b) Hardware usage of the KLP per processing unit for 8 registers



(c) Minimal delay for the KLP per register for one ALU



(d) Minimal delay of the KLP per processing unit for 8 registers

Figure 5.11.: KLP Resource Usage

and processing units. Figure 5.11a shows the number of slices, depending on the number of registers. As one can see, the dynamic scheduling needs more area than the priority based scheduling, but in particular for the dynamic scheduling, the direct support for clocks is very cheap.

Figure 5.11b shows the hardware usage depending on the number of processing units for 8 registers. Here, the differences between the configurations of the *KLP* are even smaller. As one would expect, adding additional processing units is more costly than adding registers.

The difference is much bigger for the minimal clock cycle, shown in Figure 5.11c for different numbers of registers. For the priority based scheduling, the minimal clock cycle is almost constant, while for the dynamic scheduling, where each register can depend on each other, it increases linearly. There is an anomaly with the delay, the execution for 20 registers takes longer than for 22, 24, 26, or 28 registers. This seems to come from

optimizations in the VHDL code, since the delay computed by the early performance estimation does not show this anomaly.

Figure5.11d shows the minimal delay depending on the number of processing units, for 8 registers. Here, the different configurations perform about the same. In fact, the scheduling is more involved with priorities, because first the maximal priority needs to be determined, and all active processes with this priority are scheduled. This is in particular the reason for the steep increase from 1 to 2 processing units. For the dynamic scheduling, we can simply schedule all active processes.

5.3.3. Execution Times

We compare the *KLP* to the execution of Lustre programs on a Microblaze processor, running on the same FPGA. We focus on the pure execution times and ignore all delay coming from IO handling like memory access.

For the evaluation, the following benchmarks are used, which can be found in the Appendix A:

abro a Lustre version of the well-known Esterel example.

counter a counter in Lustre, just one flow without clocks.

elevator lus a elevator controller, written in Lustre

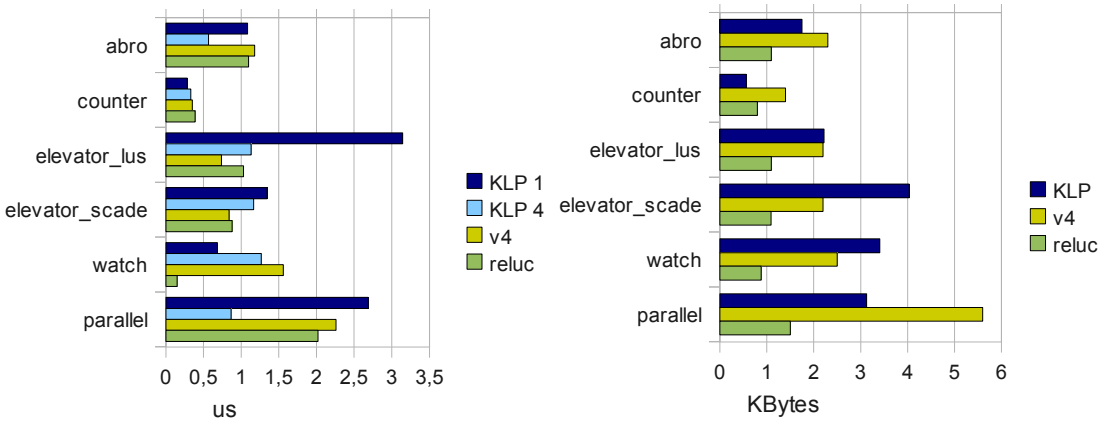
elevator scade a controller for the same elevator, written in Scade. This model contains no clocks.

watch a simple watch.

parallel an example for parallel computations without data dependencies.

The reaction times for a set of benchmarks when running on a Microblaze core with 100 MHz are compared to the runtime on the *KLP*, with 1 or 4 processing units. We synthesized the *KLP* to get the maximal possible frequency and used the software emulation to measure the number of instruction cycles that are needed to compute one reaction. Figure 5.12a shows the measured worst case reaction times. For the *KLP* with one ALU, for most examples, the execution takes longer than the execution on a Microblaze, independent of the used compiler. However, the *KLP* with 4 ALUexecutes the programs faster than the Microblaze. Note that the actual benefit depends on the program, because the parallel execution can not always be used.

Figure 5.12b compares the generated code size for the *KLP* with code generated by different Lustre compilers: the Lustre v4 compiler from VERIMAG and the reluc compiler from SCADE. For the *KLP*-compiler the code size of the *KLP*-assembler was measured. For the other compilers, C code was generated and compiled into object code for a standard processor, using the gcc. The code of the *KLP* is relatively large. There are two reasons for this:



(a) Reaction times for the KLP with one and four processing units, compared to running SW implementations on a microblaze processor

(b) Code size compiled for the KLP or for standard hardware using different Lustre compilers

Figure 5.12.: KLP Benchmarks

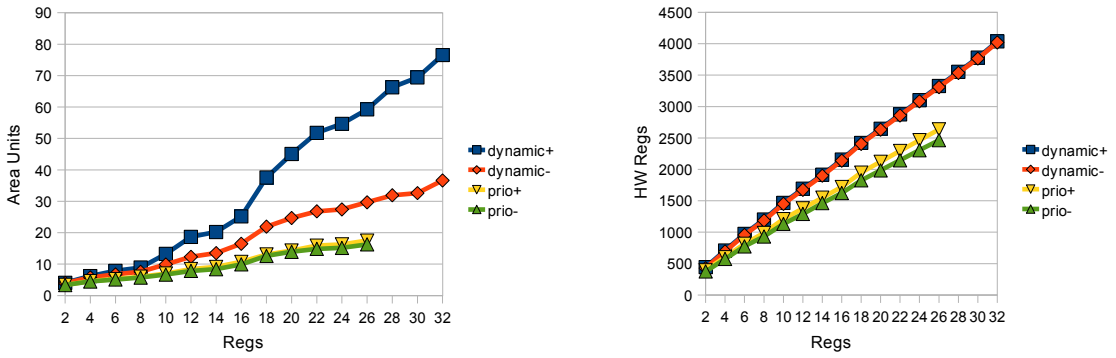
1. Additional code is needed for the parallel execution: for each register a DONE instruction is needed both for the runtime code and the initialization plus the setup code. To reduce this overhead, the compiler could analyze dependencies and combine registers that cannot be executed in parallel.

2. The compiler only performs limited optimizations, in particular compared to the reluc compiler. Note that for the simple counter example, which has few possibilities for optimization, the code for *KLP* is smaller than for the other compilers.

5.4. Hardware Description with Esterel v7

Beside the SyncCharts, Esterel Studio offers another graphical description for models: *architecture diagrams*. These are classical hardware block diagrams. Compared to SyncCharts, both the semantics and the translation to Esterel is much simpler for architectural diagrams: all modules run in parallel, and the links denote signal renamings. However, for the high-level view of the architecture, they are much more convenient than textual Esterel.

Esterel Studio allows an *early performance estimation* of the required hardware resources. The net-list code is generated and estimations for the occupied area, the number of hardware registers and the minimal delay are obtained by abstract interpretation. While these numbers are not accurate and do not consider optimizations that are performed by the synthesis tools on the data-path, the estimation helps to get an overview of the needed resources and to find the modules that consume most resources. On the other hand, the early performance estimation is much faster than a complete synthesis of the design. Figure 5.13 shows the resource usage that is reported by the early per-



(a) Area usage

(b) Register usage

Figure 5.13.: Early performance estimation of the *KLP* resource usage per register for one processing unit.

formance estimation. While the overall trend is the same as for the actual hardware synthesis in Figure 5.11, the difference between the dynamic scheduling and the priority based scheduling is much bigger in the early performance estimation than in the actual synthesis.

One of the great benefits of Esterel is the possibility for formal verification. The model checking of Esterel modules, which is implemented in Esterel Studio, was not used frequently for this processor design, because the specification of the allowed behavior of the environment was too complex. For example, we would have to specify what are valid *KLP* programs, because for invalid programs the behavior of the *KLP* is not specified. But Esterel Studio also offers the sequential equivalence check, to prove that two modules have the same behavior. This was frequently used to show that implemented optimizations do not change the behavior of the processor.

The biggest obstacle when designing hardware with Esterel are the causality cycles. The compiler will reject all cyclic programs, but in complex programs it is often hard to understand why the compiler shows a cycle, in particular when the cycle spawns over nearly all signals of the programs, which are then all highlighted in the tool. The user can then do little more than a trial and error approach to remove the cycle, *e. g.*, by adding pre modifiers to all signals which he suspects to cause the cycle. Of course, the user should try not to alter the behavior of the program at the same time. So allowing all constructive programs, and not just acyclic ones would already reduce the number of reported cycles by the compiler. Or the presentation of cycles in the tool should be improved, by giving the possibility to follow the dependencies in the cycle stepwise, together with an explanation for each dependency.

While Esterel is a convenient language to design hardware from behavior description, the programmer must be aware of the compilation steps that are taken from Esterel to the hardware description language and further into hardware, in order to implement efficient hardware.

5.5. Comparison of KEP and KLP

Since Lustre has less control constructs than Esterel, the instruction set of the *KLP* is much simpler than the one of the *KEP*. Another difference is the degree of concurrency: Esterel has a point of control, or in case of concurrency multiple points of control. In a Lustre program all equations run in parallel. The only order imposed on the execution comes from the data-dependencies. The main differences between the *KEP* and the *KLP* is that the *KLP* allows the parallel execution of instructions.

As mentioned before, the main features of the *KLP* are:

1. Direct support for clocks,
2. Multiple program counters,
3. Easy access to previous values, and
4. Parallel execution.

Taking a closer look at these, we make the observation that all this was already implemented in the *KEP*, except for the parallel execution. The clocks can be implemented by a suspend, the previous value at least of signals can be accessed, and the *KEP* supports a priority based scheduling. Therefore, we can also use the *KEP* to efficiently execute Lustre programs. The dataflow-part is computed into usual dataflow execution, as it is done when compiling Lustre to C. The control part, as the clocks, is expressed by the SUSPEND instruction.

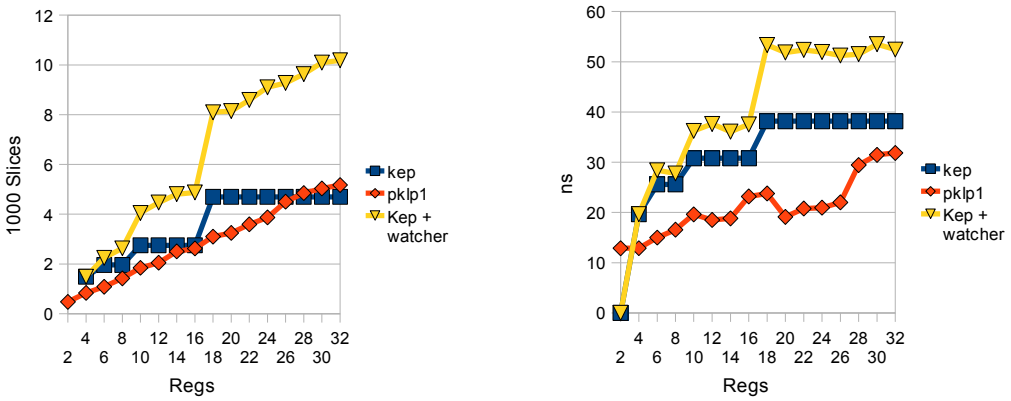
Another possibility to execute mixed control and dataflow diagrams would be to run the *KEP* and *KLP* in a tandem mode [Malik et al., 2009], executing the dataflow part on the *KLP* and the automata on the *KEP*. However, this would add communication overhead, since both processors must be able to exchange data back and forth within a tick. Furthermore, the *KEP* need to be able to reconfigure the *KLP* within a tick.

Figure 5.14 compares the resource uses of the *KEP* and the *KLP*, when both are synthesized from Esterel. Since the *KEP*-e does not contain data, the word size for the *KLP* was shrunk to two Bit. And since both preemption and concurrency are tightly coupled to the number of registers, two versions were measured for the *KEP*: one where the number of threads grows with the number of signals, but the number of watchers is fixed to four (kep). For kep+watcher, the number of watchers grows as well. This version scales clearly worse than the *KLP*. The number of registers of the *KLP* compares to the number of registers and threads of the *KEP* with a fixed number of watchers.

5.6. Further Optimizations and Open Problems

5.6.1. Static Scheduling

Since Lustre programs are required to be acyclic, there is always a static computable evaluation order of the equations, so that the computation of an equation only uses values that were already computed. So far, the compiler uses one register of the *KLP*



(a) Hardware usage of the KLP compared to the KEP

(b) Minimal delay for the KLP compared to the KEP

Figure 5.14.: KLP resource usage compared to the KEP

for each Lustre equation. However, the compiler could use the static scheduling to combine different equations, to reduce the number of necessary registers. Of course, care has to be taken that still all values that are used by other registers are available.

5.6.2. Clock registers

The *KLP* used 32 Bit both for integer and boolean registers. Having special clock registers, which only have only one value bit, would reduce the resource usage. While this is already supported by the instruction set, this would also make the *KLP* less flexible.

5.6.3. Memory Access

One of the problems in modern processor design is the memory gap: the delay to load new instructions is the bottle-neck for the whole execution. We do not address this problem here, in fact, the *KLP* even worsens the situation: it does not need one but multiple instructions in each clock cycle. Common solution to this problem are instruction caches or Very Long Instruction Word *(VLIW)* architectures, where multiple instructions are loaded simultaneously. However, this requires a static order of all instructions that can be executed simultaneously. We extended the *KLP* by a simple instruction cache that fetches the next instruction for each register. Furthermore, since the *KLP* is run on an evaluation FPGA board, it could use multiple instruction ROMs.

6. Worst Case-Reaction-Time Analysis

Contents

Apart from efficiency concerns, which may initially have been the primary driver towards reactive processing architectures, one of their advantages is their timing predictability. To leverage this, the strl2kasm compiler contains a timing analysis capability. As we here are investigating the timing behavior for reactive systems, we are specifically concerned with computing the maximal time it takes to compute a single reaction. We refer to this time, which is the time from given input events to generated output events, as Worst Case Reaction Time (*WCRT*). The WCRT determines the maximal rate for the interaction with the environment.

There are two main factors that facilitate the WCRT analysis in the reactive processing context. These are on the one hand the synchronous execution model of Esterel, and on the other hand the direct implementation of this execution model on a reactive processor. Furthermore, these processors are not designed to optimize (average) performance for general purpose computations, and hence do not have a hierarchy of caches, pipelines, branch predictors, etc. This leads to a simpler design and execution behavior and further facilitates WCRT analysis. Furthermore, there are reactive processors, such as the *KEP*, which allow to fix the reaction lengths to a previously determined number of clock cycles, irrespective of the number of instructions required to compute a specific reaction, in order to minimize the jitter.

WCRT differs from WCET fundamentally in that it deals with the timing of *stabilization* rather than *iteration* processes. As a consequence WCRT does not need to analyze the termination of computational loops, a potentially undecidable problem. WCRT assumes that all dependencies in the control flow are acyclic and the propagation of control is a monotonic process in which each atomic control point is only ever executed at most once, or a finite number of times for schizophrenic programs. On the other hand, WCRT for synchronous processing must handle *non-atomic control flow* including features such as hierarchical and concurrent threads, priorities and preemption.

We will consider three different approaches for the WCRT analysis on the *KEP*:

1. A graph based approach on the CKAG, an intermediate representation of the strl2kasm compiler [Boldt et al., 2008].
2. An interface algebra based on (max,+)-algebra [Mendler et al., 2009], and
3. a model checking approach [Roop et al., 2009b] to determine the WCRT.

6.1. The Graph Based Approach

One possibility to determine an estimation for the WCRT on the *KEP* is a search for the longest path, performed on the Concurrent KEP Assembler Graph (*CKAG*). It computes for each statement how many instruction will be needed until a time delimiting instruction is reached. The analysis computes the WCRT in terms of *KEP* instruction cycles, which roughly match the number of executed Esterel statements. As part of the WCRT analysis, we also present an approach to calculate potential instantaneous paths, which may be used in compiler analysis and optimizations that go beyond WCRT analysis. This approach is implemented in the strl2kasm compiler.

6.1.1. The Concurrent KEP Assembler Graph

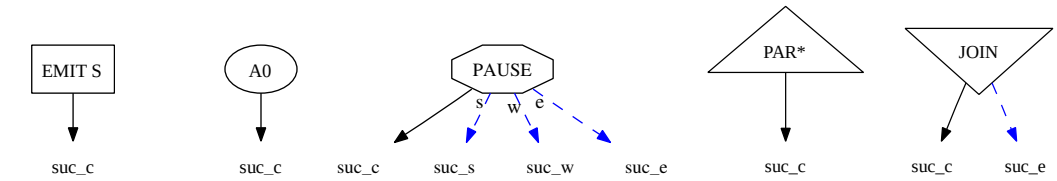
The CKAG is a directed graph composed of various types of nodes and edges to match *KEP* program behavior. It is used during compilation from Esterel to *KEP* assembler, for, *e. g.*, priority assigning, dead code elimination, further optimizations and the WCRT analysis. The CKAG is generated from the Esterel program via a simple structural translation. The only non-trivial aspect is the determination of non-instantaneous paths, which is needed for certain edge types. For convenience, we label nodes with *KEP* instructions; however, we could alternatively have used Esterel instructions as well.

The CKAG distinguishes the following sets of nodes, see also Figure 6.1:

***L*:** *label nodes* (ellipses);

***T*:** *transient nodes* (rectangles), which includes EMIT, PRESENT, etc.;

***D*:** *delay nodes* (octagons), which correspond to delayed *KEP* instructions (PAUSE, AWAIT, HALT, SUSTAIN);



(a) transient (b) label (c) delay (d) fork (e) join

Figure 6.1.: Nodes and edges of a Concurrent KEP Assembler Graph.

***F*:** *fork nodes* (triangles), corresponding to PAR/PARE;

***J*:** *join nodes* (inverted triangles), corresponding to JOIN;

***N*:** set of all nodes, with $N = T \cup L \cup D \cup F \cup J$.

We also define

***A*:** the *abort nodes*, which denote abortion scopes and correspond to [W]ABORT and SUSPEND; note that $A \subseteq T$.

For each fork node n ($n \in F$) we define

***n.join*:** the JOIN statement corresponding to n ($n.join \in J$), and

***n.sub*:** the transitive closure of nodes in threads spawned by n.

For abort nodes n ($n \in A$) we define

***n.end*:** the end of the abort scope opened by n, and

***n.scope*:** the nodes within n's abort scope.

A non-trivial task when defining the CKAG structure is to properly distinguish the different types of possible control flow, in particular with respect to their timing properties (instantaneous or delayed). We define the following types of successors for each n:

***n.suc*$_c$:** the *control successors*. These are the nodes that follow sequentially after n, considering normal control flow without any abortions. For $n \in F$, $n.suc_c$ includes the nodes corresponding to the beginnings of the forked threads.

The successors are statically inserted, based on the syntax of the Esterel program. Depending on the actual behavior, some of these can be removed. If n is the last node of a concurrent thread, $n.suc_c$ includes the node for the corresponding JOIN—unless n's thread is instantaneous and has a (provably) non-instantaneous sibling thread. Furthermore, the control successors exclude those reached via a preemption ($n.suc_w$, $n.suc_s$)—unless n is an immediate strong abortion node, in which case $n.end \in n.suc_c$.

$n.suc_w$: the *weak abort successors*. If $n \in D$, this is the set of nodes to which control can be transferred immediately, *i. e.*, when entering n at the end of a tick, via a weak abort; if n exits a trap, then $n.suc_w$ contains the end of the trap scope; otherwise it is $\emptyset$.

If $n \in D$ and $n \in m.scope$ for some abort node m, it is $m.end \in n.suc_w$ in case of a weak immediate abort, or in case of a weak abort if there can (possibly) be a delay between m and n.

$n.suc_s$: the *strong abort successors*. If $n \in D$, these are the nodes to which control can be transferred after a delay, *i. e.*, when restarting n at the beginning of a tick, via a strong abort; otherwise it is $\emptyset$.

If $n \in D$ and $n \in m.scope$ for some strong abort node m, it is $m.end \in n.suc_s$.

Note that this is not a delayed abort in the sense that an abort signal in one tick triggers the preemption in the next tick. Instead, this means that first a delay has to elapse, and the abort signal must be present at the next tick (relative to the tick when n is entered) for the preemption to take place.

$n.suc_e$: the *exit successors*. These are the nodes that can be reached by raising an exception.

$n.suc_f$: the *flow successors*. This is the set $n.suc_c \cup n.suc_w \cup n.suc_s$.

For $n \in F$ we also define two kinds of *fork abort successors*. These serve to ensure a correct priority assignment to parent threads in case there is an abort out of a concurrent statement.

$n.suc_{wf}$: the *weak fork abort successors*. This is the union of $m.suc_w \setminus n.sub$ for all $m \in n.sub$ where there exists an instantaneous path from n to m.

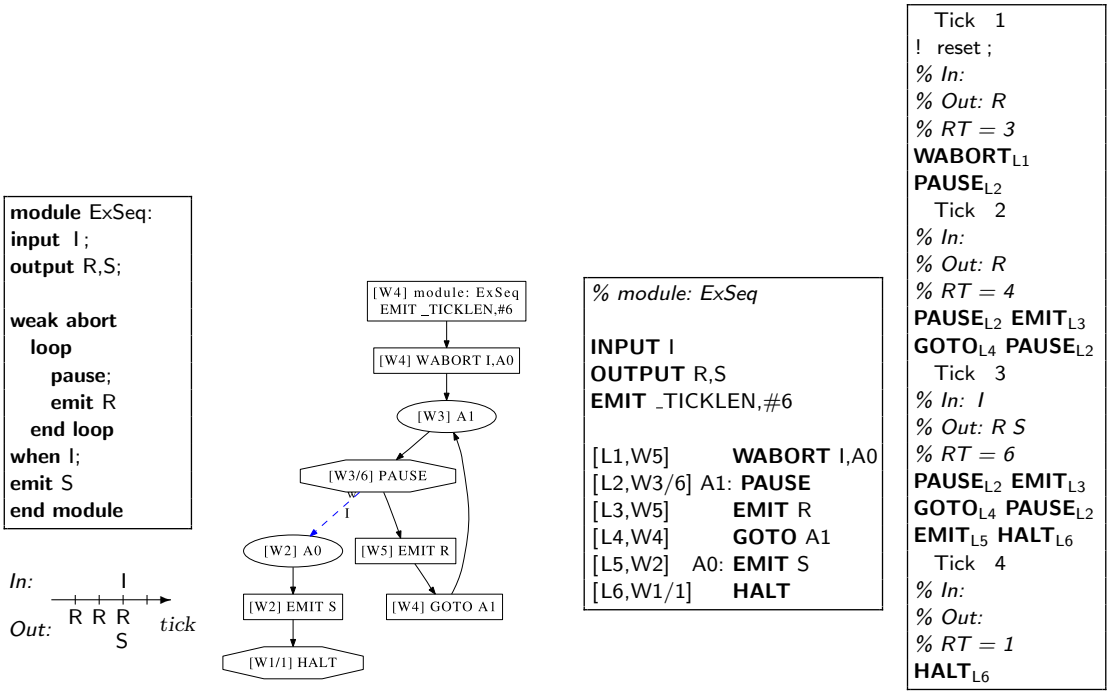
$n.suc_{sf}$: the *strong fork abort successors*. This is the set $\cup\{(m.suc_w \cup m.suc_s) \setminus n.sub \mid m \in n.sub\} \setminus n.suc_{wf}$.

In the graphical representation, control successors are shown by solid lines, all other successors by dashed lines, annotated with the kind of successor.

The CKAG is built from Esterel source by traversing recursively over its Abstract Syntax Tree (AST) generated by the CEC [Edwards, 2006]. Visiting an Esterel statement results in creating the according CKAG node. A node typically contains exactly one statement, except label nodes containing just address labels and fork nodes containing one PAR statement for each child thread initialization and a PARE statement. When a delay node is created, additional preemption edges are added according to the abortion/exception context.

Note that some of the successor sets defined above cannot be determined precisely by the compiler, but have to be (conservatively) approximated instead. This applies in particular to those successor types that depend on the existence of an instantaneous path. Here it may be the case that for some pair of nodes there does not exist such an

instantaneous path, but that the compiler is not able to determine that. In such cases, the compiler conservatively assumes that there may be such an instantaneous path.

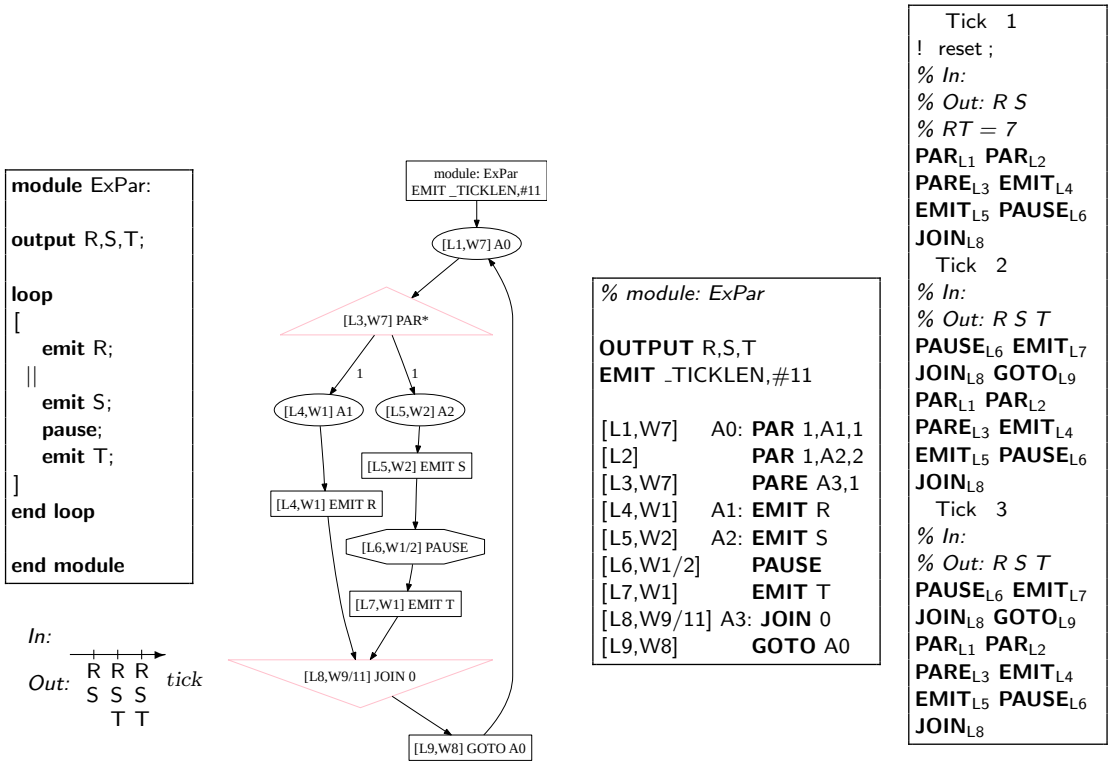


(a) Esterel code and sample trace (b) CKAG (c) KEP assembler (d) Execution trace

Figure 6.2.: A sequential Esterel example. The body of the KEP assembler program (without interface declaration and initialization of the TickManager) is annotated with line numbers L1–L6, which are also used in the CKAG and in the trace to identify instructions. For each instruction WCRT is given, for non-instantaneous statements also the WCRT when the execution starts at this instruction. The trace shows for each tick the input and output signals that are present and the reaction time (RT), in instruction cycles.

Examples As an example of a simple, non-concurrent program consider the module ExSeq shown in Figure 6.2a. As the sample execution trace illustrates, the module emits signal R in every instant, until it is aborted by the presence of the input signal I. As this is a *weak* abortion, the abortion body gets to execute (emit R) one last time when it is aborted, followed by an emission of S.

The program ExPar shown in Figure 6.3a introduces concurrency: a thread that emits R and then terminates, and a concurrent thread that emits S, pauses for an instant, emits T, and then terminates are executed in an infinite loop. During each loop iteration, the



(a) Esterel code and sample trace (b) CKAG (c) KEP assembler (d) Execution trace

Figure 6.3.: A concurrent example program.

parallel terminates when both threads have terminated, after which the subsequent loop iteration is started instantaneously, that is, within the same tick.

6.1.2. Sequential WCRT Algorithm

First we present a WCRT analysis of sequential CKAGs (no fork and join nodes). Consider again the ExSeq example in Figure 6.2a.

The longest possible execution occurs when the signal I becomes present, as is the case in Tick 3 of the example trace shown in Figure 6.2d. Since the abortion triggered by I is weak, the abort body is still executed in this instant, which takes four instructions: $\mathsf{PAUSE}_{\mathsf{L2}}$, $\mathsf{EMIT}_{\mathsf{L3}}$, the $\mathsf{GOTO}_{\mathsf{L4}}$, and $\mathsf{PAUSE}_{\mathsf{L2}}$ again. Then it is detected that the body has finished its execution for this instant, the abortion takes place, and $\mathsf{EMIT}_{\mathsf{L5}}$ and $\mathsf{HALT}_{\mathsf{L6}}$ are executed. Hence the longest possible path takes six instruction cycles.

The sequential WCRT is computed via a Depth First Search (DFS) traversal of the CKAG, see the algorithm in Figure 6.4. For each node n a value $n.inst$ is computed,

```
1  int getWcrtSeq(g)      // Compute WCRT for sequential CKAG g
2    forall n ∈ N do n.inst := n.next := ⊥ end
3    getInstSeq(g.root)
4    forall d ∈ D do getNextSeq(d) end
5    return max ({g.root.inst} ⋃ {d.next : d ∈ D})
6  end
```

```
1  int getInstSeq(n)      // Compute statements instantaneously reachable from node n
2    if n.inst = ⊥ then
3      if n ∈ T ∪ L then
4        n.inst := max {getInstSeq(c) : c ∈ n.suc_c} + cycles(n.stmt)
5      elif n ∈ D then
6        n.inst := max {getInstSeq(c) : c ∈ n.suc_w ∪ n.suc_e} + cycles(n.stmt)
7      fi
8    fi
9    return n.inst
10 end
```

```
1  int getNextSeq(d)                    // Compute statements instantaneously reachable
2    if d.next = ⊥ then                 // from delay node d at tick start
3      d.next := max {getInstSeq(c) : c ∈ d.suc_c ∪ d.suc_s} + cycles(d.stmt)
4    fi
5    return d.next
6  end
```

Figure 6.4.: WCRT algorithm, restricted to sequential programs. The nodes of a CKAG g are given by $N = T \cup L \cup D \cup F \cup J$ (see Section 6.1.1), $g.root$ indicates the first KEP statement; cycles($stmt$) returns the number of instruction cycles to execute $stmt$, see third column in Figure 5.3.

which gives the WCRT from this node on in the same instant when execution reaches the node. For a transient node, the WCRT is simply the maximum over all children plus its own execution time.

For non-instantaneous delay nodes we distinguish two cases within a tick: control can *reach* a delay node d, meaning that the thread executing d has already executed some other instructions in that tick, or control can *start* in d, meaning that d must have been reached in some preceding tick. In the first case, the WCRT from d on within an instant is expressed by the $d.inst$ variable already introduced. For the second case, an additional value $d.next$ stores the WCRT from d on within an instant; "next" here expresses that in the CKAG traversal done to analyze the overall WCRT, the $d.next$ value should not be included in the current tick, but in a next tick. Having these two values ensures that the algorithm terminates in the case of non-instantaneous loops: to compute $d.next$ we might need the value $d.inst$.

For a *delay node*, we also have to take abortions into account. The handlers (*i. e.*, their continuations—typically the end of an associated abort/trap scope) of weak abortions and exceptions are instantaneously reachable, so their WCRTs are added to the $d.inst$ value. In contrast, the handlers of strong abortions cannot be executed in the same instant the delay node is reached, because according to the Esterel semantics an abortion body is not executed at all when the abortion takes place. On the *KEP*, when a strong abort takes place, the delay nodes where the control of the (still active) threads in the abortion body resides are executed once, and then control moves to the abortion handler. In other words, control cannot move from a delay node d to a (strong) abortion handler when control reaches d, but only when it starts in d. Therefore, the WCRT of the handler of a strong abortion is added to $d.next$, and not to $d.inst$.

Weak abortions are not taken into account for $d.next$, because it cannot contribute to a longest path. An abortion in an instant when a delay node is reached will always lead to a higher WCRT than an execution in a subsequent instant where a thread starts executing in the delay node.

The resulting WCRT for the whole program is computed as the maximum over all WCRTs of nodes where the execution may start. These are the start node and all delay nodes. To take into account that execution might start simultaneously in different concurrent threads, we also have to consider the *next* value of join nodes.

Consider again the example ExSeq in Figure 6.2. Each node n in the CKAG g is annotated with a label "W$\langle n.inst \rangle$" or, for a delay node, a label "W$\langle n.inst \rangle / \langle n.next \rangle$." In the following, we will refer to specific CKAG nodes with their corresponding KEP assembler line numbers L$\langle n \rangle$. It is $g.root$ = L1. The sequential WCRT computation starts initializing the *inst* and *next* values of all nodes to $\bot$ (line 2 in getWcrtSeq, Figure 6.4). Then getInstSeq(L1) is called, which computes L1.$inst$:= max { getInstSeq(L2) } + cycles($\text{WABORT}_{\text{L1}}$). The call to getInstSeq(L2) computes and returns L2.$inst$:= cycles(PAUSE_{L2}) + cycles(EMIT_{L5}) + cycles(HALT_{L6}) = 3, hence L1.$inst$:= 3 + 2 = 5. Next, in line 4 of getWcrtSeq, we call getNextSeq(L2), which computes L2.$next$:= getInstSeq(L3) + cycles(PAUSE_{L2}). The call to getInstSeq(L3) computes and returns L3.$inst$:= cycles(EMIT_{L3}) + cycles(GOTO_{L4}) + L2.$inst$ = 1 + 1 + 3 = 5. Hence L2.$next$:= 5 + 1 = 6, which corresponds to the longest path triggered by the presence of signal I, as we have seen earlier. The WCRT analysis therefore inserts an "EMIT _TICKLEN, #6" instruction before the body of the KEP assembler program to initialize the TickManager (see Chapter 4) accordingly, as can be seen in Figure 6.2c.

6.1.3. Instantaneous Statement Reachability for Concurrent Esterel Programs

It is important for the WCRT analysis whether all threads between join and its corresponding fork can terminate immediately. If this is the case, we have to sum up the instantaneous execution time before the fork, after the join, and for the threads itself. Otherwise, we know that the code before the fork and after the join are never executed within the same instant. The algorithm for instantaneous statement reachability computes for a source and a target node whether the target is reachable instantaneously from

the source. Source and target have to be in sequence to each other, *i. e.*, not concurrent, to get correct results.

In simple cases like EMIT or PAUSE the sequential control flow successor is executed in the same instant respectively next instant, but in general the behavior is more complicated. The parallel, for example, will terminate instantaneously if all sub-threads are instantaneous or an EXIT will be reached instantaneously; it is non-instantaneous if at least one sub-thread is not instantaneous.

The complete algorithm is presented in detail elsewhere [Boldt, 2007a]. The basic idea is to compute for each node three potential reachability properties: *instantaneous*, *non-instantaneous*, and *exit-instantaneous*. Note that a node might be as well (potentially) *instantaneous* as (potentially) *non-instantaneous*, depending on the signal context. Computation begins by setting the *instantaneous* predicate of the source node to *true* and the properties of all other nodes to *false*. When any property is changed, the new value is propagated to its successors. If we have set one of the properties to *true*, we will not set it to *false* again. Hence the algorithm is monotonic and will terminate. Its complexity is determined by the amount of property changes which are bounded to three for all nodes, so the complexity is $O(3 * |N|) = O(|N|)$.

The most complicated computation is the property *instantaneous* of a join node, because several attributes have to be fulfilled for it to be *instantaneous*:

- For each thread, there has to be a (potentially) instantaneous path to the join node.

- The predecessor of the join node must not be an EXIT, because EXIT nodes are no real control flow predecessors. At the Esterel level, an exception (exit) causes control to jump directly to the corresponding exception handler (at the end of the corresponding trap scope); this jump may also cross thread boundaries, in which case the threads that are jumped out of and their sibling threads terminate.

  To reflect this at the KEP level, an EXIT instruction does not jump directly to the exception handler, but first executes the JOIN instructions on the way, to give them the opportunity to terminate threads correctly. If a JOIN is executed this way, the statements that are instantaneously reachable from it are not executed, but control instead moves on to the exception handler, or to another intermediate JOIN. To express this, we use the third property besides *instantaneous* and *non-instantaneous*: *exit-instantaneous*.

Roughly speaking the *instantaneous* property is propagated via for-all quantifier, *non-instantaneous* and *exit-instantaneous* via existence-quantifier.

Most other nodes simply propagate their own properties to their successors. The delay node propagates in addition its *non-instantaneous* predicate to its delayed successors and *exit nodes* propagate *exit-instantaneous* reachability, when they themselves are reachable instantaneously.

6.1.4. General WCRT Algorithm

The general algorithm, which can also handle concurrency, is shown in Figure 6.5. It emerges from the sequential algorithm that has been described in Section 6.1.2 by enhancing it with the ability to compute the WCRT of fork and join nodes. Note that the instantaneous WCRT of a join node is needed only by a fork node, all other transient nodes and delay nodes do not use this value for their WCRT. The WCRT of the join node has to be accounted for just once in the instantaneous WCRT of its corresponding fork node, which allows the use of a DFS-like algorithm.

The instantaneous WCRT of a fork node is simply the sum of the instantaneously reachable statements of its sub-threads, plus the PAR statement for each sub-thread and the additional PARE statement.

The join nodes, like delay nodes, also have a *next* value. When a fork-join pair (f, j) could be *non-instantaneous*, we have to compute a WCRT $j.next$ for the next instants analogously to the delay nodes. Its computation requires first the computation of all sub-thread *next* WCRTs. Note that in case of nested concurrency these *next* values can again result from a join node. But at the innermost level of concurrency the *next* WCRT values all stem from delay nodes, which will be computed before the join *next* values. The delay *next* WCRT values are computed the same way as in the sequential case except that only successors within of the same thread are considered. Successors of a different thread are called *inter-thread-successors* and their WCRT values are handled by the according join node. The join *next* value is the maximum of all inter-thread-successor WCRT values and the sum of the maximum *next* value for every thread.

If the parallel does not terminate instantaneously, all directly reachable states are reachable in the next instant. Therefore we have to add the execution time for all statements that are instantaneously reachable from the join node.

The whole algorithm computes first the *next* WCRT for all delay and join nodes; it computes recursively all needed *inst* values. Thereafter the instantaneous WCRT for all remaining nodes is computed. The result is simply the maximum over all computed values.

Consider the example in Figure 6.3a. First we note that the fork/join pair is always *non-instantaneous*, due to the $\mathsf{PAUSE}_{\mathsf{L6}}$ statement. We compute $\mathsf{L6}.next = \mathsf{cycles}(\mathsf{PAUSE}_{\mathsf{L6}}) + \mathsf{cycles}(\mathsf{EMIT}_{\mathsf{L7}}) = 2$. From the fork node L3, the PAR and PARE statements, the instantaneous parts of both threads and the JOIN are executed, hence $\mathsf{L3}.inst = 2 \times \mathsf{cycles}(\mathsf{PAR}) + \mathsf{cycles}(\mathsf{PARE}) + \mathsf{cycles}(\mathsf{JOIN}) + \mathsf{L4}.inst + \mathsf{L5}.inst = 2+1+1+1+2 = 7$. It turns out that the WCRT of the program is $\mathsf{L8}.next = \mathsf{L6}.next + \mathsf{L8}.inst = 2 + 9 = 11$. Note that the JOIN statement is executed twice.

A known difficulty when compiling Esterel programs is that due to the nesting of exceptions and concurrency, statements might be executed multiple times in one instant. This problem, also known as *reincarnation*, is handled correctly by the algorithm. Since we compute nested joins from inside to outside, the same statement may effect both the instantaneous and non-instantaneous WCRT, which are added up in the next join. This exactly matches the possible control-flow in case of reincarnation. Even when a statement is executed multiple times in an instant, we compute a correct upper bound

```
1  int getWcrt(g)     // Compute WCRT for a CKAG g
2    forall n ∈ N do n.inst := n.next := ⊥ end
3    forall d ∈ D do getNext(d) end
4    forall j ∈ J do getNext(j) end // Visit according to hierarchy (inside out)
5    return max ({getInst(g.root)} ⋃ {n.next : n ∈ D ∪ J})
6  end
```

```
1  int getInst(n)      // Compute statements instantaneously reachable from node n
2    if n.inst = ⊥ then
3      if n ∈ T ∪ L then
4        t.inst := max {getInst(c) : c ∈ suc_c \ J} + cycles(n.stmt)
5      elif n ∈ D then
6        n.inst := max {getInst(c) : c ∈ suc_w ∪ suc_e \ J} + cycles(n.stmt)
7      elif n ∈ F then
8        n.inst := ∑_{t∈n.suc_c} t.inst + cycles(n.par_stmts) + cycles(PARE)
9        prop := reachability(n, n.join) // Compute instantaneous reachability of join from fork
10       if prop.instantaneous or prop.exit_instantaneous then
11         n.inst += getInst(n.join)
12       elif prop.non_instantaneous then
13         n.inst += cycles(JOIN) // JOIN is always executed
14       fi
15     elif n ∈ J then
16       n.inst := max{getInst(c) : c ∈ suc_c ∪ suc_e} + cycles(n.stmt);
17     fi
18   fi
19   return n.inst
20 end
```

```
1  int getNext(n)                    // Compute statements instantaneously reachable
2    if n.next = ⊥ then              // from delay node d at tick start
3      if n ∈ D then
4      n.next := max {getInst(c) : c ∈ suc_c ∪ suc_s \ J ∧ c.id = n.id} + cycles(n.stmt)
5      // handle inter thread successors by their according join nodes:
6      for m ∈ {c ∈ suc_c ∪ suc_s \ J : c.id ≠ n.id} do
7        j := according join node with j.id = m.id
8        j.next = max (j.next , getInst(m)+cycles(m.stmt)+cycles(j.stmt))
9      end
10     elif n ∈ J then
11       prop := reachability(n.fork, n) // Compute reachability predicates
12       if prop.non_instantaneous then
13         n.next := max ((∑_{t∈n.fork.suc_c} max{m.next : t.id = m.id}) + n.inst , n.next)
14       fi
15     fi
16   fi
17   return n.next
18 end
```

Figure 6.5.: General WCRT algorithm.

for the WCRT.

Regarding the complexity of the algorithm, we observe that for each node its WCRT's *inst* and *next* are computed at most once, and for all fork nodes a fork-join reachability analysis is additionally made, which has itself $O(|N|)$. So we get altogether a complexity of $O(|N| + |D| + |J|) + O(|F| * |N|) = O(2 * |N|) + O(|N|^2) = O(|N|^2)$.

6.1.5. Unreachable Paths

Signal informations are not taken into account in the algorithms described above. This can lead to a conservative (too high) WCRT, because the analysis may consider unreachable paths that can never be executed. In Figure 6.6a we see an unreachable path increasing unnecessarily the WCRT because of demanding signal I present and absent instantaneously, which is inconsistent. Nevertheless there is no dead code in the graph, but only two possible paths regarding to path signal predicates.

Figure 6.6b shows an unreachable parallel path that leads to a too high WCRT of the fork node, because the sub-paths cannot be executed at the same time. Furthermore, the parallel is declared as possibly instantaneous, even though it is not. Therefore, all statements which are instantaneously reachable from the join node are also added.

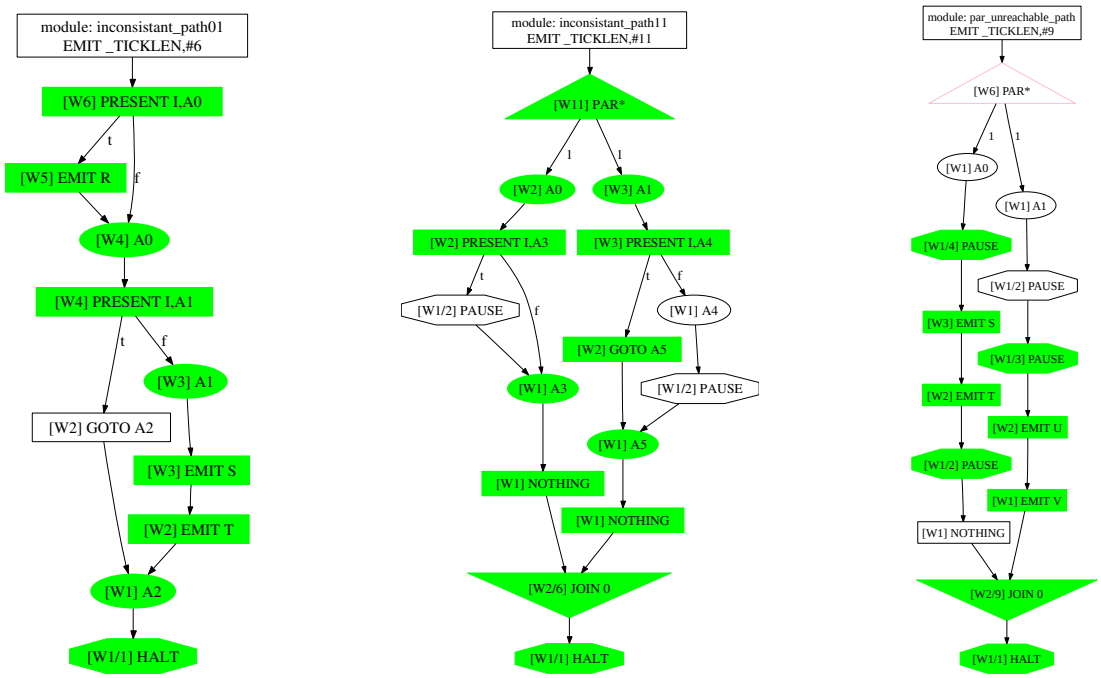
Another unreachable parallel path is shown in Figure 6.6c. This path is unreachable not because of signal informations but because of instantaneous behavior: the maximal paths of the two threads are never executed in the same instant. In other words, the system is never in a configuration (collection of states) such that both code segments become activated together. Instead of taking for each thread the maximum next WCRT and summing up it would be more exact to sum up over all threads next WCRT's executable instantaneously and then taking the maximum of these sums. Therefore we would have to enhance the reachability algorithm of the ability to determine how many ticks later a statement could be executed behind another. However, in this case the possible tick counts can become arbitrarily high for each node, so we would get a higher complexity and a termination problem. The analysis is conservative in simply assuming that all concurrent paths may occur in the same instant, and that all can be executed in the same instant as the join. While these special cases could be implemented in the analysis, it would increase the complexity of the algorithm. We will see in next section how such cases can be handled using an interface algebra.

6.1.6. Experimental Results

To evaluate the WCRT analysis approach presented here, it was implemented in the strl2kasm compiler.

Validation

To validate the correctness of the compilation scheme, as well as of the KEP itself, we have collected a fairly substantial validation suite, currently containing some 500 Esterel



(a) Inconsistent sequential path

(b) Inconsistent parallel path

(c) Unreachable configuration

Figure 6.6.: Unreachable Path Examples.

programs. These include all benchmarks made available to us, such as the *Estbench*[1], and other programs written to test specific situations and corner cases. An automated regression procedure compiles each program into KEP assembler, downloads it into the KEP, provides an input trace for the program, and records the output at each step.

For each program, the *Average Case Reaction Time* (ACRT) and WCRT for each program are measured. For these measurements, the KEP is operating in "freely running" mode, *i. e.*, _TICKLEN is left unspecified; the default would be to set _TICKLEN according to the (conservatively) estimated WCRT, in which case the measured ACRT and WCRT values would be equal to the estimated WCRT. The full benchmark suite runs through without any differences in output, and the analyzed WCRT is always safe; *i. e.*, not lower than the measured WCRT.

Esterel Studio is used to generate the input trace, using the "Full Transition Coverage" mode. Note that the traces obtained this way still did not cover all possible paths. However, at this point we consider it very probable that a compilation approach that handles all transition coverage traces correctly would also handle the remaining paths.

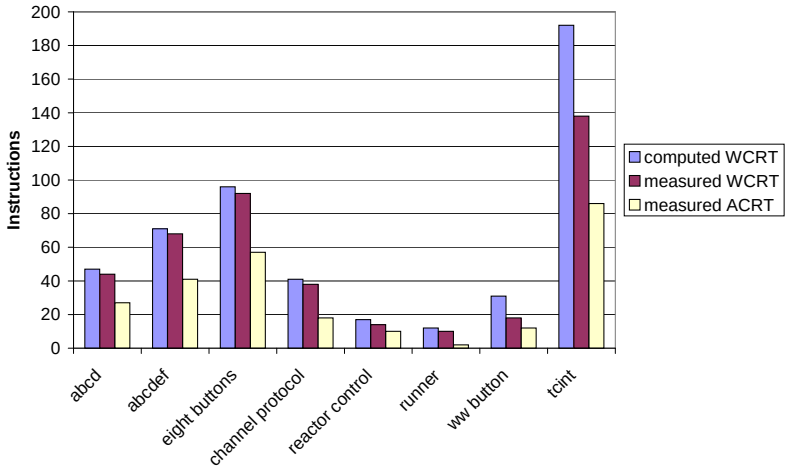


Figure 6.7.: Estimated and measured Worst and Average Case Reaction Times.

Accuracy of WCRT Analysis

As mentioned before, the WCRT analysis is implemented in the *KEP* compiler, and is used to automatically insert a correct EMIT _TICKLEN instruction at the beginning of the program, such that the reaction time is constant and as short as possible, without ever raising a timing violation by the TickManager. As discussed in Section 6.1.6, we measured the maximal reaction times and compared it to the computed value. Figure 6.7 provides a qualitative comparison of estimated and measured WCRT and measured ACRT, more details are given in Figure 6.8. The WCRT is never underestimated and the results are on average 22% too high. For each program, the lines of code, the computed WCRT and the measured WCRT with the resulting difference is given. The the average WCRT analysis time was measured on a standard PC (AMD Athlon XP, 2.2GHz, 512 KB Cache, 1GB Main Memory); as the table indicates, the analysis takes only a couple of milliseconds.

The table also compares the ACRT with the WCRT. The ACRT is on average about two thirds of the WCRT, which is relatively high compared to traditional architectures. In other words, the worst case on the *KEP* is not much worse than the average case, and padding the tick length according to the WCRT does not waste too much resources. On the same token, designing for worst-case performance, as typically must be done for hard real-time systems, does not cause too much overhead compared to the typical average-case performance design. Finally, the table also lists the number of scenarios generated by Esterel-Studio and accumulated logical tick count for the test traces.

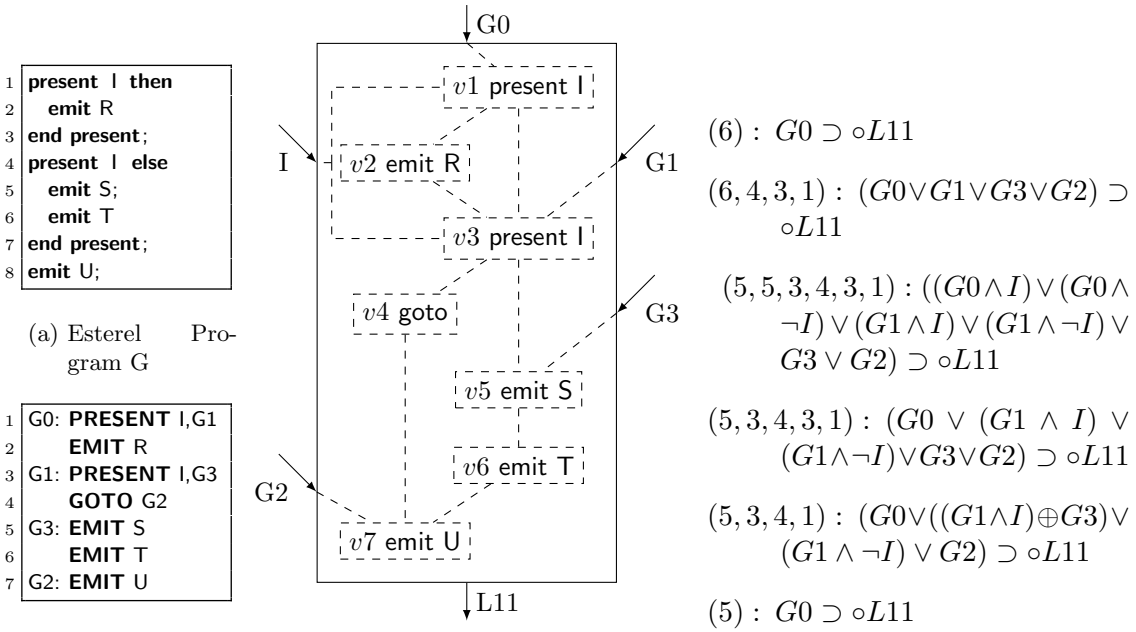
There is still significant room for improvement. Signal status are not taken into account, therefore the analysis includes some unreachable paths. Considering all signals would lead to an exponential growth of the complexity, but some local knowledge should

[1] `www1.cs.columbia.edu/~sedwards/software.html`

Esterel		WCRT			t_{an}	ACRT		Test	Ticks
Module name	LoC	WC_e	WC_m	$\Delta_{e/m}$	[ms]	AC_m	AC_m/WC_m	cases	
abcd	152	47	44	7%	1.0	27	61%	161	673
abcdef	232	71	68	4%	1.5	41	60%	1457	50938
eight_buttons	332	96	92	4%	2.0	57	62%	13121	45876
channel_protocol	57	41	38	8%	0.4	18	47%	114	556
reactor_control	24	17	14	21%	0.2	10	71%	6	20
runner	26	12	10	20%	0.3	2	20%	131	2548
ww_button	94	31	18	72%	1.0	12	67%	8	37
tcint	410	192	138	39%	2.8	86	62%	148	1325

Figure 6.8.: Detailed comparison of WCRT/ACRT times. The WC_e and WC_m data denote the estimated and measured WCRT, respectively, measured in instruction cycles. The ratio $\Delta_{e/m} := WC_e/WC_m - 1$ indicates by how much the analysis overestimates the WCRT. AC_m is the measured Average Case Reaction Time (ACRT), AC_m/WC_m gives the ratio to the measured WCRT. Test cases and Ticks are the number of different scenarios and logical ticks that were executed, respectively.

be enough to rule out most unreachable paths of this kind. Also a finer grained analysis of which parts of parallel threads can be executed in the same instant could lead to better results. However, it is hard to do this analysis on the CKAG, instead we will use an interface algebra, presented in the next section, to put the analysis on a theoretical sound foundation.



(a) Esterel Program G

(b) Assembler Fragment G

(c) Control flow graph

(d) Some Interface Types

Figure 6.9.: Example program G

6.2. Interface Algebra

The WCRT analysis technique presented in the last section already provides fairly promising results. However, as noted in the previous section, this heuristics still makes conservative and simplifying assumptions and is not grounded in a formal timing model. To illustrate this, consider the small program G in Figure 6.9(a) and the corresponding assembler (b). The longest-path heuristic implemented in the strl2kasm compiler will compute a WCRT of 6. This, however, is overly conservative, as the longest path makes contradictory assumptions (signal I present and absent at the same time), similar to the example in Figure 6.6a. Furthermore, the WCRT algorithm is neither compositional nor scalable in terms of precision. They are global analysis on the complete and fully-expanded control-flow graph of a monolithic program and run at the ground level of atomic program statements rather than hierarchical sub-systems.

In this section a theory of WCRT interfaces for synchronous programming is proposed which (1) give precise statements about exactness and coverage of timing values, supporting a variety of timing abstractions, and (2) are dedicated to express the imperative synchronous programming languages. The interface can be employed to obtain a type-directed and modular WCRT analysis that is scalable across component hierarchies and the software-hardware abstraction boundary. As an interface theory the WCRT algebra operates on matrices of delay values characterizing whole sub-systems rather than individual nodes like the graph-theoretic WCRT algorithm that was presented in the last

section does. Like the propositional stabilization theory presented in [Mendler, 2000] it combines max-plus algebra $(\mathbb{N}, max, +, 0, -\infty)$ [Baccelli et al., 1992] with an intuitionistic refinement of Boolean logic to reason about implicit control-flow. For a complete explanation of the interface algebra see von Hanxleden et al. [2008] and Mendler et al. [2009].

6.2.1. The WCRT Algebra

The *interface type* of a program fragment is an implication $\phi \supset \psi$ between input controls $\phi = \bigvee_{i=1}^{m} \zeta_i$ and output controls $\psi = \bigoplus_{j=1}^{n} \circ\xi_j$. The input controls ϕ capture all the possible ways in which the program fragment can be started within an instant. It can contain both labels to code where the execution of the fragment starts and signal statuses that come from the environment. The output controls sum up the ways in which the fragment can be exited during the instant. The input controls are combined by $\bigvee$, because the environment of the fragment has to ensure that exactly one input control is active. The output controls are combined by $\bigoplus$, because they are resolved by the system, *i. e.*, the fragment must ensure that exactly one output control is active (see Mendler et al. [2009] for details). Intuitively, $\phi \supset \psi$ says that whenever any executions enters the program through one of the input controls ζ_i, then within some bounded number d_{ij} of instruction cycles all these executions are guaranteed to exit through one of the output controls ξ_j. The bounds d_{ij} may depend on the choice of input and output control, in general. To capture the bounds, we associate with each interface type a delay matrix of shape $n \times m$. The type specifications then become logical expressions of the form $D : \phi \supset \psi$ consisting of a timing matrix D together with an interface type $\phi \supset \psi$. The former describes the quantitative aspect of scheduling, the latter captures the qualitative part of the interface.

6.2.2. An Example

To illustrate the use of WCRT types consider again the small program G in Figure 6.9. Each node v_1–v_7 in the control-flow graph (c) of the associated Esterel program (a) is compiled into an assembler instruction (b) which is entered either sequentially through its instruction number L1–L7 or through an explicit jump to a control flow label such as G0–G3. For instance, node v_3 is accessed both through its linear instruction number L6 as well as by jump to its label G1. In contrast, node v_4 is only accessed through its line number L7 while node v_5 only by jumping to its label G3. The present nodes v_1 and v_3 are tests which branch to their two successor instructions depending on the status of signal I. If I is present then v_1 moves to instruction v_2 which immediately follows it, and if I is absent then v_1 passes control to instruction v_3 by jumping to label G1.

An interface which only considers the input $G0$ and computes the longest path through G is $(6) : G0 \supset \circ L11$. A full WCRT specification encapsulating program G as a component would require mention of program labels $G1$, $G3$, $G2$ which are accessible from outside for jump statements. Therefore, the interface type of G would be $(6, 4, 3, 1) : (G0 \vee G1 \vee G3 \vee G2) \supset \circ L11$. This is still not the most exact description of

G since it does not express the dependency of the WCRT on signal I. In particular, the longest path of length 6 from $G0$ to $L11$ is not executable. To capture this we consider signal I as just another control input and refine the WCRT scheduling type of G as follows: $(5,5,3,4,3,1) : ((G0 \wedge I) \vee (G0 \wedge \neg I) \vee (G1 \wedge I) \vee (G1 \wedge \neg I) \vee G3 \vee G2) \supset \circ L11$. The inclusion of signal I in the interface has now resulted in the distinction of two different delays 3 and 4 for $G1 \supset \circ L11$ depending on whether I is present or absent during the reaction. On the other hand, $G0$ split into controls $G0 \wedge I$ and $G0 \wedge \neg I$ produces the same delay of 5 instruction cycles in both cases, which is a decrease of WCRT compared to 6 from above. Assuming that input signal I is causally stable, *i. e.*, $I \oplus \neg I \cong \mathit{true}$, the two entries of value 5 can be merged into a single value as in $(5,3,4,3,1) : (G0 \vee (G1 \wedge I) \vee (G1 \wedge \neg I) \vee G3 \vee G2) \supset \circ L11$. In the same vein, we could further bundle $G1 \wedge I$ and $G3$ into a single input control $(G1 \wedge I) \oplus G3$ with delay 3. This finally gives $(5,3,4,1) : (G0 \vee ((G1 \wedge I) \oplus G3) \vee (G1 \wedge \neg I) \vee G2) \supset \circ L11$. Still, if we only ever intend to use G as a composite node from $G0$ to $L11$, the typing $(5) : G0 \supset \circ L11$, which takes care of signal dependency on I, might be sufficient.

All operations on interfaces and WCRT analyses are supported by semantically sound transformation rules in the WCRT type algebra. The logical manipulation of types often can be done implicitly and hard-coded into the graph-theoretic search strategies that make up the cleverness of a particular WCRT algorithm. Where interface types are not used directly in the calculations they provide for a highly compositional fine-grain analysis which allows us to validate WCRT algorithms in terms of precise statements about correctness and exactness. Due to their logical-symbolic nature WCRT interfaces can be applied in rather general situations which involve data and higher control-flow constructs as used in synchronous programming.

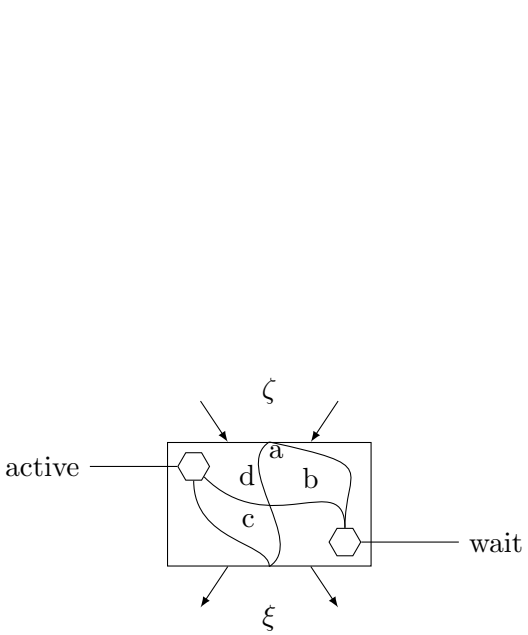
6.2.3. Classification of Interfaces

Figure 6.10 depicts a program fragment T abstracted into a reactive box with input and output controls. The paths inside T seen in Figure 6.10 illustrate the four ways in which a reactive node T may participate in the execution of a logical tick: Threads may (a) pass straight through the node entering at some input control ζ and exiting at output control ξ; (b) enter through ζ but pausing inside, waiting there for the next instant; (c) start the tick inside the node and eventually (instantaneously) leave through some exit control ξ, or (d) start inside the node and never leave it during the current instant. These paths or rather sections of a path are called *through paths*, *sink paths*, *source paths* and *internal paths*, respectively.

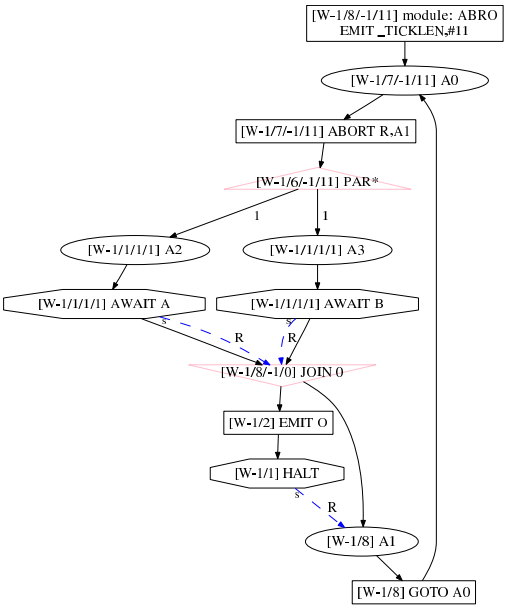
The interface type for such a node T (considering only one input control ζ and one output control ξ) separates these different paths and associated WCRT values:

$$T \quad = \quad \begin{pmatrix} d_{thr} & d_{src} \\ d_{snk} & d_{int} \end{pmatrix} : (\zeta \vee \mathit{active}) \supset (\circ\xi \oplus \circ \mathit{wait})$$

If one of the paths does not exist its associated delay is set to $-\infty$. A node T can be classified according to the paths that are executable in it. We define the (not necessarily



(a) The four types of thread paths: through path (a), sink path (b), source path (c), internal path (d).



(b) Annotated CKAG for ABRO generated by the analysis. The through, source, sink and internal WCRT is shown for each node. -1 denotes infeasible paths.

Figure 6.10.: Different types of thread paths

disjoint) sets of *through* nodes, $N_{thr} = \{T \mid d_{thr} \geq 0\}$, *source* nodes, $N_{src} = \{T \mid d_{src} \geq 0\}$, *sink* nodes, $N_{snk} = \{T \mid d_{snk} \geq 0\}$, and *internal* nodes, $N_{int} = \{T \mid d_{int} \geq 0\}$. A *delay node* is a node with at least one non-instantaneous path ($N_{del} = N_{src} \cup N_{snk} \cup N_{int}$). A *strong delay node* is a delay node without any through path ($N_{sdel} = N_{del} \setminus N_{thr}$). A *transient node* is a through node that contains only through paths, *i. e.*, $d_{src} = d_{snk} = d_{int} = -\infty$ ($N_{trans} = N_{thr} \setminus N_{del}$). Each cyclic dependency loop in the program must be broken by at least one strong delay node, which corresponds to the rule mentioned earlier that forbids instantaneous loops.

In general, the interface type of a program T will mention a number of controls $\zeta_1, \zeta_2, \ldots \zeta_m$ and $\xi_1, \xi_2, \ldots, \xi_n$ on the input and output side for which the type would be

$$T = D : (\zeta_1 \vee \zeta_2 \cdots \vee \zeta_m) \supset (\circ\xi_1 \oplus \circ\xi_2 \oplus \cdots \oplus \circ\xi_n) \tag{6.1}$$

with a WCRT matrix D of shape $n \times m$. A composite program will be made up of a number of program fragments T_i each with its interface $D_i : \phi_i \supset \psi_i$. The total specification is the logical conjunction $\bigwedge_i D_i : \phi_i \supset \psi_i$ in WCRT type algebra. The basic controls appearing in ϕ_i, ψ_i describe the causal dependencies between the nodes T_i. In

Esterel Module	Measured WCRT_m	Graph WCRT_g	Graph $\Delta_{g/m}$	Interface WCRT_i	Interface $\Delta_{i/m}$
abro	11	11	0%	11	0%
channel protocol	22	41	86%	36	63%
runner	16	18	13%	16	0%
traffic light	13	15	15%	14	8%
schizo	10	11	10%	11	10%

Figure 6.11.: Comparison between the WCRT computed by graph based approach (WCRT_g) and using the interface algebra (WCRT_i) on a selected set of benchmarks. WCRT_m is the measured Worst case reaction time. The ratio $\Delta_{e/m} := WCRT_e/WCRT_m$ for $e = g$ and $e = m$ gives the overestimation for both analysis.

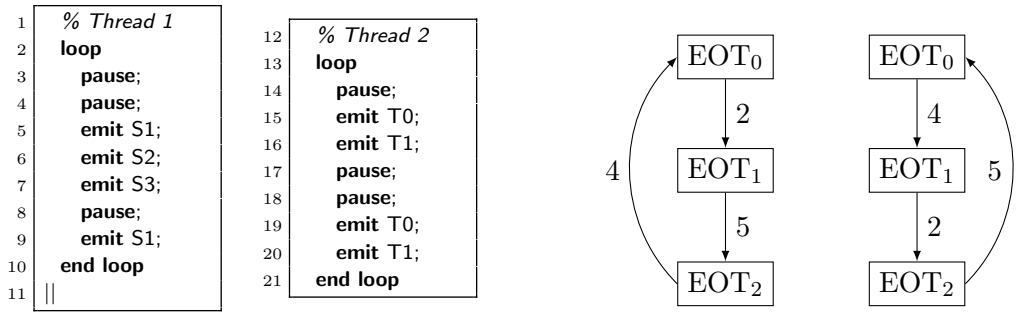
its general form, WCRT analysis amounts to a transformation

$$\bigwedge_i D_i : \phi_i \supset \psi_i \quad \preceq \quad D : \phi \supset \psi \tag{6.2}$$

in which the individual timing interfaces D_i are combined into a total delay matrix D for an external interface $\phi \supset \psi$ such that D is the smallest (component-wise) matrix of values such that (6.2) holds. The external interface $\phi \supset \psi$ determines the functional precision with which we are computing the WCRT of a composite system. For instance, instead of an interface like (6.1), which distinguishes m input and n output controls, a less discriminative type $\zeta \supset \circ\xi$ with $\zeta =_{df} \bigvee_{i\in I} \zeta_i$ and $\xi =_{df} \circ \bigoplus_{j\in J} \xi_j$ might consider merely subsets $I \subseteq \{1, \ldots, m\}$ and $J \subseteq \{1, \ldots, n\}$ of inputs and outputs bundled into a single control. Such an interface $\zeta \supset \circ\xi$, which specifies only one delay value is more abstract than (6.1). Of course, we do not expect to get an equivalence $\cong$ but only an inclusion $\preceq$ in (6.2) if the calculation of D involves timing abstractions. We can trade off precision and efficiency of the WCRT analysis within wide margins by choosing different types $\phi_i \supset \psi_i$ for the components and $\phi \supset \psi$ for the composite program in (6.2). By logical transformations of interfaces, various optimizations can be achieved including such as those employed by classic combinational timing analysis [Mendler, 2000].

6.2.4. Implementation

To evaluate the approach, I prototypically implemented some of the key ideas. Blocks are identified with threads and compute the through, source, sink and internal WCRT for each thread independently. All outgoing transitions from a thread are abstracted into one. Like the graph based approach, the analysis is implemented on top of the CKAG. Figure 6.10b shows the CKAG for Abro annotated with through, source, sink and internal WCRT values that is generated by the analysis. A -1 denotes infeasible paths, *e. g.*, for the main node neither a through nor an exit value can be specified. The



(a) Esterel Code

(b) Coressponding TFSM

Figure 6.12.: Motivating example WCRT analysis based on model checking

results for some test-cases can be found in Figure 6.11. Since the approach does not consider traps yet, we had to replace traps by local signals and weak abortion. This is trivial for these examples. In general, the transformation can be done analogously to the hardware synthesis from Esterel [Potop-Butucaru et al., 2007]. The transformation is also the reason why the computed WCRT differs from the results reported in the last section.

This limited implementation already leads to improvements over the graph based analysis presented in the last section in some of the tested example cases, shown in Figure 6.11. Still, the analysis is not as exact as it could be.The interface approach so far does not distinguish between immediate and delayed aborts. The implementation could also be improved, *e. g.*, by unbundling outgoing thread transitions and other heuristics. The theory could be further strengthened, *e. g.*, by directly integrating abortion in the control flow graph.

6.3. Using Model-Checking

While the interface algebra presented in the last section gives good means to express WCRT properties of synchronous programs, and also leads to an efficient algorithm, there are still some drawbacks. In particular, the efficiency of the algorithm depends on the used heuristics: when can we combine information? When should we abstract data? In particular, when combining the WCRT of parallel threads, we take the sum of the maximal WCRT properties to get the combined property. This is pessimistic, since the maximum execution for the threads might not be reachable in the same instance, or they could not be reachable with the same active signals. While both dependencies can be expressed in the WCRT algebra, such a detailed analysis might not scale.

Consider the simple Esterel program in Figure 6.12a. Figure 6.12b shows the finite state machines for both threads, where the computation needed for each tick between time delimiting instructions (end of tick, *EOT*), is attached to the transitions. These finite state machines are called Timed Finite State Machines (*TFSMs*). Note that the loop is implemented by a goto whose execution takes an additional instruction cycle.

```
1  signal cnt:=0: integer,
2         pre_cnt : integer in
3  %pre
4    var v:= 0: integer in
5      loop
6        emit pre_cnt(v);
7        v:= ?cnt;
8        pause;
9      end loop
10   end var
11 ||
12 % sampler
13   signal sample:=0:integer in
14   var i :=0:integer in
15     pause;
16     i:= init(N);
17     loop
18       emit sample(?sens);
19       pause;
20       trap T in
21         loop
22           if ?cnt<N then
23             exit T
24           end if;
25           pause
26         end loop
27       end trap;
28       emit WriteBuf(i);
29       emit WriteVal(?sample);
30       pause;
31       i := i+1;
32       if i=N then i:=0; end if
33     end loop
34   end var
35   end signal
36 ||
37 % display
38   signal out:=0:integer in
39   var i: integer in
40     pause;
41     loop
42       trap T in
43         loop
44           if ?cnt>0 then
45             exit T
46           end if;
47           pause
48         end loop
49       end trap;
50       emit WriteVal(i);
51       emit out(?ReadVal);
52       pause;
53       i := i+1;
54       if i=N then i:=0; end if;
55       emit cnt(?pre_cnt+1);
56       pause;
57       emit WriteLcd(?out)
58     end loop
59   end var
60   end signal
61 end signal
```

Figure 6.13.: A producer consumer example in Esterel.

Both the implementation based on the longest path in the CKAG and on the interface algebra compute a maximal tick length of 11, by adding the maximal reaction times of the threads $(5+5)$ and one additional instruction cycle to execute the join. However, in this example it is obvious that the maximal tick length of both threads will never occur in the same instant, hence the "correct" WCRT is 10 $(4+5+1)$.

The WCRT analysis of a synchronous program is equivalent to the model checking question to compute this greatest fixed point. In the context of reactive processing, model checking was first used to determine the WCRT of PRET-C or Precision Timed C programs, a synchronous extension of the C language, by Roop et al. [2009a]. Here we show that their approach can also be applied for computing the WCRT of a *KEP* assembler program. The approach consists of three steps: First, the program (or the corresponding CKAG) is transformed into a Timed Concurrent Control Flow Graph (*TCCFG*), which is further transformed into flat TFSMs, from which timed automata as input to the model checker UPPAAL[2] are derived. A detailed introduction to PRET-C including its semantics is given by Andalam et al. [2009].

We will use the producer consumer program in Figure 6.13 as a running example. This example is similar to the one used by Roop et al. [2009a] zo demonstrate the timing analysis of PRET-C. The program was slightly adjusted, because for the original program, model checking gives no benefit over the analysis from Section 6.1: The longest execution of the producer occurs in the first instant, while the longest execution of the consumer occurs in a later tick. This special case is already handled by the WCRT analysis implemented in the strl2kasm compiler and is also handled by the interface

[2] `www.uppaal.org`

algebra that distinguishes between sink and internal paths.

Since the method was designed for PRET-C, it does not directly support the various forms of preemption that are possible in *KEP* assembler. While these could be integrated, another possibility is to combine this approach with the interface algebra: model-checking is used to get a detailed analysis of parallel threads, which is then used to get a timing interface for the complete block. Possible abortions can then be handled by the interface algebra.

In the next section the intermediate TCCFG format is introduced, from which the TFSMs and finally the timed automata are derived. On these the actual model checking is performed.

Timed Concurrent Control Flow Graph

The TCCFG is a control flow graph similar in spirit to the CKAG or the CCFG of Edwards and Zeng [2007]. Figure 6.14b shows the *KEP* assembler for the producer consumer example together with the generated TCCFG. While this is an Esterel example, it contains none of the problems stated above. The TCCFG was directly derived from the CKAG generated by the strl2kasm compiler by removing label nodes and combining nodes with a unique successor into one node.

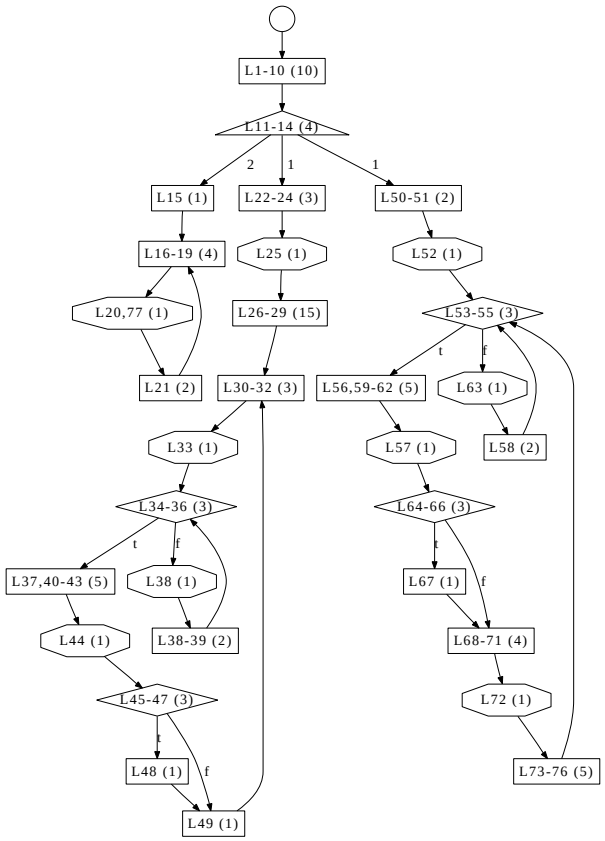
A TCCFG has the following types of nodes:

- Start/end node: Every TCCFG has a unique start node where the control begins and may have an end node, if the program can terminate. These nodes are drawn as concentric circles.
- Fork/join nodes: These are needed to mark concurrent threads of control and where these threads start and end. These are drawn as triangles. The thread priority is annotated to the outgoing transition of the fork.
- Action nodes: These are used for any C function call or data computation. They are denoted by rectangles.
- EOT nodes: These nodes indicate a local end of tick. We denote them, like the pause nodes in the CKAG, as octagons.
- Control flow nodes: There are two types of control flow nodes: conditional nodes to implement conditional branching (denoted by a rhombus) and jump nodes for mapping unconditional branches (which are needed to emulate infinite loops).
- Node weights: Each node is annotated with the cost of the node. This value represents the exact number of clock cycles needed to execute the assembler instructions for that node.

It is quite easy to spot that the TCCFG is a faithful model of the control flow of the original source and is a one-to-one mapping of the source code into a graph code format, each node in the TCCFG is annotated with the corresponding lines in the *KEP* assembler.

```
1        SETV SENS,#0
2        SETV WRITEBUF,#0
3        SETV WRITEVAL,#0
4        SETV READBUF,#0
5        SETV READVAL,#0
6        SETV WRITELCD,#0
7        SIGNALV CNT
8        SETV CNT,#0
9        SIGNALV PRE_CNT
10       SETV PRE_CNT,#0
11       PAR 2,A0,2
12       PAR 1,A1,1
13       PAR 1,A2,3
14       PARE A3,1
15  A0:  LOAD V,#0
16  A4:  EMIT PRE_CNT,V
17       PRIO 1
18       LOAD V,?CNT
19       PRIO 2
20       PAUSE
21       GOTO A4
22  A1:  SIGNALV SAMPLE
23       SETV SAMPLE,#0
24       LOAD I,#0
25       PAUSE
26       LOAD _TMP0,#10
27       CALL INIT
28       LOAD REG0,_TMP0
29       LOAD I,REG0
30  A5:  EMIT SAMPLE
31       LOAD REG0,?SENS
32       SETV SAMPLE,REG0
33       PAUSE
34  A7:  LOAD REG0,?CNT
35       CMPS REG0,#10
36       JW L,A8
37       EXIT T,A7
38  A8:  PAUSE
39       GOTO A7
40  T:   EMIT WRITEBUF,I
41       EMIT WRITEVAL
42       LOAD REG0,?SAMPLE
43       SETV WRITEVAL,REG0
44       PAUSE
45       ADD I,#1
46       CMPS I,#10
47       JW EE,A9
48       LOAD I,#0
49  A9:  GOTO A5
50  A2:  SIGNALV OUT
51       SETV OUT,#0
52       PAUSE
53  A12: LOAD REG1,?CNT
54       CMPS REG1,#0
55       JW G,A13
56       EXIT T_0,A12
57  A13: PAUSE
58       GOTO A12
59  T_0: EMIT WRITEVAL,I_0
60       EMIT OUT
61       LOAD REG1,?READVAL
62       SETV OUT,REG1
63       PAUSE
64       ADD I_0,#1
65       CMPS I_0,#10
66       JW EE,A14
67       LOAD I_0,#0
68  A14: EMIT CNT
69       LOAD REG1,?PRE_CNT
70       ADD REG1,#1
71       SETV CNT,REG1
72       PAUSE
73       EMIT WRITELCD
74       LOAD REG1,?OUT
75       SETV WRITELCD,REG1
76       GOTO A12
77  A3:  JOIN 0
```

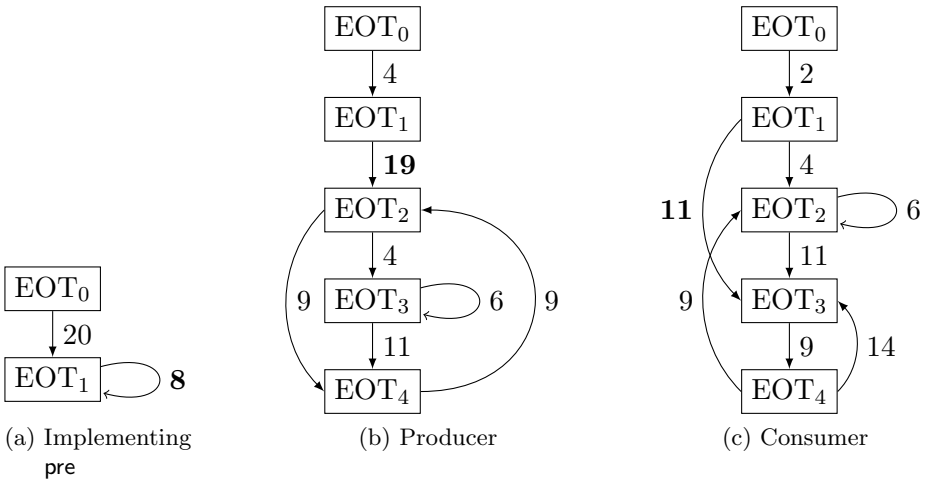
(a) KEP assembler



(b) Timed Concurrent Control Flow Graph (TTCFG)

Figure 6.14.: KEP assembler for the producer consumer example and the corresponding TTCFG. Each node is annotated with the corresponding assembler lines and the number of instruction cycles that are needed to execute them

Timed Finite State Machines



(a) Implementing pre (b) Producer (c) Consumer

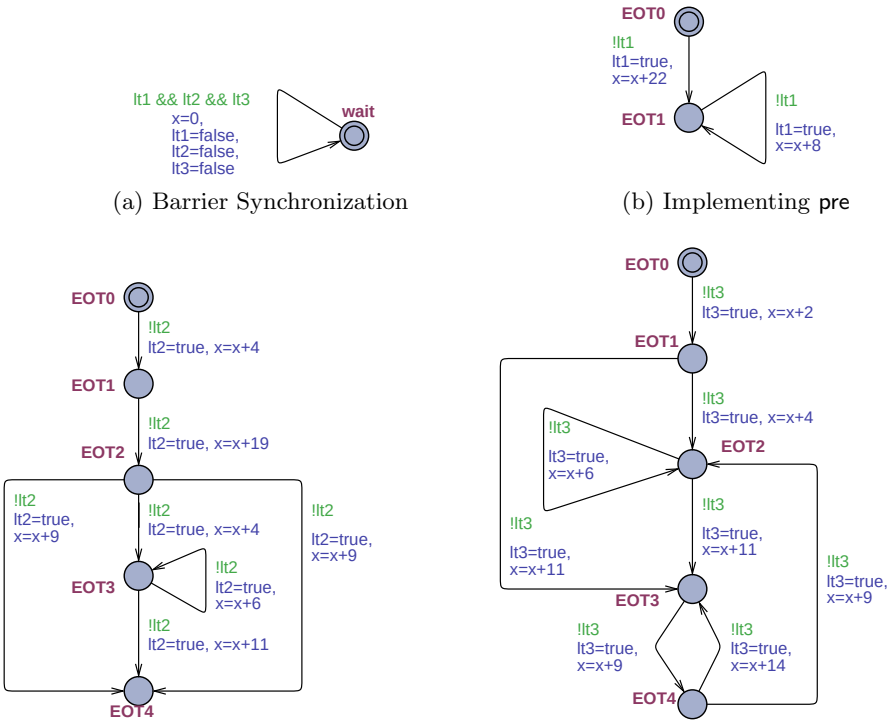
Figure 6.15.: Timed Finite State Machines (*TFSMs*) for the producer consumer example. The maximal execution times that can occur within one tick are marked bold.

The TCCFG can be transformed into a set of timed automatas. An upper and a lower bound of the WCRT can be computed by adding the maximal and minimal reaction times on parallel thread, similar as it is done in Section 6.1 and 6.2. On this range, a binary search is performed to get the minimal x, for which the execution time at the end of a tick is always smaller than x.

For illustration, we first map the TCCFG to an equivalent TFSM. The TFSM corresponding to the two threads of the producer consumer TCCFG of Figure 6.14b is shown in Figure 6.15. This mapping is done by a depth first search from every EOT node to all EOT nodes that are reachable from this node. During the traversal, the cost of every node is simply added to obtain the total cost between these two EOTs, where the cost of the fork and join are added to the first spawned thread. For example, the cost of the edge between EOT_0 and EOT_1 of the producer is 4 clock cycles, which is obtained by adding the costs of all the nodes between these two ticks. The cost of an individual node is obtained by combining the individual costs of the nodes in the CKAG. The initialization costs are added to the first thread and the costs for the JOIN instruction are added to the third thread. The next step is the mapping of the TFSM to a timed automata.

Timed Automata

Timed automata, proposed by Alur and Dill [1994], extend finite state machines with real-valued clocks such that transition guards can be based on these clocks. All clocks progress synchronously. These clocks may be treated like other programming variables



(a) Barrier Synchronization (b) Implementing pre

(c) Producer (d) Consumer

Figure 6.16.: Timed Automata for the producer consumer example

in the sense that they can be read and written. In addition, they can be compared and common arithmetic operators can be applied to them. Clock conditions can be used as transition guards where the only logical operators that can be used are $<$, $\leq$, $=$, $\geq$, $>$. A clock variable or the difference between two clock variables can be compared against a natural number to form a given clock constraint. A global system then consists of a network of timed automata such that these timed automata communicate asynchronously. Communication between timed automata can be done using shared variables or using channels (which are either point to point or broadcast). There are several differences between timed automata and TFSMs and their compositions: 1) timed automata use dense clocks while the transitions of TFSMs are guarded by integers representing the execution cost. Hence, there is no need to use any clock variables in the timed automaton model; a simple integer to capture the cost of a transition is sufficient. 2) The composition between TFSMs is strictly synchronous while timed automaton compositions are asynchronous. An additional timed automaton, called a *barrier*, is added to realize the synchronous semantics. Basically, the barrier node is a master node that starts a new tick as soon as all threads have finished. The mapping is illustrated in Figure 6.16 using the same producer consumer example as shown in Figure 6.15.

The overall mapping is achieved by mapping each TFSM to an equivalent timed au-

tomaton. The global variable x captures the cost of a complete tick of the program. The Boolean variables lt1, lt2, and lt3 capture if a given thread has completed its *local tick*.

The *barrier* (Figure 6.16a) has just one state. It waits until lt1, lt2, lt3 have been set to true by the corresponding automata, *i. e.*, a global tick is started since all threads have finished their local ticks. In response to this, it starts a new tick by setting all local tick variables to false again. It also initializes the costs of the current tick to 0.

Each timed automaton contains the same states and transitions as the corresponding TFSM. For a transition $[EOT_i] \xrightarrow{d} [EOT_j]$, the corresponding transition in the timed automaton has the label $[EOT_i] \xrightarrow{\frac{\neg lt}{lt=true, x=x+d}} [EOT_j]$ The guard waits for the activation of a new tick by the barrier, *i. e.*, for its local tick variable to be false. In the transition action, the local tick variable of the automaton is set to true, hence deactivating all other transitions until the next step of the barrier takes place. Furthermore, the reaction time that are associated with the transition are added to the global costs in x.

WCRT as a Model Checking Property

We can compute the WCRT of the program by model checking the property of the form $A\Box(\bigwedge_i lt_i \Rightarrow x \leq val)$, whenever the global tick happens, the combined execution times for all threads are below a constant value val. The $WCRT_{tight}$ is the minimal value for which this property holds. The tight WCRT value, $WCRT_{tight}$ lies between the $WCRT_{min}$ and $WCRT_{max}$ values. Both $WCRT_{max}$ and $WCRT_{min}$ can be obtained by summing up the maximal and minimal local tick values for each thread. For example, in the producer consumer case, the $WCRT_{min} = 8 + 6 + 2$ and $WCRT_{max} = 20 + 19 + 14$. Hence, $WCRT_{tight}$ has a value in the interval $[16, 54]$. Standard binary search can be used to minimize the number of queries. For example, to obtain the tight value for the producer consumer case, we have to write at most 6 queries ($log_2(54 - 16)$). In the producer consumer case, the tight value obtained by the above analysis is $38 = 8{+}19{+}11$, compared to 41 with the graph based approach or the interface algebra, where the sum of the maximum of all threads is computed.

The complexity of the proposed WCRT analysis is $O((WCRT_{max} - WCRT_{min}) \times |M| \times |\phi|)$ for checking a single query. In the worst case there are $log_2(WCRT_{max} - WCRT_{min})$ queries necessary for the binary search, hence the overall complexity is $O(log_2(WCRT_{max} - WCRT_{min}) \times (WCRT_{max} - WCRT_{min}) \times |M| \times |\phi|)$.

6.4. Comparison

Now we have seen three different methods to determine the WCRT of a reactive program. Table 6.17 gives a short overview of the differences of the approaches. While the graph based approach is the fastest, it lacks support for data-handling and a more fine grained analysis of thread interactions. Both the interface algebra and the model-checking allow a tight analysis, but this is obviously not efficient for the interface algebra. Here, is the possibility to apply different heuristics to trade performance vs. exactness of the analysis.

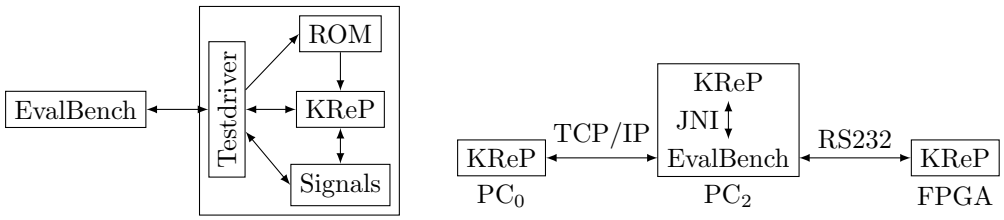
	Graph	Algebra	Model Checking
Complexity	$O(\|N\|^2)$	Depends on heuristic	$O(log_2(WCRT_{range}) \times WCRT_{range} \times \|M\| \times \|\phi\|)$
Concurency	max	max /precise	precise
Consider Data	No	Yes	Yes
Implementation	complete (KEP)	partly (KEP)	complete (PRET-C)

Figure 6.17.: Comparison of the different aprroaches for WCRT analysis

While the search for the longest path is an approach that is closely coupled to the *KEP* and its compiler, the model-checking approach and the interface algebra can be easily combined. The model checking, which has a higher complexity, can be used to get a tight WCRT of subsystems, the combination of these is then performed using the interface algebra.

7. Evaluation with KIELER

To evaluate the reactive processors, we need to execute programs on them and must be able to set inputs and read the generated outputs. We also need the information which instructions were executed and how long the execution took. For this purpose, both the *KEP* and the *KLP* were extended by a *test-driver*, which allows to load programs to the internal ROM, to set signals and to start a tick. The test-driver can also send, on request, the status and values of signals, the reaction time for the last tick, and the execution trace. To do this the test drivers of the *KEP* and the *KLP* communicate with an evaluation bench. Figure 7.1 shows the communication between the KReP Evalbench and the *KEP* or the *KLP*.



(a) Extension with a test driver

(b) Communication Paths

Figure 7.1.: Communication between the Evalbench and the KEP/KLP

The original *KEP EvalBench* was designed by Xin Li for the *KEP*. It was a monolithic visual basic program that communicated with the *KEP* over an RS232 connection. It could either execute programs stepwise with manually set inputs and show the execution trace by directly marking the *KEP* assembler. Or it could execute trace files in the eso format of Esterel Studio and compare the generated output to the reference outputs in the trace file. By running it in a batch mode, it was possible to automatically evaluate a benchmark suite and to generate reports.

With the implementation of the *KEP-e*, there was the possibility for software simulation of the *KEP*, hence the RS232 connection was not flexible enough. Another disadvantage of the original EvalBench was that it could only run on Windows platforms. Therefore, the first evaluations of the *KEP-e* were performed by running a software simulation of the *KEP-e* on an computer with Linux, which was connected via RS232 to a Windows PC running the EvalBench. To overcome these problems, the KReP Evalbench was developed as an Eclipse rich client application[1]. It could directly communicate with the software emulation, and it could automatically execute benchmark suites directly

[1] `wiki.eclipse.org/index.php/Rich_Client_Platform`

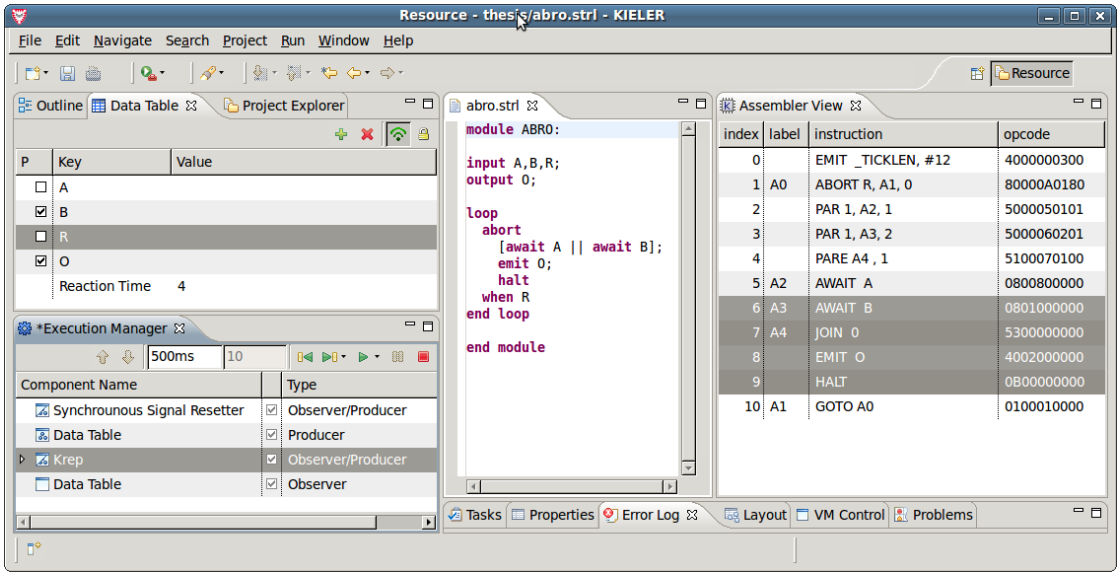


Figure 7.2.: Execution of Esterel on the KEP within KIELER.

without a batch mode. Due to the modular design, it could be easily extended, *e. g.*, by the protocol for the *KLP*. It could read *KEP* or *KLP* assembler as an input, but also SyncCharts which were then automatically compiled by smakc!.

Since the KReP Evalbench is an Eclipse application, it was naturally to merge it with the Kiel Integrated Environment for Layout for the Eclipse Rich Client Platform[2] (*KIELER*) project. This project focuses on the modelling of complex systems, *e. g.*, by developing new means to build graphical models, and on the dynamic visualization of the systems. The KReP Evalbench is now part of KIELER. Additionally, the strl2kasm compiler was integrated into the tool, so that the user can write Esterel programs and execute them on the *KEP* within one tool, a s shown in Figure 7.2. In the middle of the tool the Esterel program ABRO is shown. The currently executed assembler instructions are marked in the assembler view on the right. Input and output signals, as well as the reaction time for the last tick are displayed in the data table on the left. The execution is controlled by the execution manager below the data table. Three components are used for the execution: the *Synchronous Signal Resetter* makes sure that signals are absent per default, the *Data Table* is used to set inputs and to display outputs, and the *Krep* component is responsible for the compilation and the communication with the *KLP* and the *KEP*.

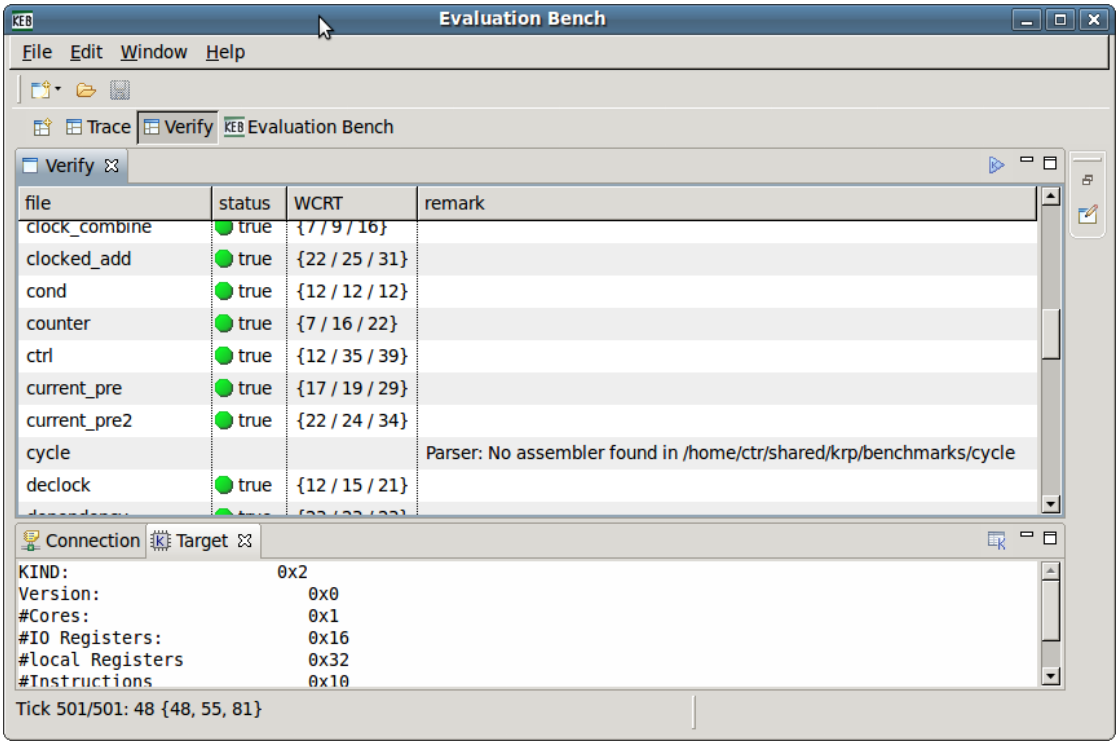


Figure 7.3.: Automatic execution of a benchmark suite.

7.1. Execution Modes

The KReP Evalbench can be run in three different modes:

- In the *normal mode*, which is shown in Figure 7.2, the user can directly set inputs, perform a step, and the outputs as well as the executed assembler lines of the program are shown.

- In the *trace mode*, an input trace in either rif or eso format is run with the current program. For each step, the generated output is compared to the reference output and the inputs and outputs are shown to the user, as well as the information whether the behaviors match.

- In the *verify mode*, all programs and traces in a given folder are executed, and an overview is shown to the user, which programs were executed successfully with the expected behavior and which failed and why. Also the best, average, and worst case reaction times are measured for each example in the benchmark. This mode is shown in Figure 7.3. Here, all executions succeeded, but for the cycle example,

[2] www.informatik.uni-kiel.de/rtsys/kieler/

Action	KReP Evalbench → *KLP*	*KLP* → KReP Evalbench
verify (V)	56	29 3A 39 38 37 36 35 34 33 32 31 30
info (I)	49	2 *Version* 1 *#IO #REG #ROM #ALU*
reset (R)	52	
write (W)	57 *PC* I_0 I_1 I_2 I_3	FF
set value (S)	53 *ID* V_0 V_1 V_2 V_3	FF
get value (G)	47 *ID*	V_0 V_1 V_2 V_3
tick (T)	54	*tick length*
trace (E)	45	*trace*
run (R)	43	
halt (H)	48	

Figure 7.4.: Communication protocol of the KLP. All values are hexadecimal

no *KLP* assembler was generated. Note that the status is only set to false if the behavior differs. This is the original verify mode of the KReP Evalbench, which is not connected to the specific KIELER features. In the future, KIELER will be extended by a more generic extension for automatic testing, which will replace the specific verify mode of the KReP Evalbench.

7.2. Communication

The *KReP* communicates with the KReP Evalbench via the RS232 when run on an FPGA. The software emulation can either communicate via TCP/IP or it can directly be embedded into the KReP Evalbench via the Java Native Interface (*JNI*). For the communication via JNI, where the KReP Evalbench triggers each instruction cycle of the *KLP*, also a trace of the communication is saved in the esi format. This can then be used to simulate the behavior in Esterel Studio to see the detailed behavior of the *KLP*. The possible communication paths between the KReP Evalbench and the *KReP* are shown in Figure 7.1b.

KEP protocol The *KLP* and the *KEP* use different protocols for the communication. The *KEP* protocol only sends ASCII characters, so to send the value 0x42, the two bytes 0x34 and 0x32 are transmitted to the *KEP*. This has the advantage that the protocol can be directly written by the developer. And the protocol uses end markers, so to send a program to the *KEP*, the ASCII encoding of the opcode is sent, followed by the ASCII code of "X". Unfortunately, the protocol does not implement acknowledgements for all transactions. A full description of the protocol is given by Li [2007] and Tiedje [2008].

KLP protocol The protocol of the *KLP* is byte oriented. The first byte of each command indicates which action is performed and how many bytes are following. For every transaction, there is an explicit acknowledgement, therefore no end marker is needed.

All communication is triggered by the KReP Evalbench. Figure 7.4 gives an overview over the protocol. The actions are:

verify: verifies that the communication works by sending back a constant byte sequence.

info: read the information on the configuration of the *KLP*, such as the number of IO connections, the number of registers, the size of the ROM and the number of processing units.

reset: reset the *KLP* completely.

write: write one instruction of four bytes to the ROM.

set value: set the value of one register.

get value: read the value of one register.

tick: start a tick. The number of clock cycles that were needed for the execution is returned.

trace: get the trace of the last tick. The size of the returned byte sequence depends on the size of the ROM: for each address, it is marked whether this instruction was executed. *E. g.* returning a 5 indicates that the first and the third instruction were executed.

run: starts running the *KLP* in a *free mode*, *i. e.*, ticks are not triggered by the KReP Evalbench and only external inputs from the environment are read. In this mode, the only actions that are accepted by the *KLP* are reset and halt.

halt: stop the *KLP* from running freely and return to the normal mode, where inputs can be set by the KReP Evalbench and all communication actions are allowed.

8. Conclusion and Outlook

8.1. Conclusion

The execution on the *KEP* has many benefits compared to the compilation of Esterel into software: smaller programs, ofter faster execution, in particular for complex nesting of preemptions, and, combined with the WCRT analysis, a constant reaction time without jitter. The translation from SyncCharts to *KEP* assembler shows that the *KEP* can not only be used to execute Esterel code, but also other synchronous languages. The direct translation showed to be more efficient than the exiting translation via Esterel, in particular due to the direct support of GOTO, which is essential for the efficient execution of SyncCharts. The formal semantics for the *KEP* is a first step into proving the correctness of the compilation. Since the translation of Esterel into *KEP* assembler is rather simple compared to the compilation of Esterel into efficient code for common processors, showing the correctness of the compilation process should be much simpler as well. But this needs a formal and accurate semantics for the *KEP*. While the semantics given in this thesis defines the behavior of *KEP* assembler for a relevant subset of the *KEP* instruction set, there are still some aspects missing. The semantics so far does not handle the execution times of the *KEP*,which would be necessary to prove the correctness of the WCRT analysis.

For the *KLP*, we can conclude that the approach is already promising, but both the hardware description and the compiler should be further improved. But we also see that a reactive benefit for a dataflow language like Lustre does not give the same benefit as for Esterel, since the efficient execution of Lustre on a common processor is much simpler than the of Esterel. The *KLP* introduces parallel execution into the domain of reactive processors, which before either expressed concurrency by multi-threading, as for the *KEP* or the StarPro, or uses parallel processors, as in the EMPEROR, which does not support full Esterel. On the other hand, the parallel execution for Lustre is much simpler than for Esterel, because the only form of preemption is suspension. For the compilation from Scade, which supports abortion, the abortion is not run in parallel, but checked at the start and at the end of parallel code blocks.

We originally chose Esterel for the description of the *KLP* and the *KEP*-e for two reasons: to improve the maintainability of the *KEP* and to evaluate Esterel capabilities for hardware design, because we only used it for software synthesis before. While the experiments show that even programmers without hardware knowledge can generate hardware of reasonable efficiency with Esterel, the design of optimal hardware still requests both hardware knowledge and a detailed understanding of the hardware generation from Esterel.

One of the main benefits of reactive processors, in particular of the *KEP*, is that they simplify the WCRT analysis compared to common processors. We presented three different approaches for the WCRT analysis of the *KEP*:

1. a graph based approach that is directly implemented in the compiler. This approach determines the longest path on the intermediate format of the compiler. It makes some overly conservative approximations regarding parallel execution and data dependencies.

2. The interface algebra gives a flexible framework for WCRT analysis, depending on the chosen heuristics, it can be used for a precise analysis or different levels of approximation.

3. The model checking approach developed for PRET-C allows for a precise analysis, but it does not integrate too well with the preemption mechanism of the *KEP*, where any instruction can trigger a preemption.

8.2. Outlook

Reactive Processors The current development in the area of reactive processing is less specific hardware, like in patched approach of the PRET-C or SyncCharts in C, which is completely implemented in software. Similarly, it would be interesting to see how the approach of the *KLP* can be applied to common hardware, like multi-processor machines or also GPUs. The main difference here is that the *KLP* can execute a different instruction for each register, while common GPUs execute the same instruction on all cores. Hence the *KLP* is a *Multiple Instruction Multiple Data* (MIMD) processor, while GPUs are usually *Single Instruction Multiple Data* (SIMD) processors. In general, it would be interesting to see how the ideas of reactive processing can be applied to standard processor, as it is done by the SC, which implements the instruction set as C Macros, or the PRET-C, which applies only small modifications to existing processors. Another interesting question is whether existing reactive processors can be used for other languages than they were originally intended for, such as the execution of Scade models on the *KEP*.

Timing Analysis While the timing analysis of the *KEP* already gives good results, there are still some parts missing. The interface algebra needs to be extended to delayed preemption, and it would be interesting to implement and compare different heuristics. The model checking approach could also be implemented for the *KEP* and combined with the interface algebra. The WCRT analysis for the *KLP* still needs to be implemented.

The WCRT for the *KLP* differs from the one for the *KEP*, in that the *KLP* has much less control structures. Since the only form of preemption that is directly supported by the *KLP* is suspension, we do not need to compute possible continuations. Hence, for a *KLP* with one processing unit, a simple counting algorithm already gives good results,

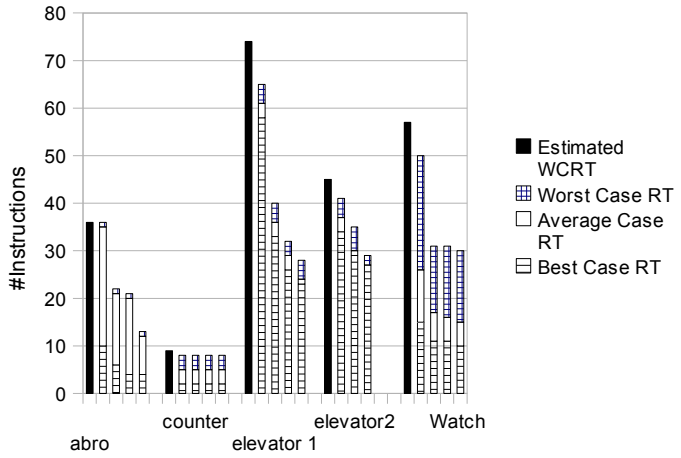


Figure 8.1.: Estimated as well as minimal, average and maximal reaction times (rt) for different numbers of Alus

even when taking the rough abstraction that the longest path for each equation is possible simultaneously. Of course, this might not be the case due to data-dependencies.

Figure 8.1 shows the computed WCRT compared to the measured one. For each example, five bars are shown: the dark bar on the left gives the estimated WCRT, the other bars give the number of instructions that are actually executed at runtime. While the estimated number of instructions is a good approximation of the execution on a *KLP* with just one ALU, it naturally overestimates the execution time for multiple Arithmetic Logical Units (ALUs).

A simple approach for the computation of the WCRT for a *KLP* with multiple ALUs is the following: As long as the priorities are not changed during the execution, we can compute the WCRT for each priority independently. To consider the worst possible scheduling, we divide the priorities into N sets, when computing the WCRT for N ALUs. An estimation for the complete WCRT is the sum of the WCRT for all priorities.

Consider we have a program with the following equations with priority 0:

```
1  A = I*I * 2*I;    // WCRT=3
2  B = I+J/2;        // WCRT=3
3  C = I * J;        // WCRT=2
```

And with priority 1 we have the equations:

```
1  O = max(B,C)      // WCRT=4
2  P = B+C;          // WCRT=2
```

The WCRT per equation takes also the DONE instruction into account. Hence we get the following WCRT sets per priority: $0 : \{3, 3, 2\}$ and priority $1 : \{4, 1\}$. If we compute this on two ALUs we get the partition $\{3, 3\}$ and $\{2\}$ for priority 0; for priority 1 we get the trivial partition $\{4\}$ and $\{1\}$. The combined WCRT is $3 + 3 + 4 = 10$.

The *KLP* already brings true parallel execution to reactive processing, but only with a very close connection between the parallel execution units. It is still an open question,

how to combine ideas from multi-core processors, with a rather loose coupling between the different processing units, with the different forms of preemption that are supported by reactive processing. In particular, the efficient execution of strong abortion combined with immediate communication between concurrent threads within one tick, while strictly adhering the synchronous semantics, is not trivial.

A. Benchmarks

A.1. ABRO

The ABRO mimics the behavior of the Esterel ABRO module by implementing the automaton. Each state is implemented by an boolean variable, a state is true if it was true in the previous tick and no outgoing transition is triggered, or if an incoming transition is triggered.

```
1  node abro(a, b, r: bool) returns (o: bool);
2  var
3    s_ab, -- wait for a and b
4    s_a,  -- wait for a
5    s_b,  -- wait for b
6    s_0: bool; -- wait for r
7  let
8    s_ab = true → r or (pre(s_ab) and not (a or b));
9    s_a  = false → not r and ((pre(s_a) and not a) or (pre(s_ab) and b and not a));
10   s_b  = false → not r and ((pre(s_b) and not b) or (pre(s_ab) and not b and a));
11   s_0  = false → not r and (pre(s_0)
12                             or (pre(s_a) and a)
13                             or (pre(s_b) and b)
14                             or (pre(s_ab) and a and b));
15   o    = false→not pre(s_0) and s_0;
16 tel.
```

A.2. Counter

A simple counter of rising edges, which can be reseted:

```
1  node counter (R, X: bool) returns (C: int);
2  let
3    C = 0 → if R then 0
4                 else if X and not pre X
5                      then pre(C)+1
6                      else pre(C);
7  tel
```

A.3. Elevator Lustre

The controller of a simple elevator:

```
node elevator_lus (ButtonUp, ButtonDown, ButtonAlarm,
                   IsUp, IsDown, Second:bool)
returns (Move:int; AlarmLamp:bool)
var LampCount: int;
    LastDirection : int;
let
  LastDirection = if ButtonAlarm then pre(Move)
                                 else pre( LastDirection );
  LampCount = -1 -> if ButtonAlarm
                    then 5
                    else if pre(LampCount) < 0
                         then -1
                         else if Second
                              then pre(LampCount) - 1
                                   else pre(LampCount);
  AlarmLamp = LampCount > 0;
  Move = if LampCount = 0
                then LastDirection
         else if not ButtonAlarm
                      and ((ButtonUp and not IsUp) or (pre(Move)=1 and not IsUp))
              then 1
                    else if not ButtonAlarm
                            and((ButtonDown and not IsDown)
                                or (pre(Move)=-1 and not IsDown))
                    then -1
                    else 0;
tel
```

A.4. Elevator Scade

A controller for the same elevator, but generated from Scade:

```
node elevator_chsch(
    ButtonUp : bool;
    ButtonDown : bool;
    ButtonAlarm : bool;
    IsUp : bool;
    IsDown : bool;
    Second : bool)
  returns (Move : int; AlarmLamp : bool);
var
  down : bool;
  secCounter : int;
  moveDir : int;
  _L4 : int;
  _L11 : bool;
  _L12 : bool;
  _L13 : bool;
  _L14 : bool;
  _L16 : bool;
  _L19 : int;
  _L20 : int;
```

```
21      _L21 : int;
22      _L26 : int;
23      _L28 : bool;
24      _L29 : bool;
25      _L32 : int;
26      _L33 : bool;
27      _L35 : int;
28      _L36 : int;
29      _L37 : int;
30      _L34 : int;
31      _L41 : int;
32      _L42 : bool;
33      _L43 : int;
34      _L44 : bool;
35      _L45 : bool;
36      _L46 : bool;
37    let
38      down= true → pre(_L13);
39      _L4= 0;
40      _L32= if _L11 then _L4 else _L34;
41      _L11= IsDown or IsUp;
42      _L12= IsDown or IsUp;
43      _L13= if _L12 then _L14 else down;
44      _L14= IsDown;
45      _L16= moveDir <> 0 and ButtonAlarm;
46      secCounter= 0 → pre(_L26);
47      _L26= if _L16 then _L19 else _L20;
48      _L19= 5;
49      _L20= if _L28 then _L21 else secCounter;
50      _L21= secCounter − 1;
51      _L28= Second and secCounter <> 0;
52      _L29= moveDir = 0 and (ButtonUp and down or ButtonDown and not down);
53      _L33= ButtonUp and down;
54      _L35= if _L33 then _L36 else _L37;
55      _L36= 1;
56      _L37= −1;
57      _L34= if _L29 then _L35 else moveDir;
58      moveDir= 0 → pre(_L32);
59      _L41, _L44= if _L42 then (_L43, _L45) else (moveDir, _L46);
60      _L42= secCounter <> 0;
61      _L43= 0;
62      AlarmLamp= _L44;
63      Move= _L41;
64      _L45= true;
65      _L46= false;
66    tel;
```

A.5. Watch

The implementation of a simple watch, which counts seconds and minutes.

```
1  node watch(time_unit: bool) returns (s,m:bool; ds,dm: int);
2  let
3    (s,ds) = COUNT(time_unit, m, 10);
4    (m,dm) = COUNT(s,false,60);
```

```
5  tel;
6
7  node COUNT(trigger, reset: bool; g: int) returns (x: bool; d: int;);
8  let
9    x = EDGE(current(DIVIDE(g when (true → trigger))));
10   d = current((0→if reset then 0 else (pre(d) + 1)) when (x or reset));
11 tel
12
13 node EDGE(b: bool) returns (edge: bool);
14 let
15   edge = b→(b and not pre(b));
16 tel;
17
18 node DIVIDE (scale: int) returns (quotient: bool);
19 var n: int;
20 let
21   (n,quotient) = (0,true)→(if (pre(n) + 1) = scale
22                            then (0, true)
23                            else (pre(n) + 1, false));
24 tel;
```

A.6. Parallel

Parallel computations without data dependencies.

```
1  node parallel (X,Y:int) returns (O1,O2,O3,O4:int);
2  let
3    O1=0;
4    O2=1;
5    O3=2;
6    O4=3;
7  tel
```

Bibliography

Jauhar Ali and Jiro Tanaka. Converting Statecharts into Java code. In *Proceedings of the Fourth World Conference on Integrated Design and Process Technology (IDPT '99)*, Dallas, Texas, June 2000. Society for Design and Process Science (SDPS).

Rajeev Alur and David L. Dill. A theory of timed automata. *Theoretical Computer Science*, 126:183–235, 1994.

Sidharta Andalam, Partha Roop, Alain Girault, and Claus Traulsen. PRET-C: A new language for programming precision timed architectures. Technical Report 6922, INRIA Grenoble Rhône-Alpes, 2009. `http://hal.inria.fr/docs/00/39/16/21/PDF/rr.pdf`.

C. André, F. Boulanger, M.-A. Péraldi, J. P. Rigault, and G. Vidal-Naquet. Objects and synchronous programming. *Europ. J. on Automated Syst.*, 31(3):417–432, 1997.

Charles André. Semantics of S.S.M (Safe State Machine). Technical report, Esterel Technologies, Sophia-Antipolis, France, April 2003. `http://www.esterel-technologies.com`.

L. Arditi, G. Berry, J. Dormoy, L. Blanc, S. Granier, and M. Perreaut. Generating efficient hardware with Esterel v7 and Esterel Studio, April 2005.

F. L. Baccelli, G. Cohen, G. J. Olsder, and J.-P. Quadrat. *Synchronisation and Linearity*. John Wiley & Sons, 1992.

Felice Balarin, Massimiliano Chiodo, Paolo Giusto, Harry Hsieh, Attila Jurecska, Luciano Lavagno, Alberto Sangiovanni-Vincentelli, Ellen M. Sentovich, and Kei Suzuki. Sythesis of Software Programs for Embedded Control Applications. In *IEEE Transactions of Computer-Aided Design of Integrated Circuits and System*, volume 18, pages 834–849, June 1999.

J. Benkoski and A. J. Strojwas. Timing verification by formal signal interaction modeling in a multi-level timing simulator. In *Design Automation Conference*, pages 668–673, 1989.

Albert Benveniste, Paul Le Guernic, and Pascal Aubry. Compositionality in dataflow synchronous languages: Specification and code generation. In Willem P. de Roever, Hans Langmaack, and Amir Pnueli, editors, *COMPOS*, volume 1536 of *Lecture Notes in Computer Science*, pages 61–80. Springer, 1997.

Albert Benveniste, Paul Caspi, Stephen A. Edwards, Nicolas Halbwachs, Paul Le Guernic, and Robert de Simone. The Synchronous Languages Twelve Years Later. In *Proceedings of the IEEE, Special Issue on Embedded Systems*, volume 91, pages 64–83, January 2003.

Christoph Berg, Jakob Engblom, and Reinhard Wilhelm. Requirements for and design of a processor with predictable timing. In Lothar Thiele and Reinhard Wilhelm, editors, *Perspectives Workshop: Design of Systems with Predictable Behaviour*, number 03471 in Dagstuhl Seminar Proceedings. Internationales Begegnungs- und Forschungszentrum für Informatik (IBFI), Schloss Dagstuhl, Germany, 2004. `http://drops.dagstuhl.de/opus/volltexte/2004/5`.

Gérard Berry. The foundations of Esterel. *Proof, Language and Interaction: Essays in Honour of Robin Milner*, 2000. Editors: G. Plotkin, C. Stirling and M. Tofte.

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

Gérard Berry. Esterel on Hardware. *Philosophical Transactions of the Royal Society of London*, 339:87–104, 1992.

Gérard Berry. *The Constructive Semantics of Pure Esterel*. Draft Book, 1999. `ftp://ftp-sop.inria.fr/esterel/pub/papers/constructiveness3.ps`.

Gérard Berry and Laurent Cosserat. The ESTEREL Synchronous Programming Language and its Mathematical Semantics. In *Seminar on Concurrency, Carnegie-Mellon University*, volume 197 of *Lecture Notes in Computer Science (LNCS)*, pages 389–448. Springer-Verlag, 1984. ISBN 3-540-15670-4.

Gérard Berry and Ellen Sentovich. Multiclock esterel. In *CHARME '01: Proceedings of the 11th IFIP WG 10.5 Advanced Research Working Conference on Correct Hardware Design and Verification Methods*, pages 110–125, London, UK, 2001. Springer-Verlag.

Darek Biernacki, Jean-Louis Colaco, Grégoire Hamon, and Marc Pouzet. Clock-directed Modular Code Generation of Synchronous Data-flow Languages. In *ACM International Conference on Languages, Compilers, and Tools for Embedded Systems (LCTES)*, Tucson, AZ, USA, June 2008.

Marian Boldt. Worst-case reaction time analysis for the KEP3. Study thesis, Christian-Albrechts-Universität zu Kiel, Department of Computer Science, May 2007a. `http://rtsys.informatik.uni-kiel.de/~biblio/downloads/theses/mabo-st.pdf`.

Marian Boldt. A compiler for the Kiel Esterel Processor. Diploma thesis, Christian-Albrechts-Universität zu Kiel, Department of Computer Science, December 2007b. `http://rtsys.informatik.uni-kiel.de/~biblio/downloads/theses/mabo-dt.pdf`.

Marian Boldt, Claus Traulsen, and Reinhard von Hanxleden. Worst case reaction time analysis of concurrent reactive programs. *Electronic Notes in Theoretical Computer Science*, 203(4):65–79, June 2008. Proceedings of the International Workshop on Model-Driven High-Level Programming of Embedded Systems (SLA++P'07), March 2007, Braga, Portugal.

A. Bouajjani, J.-C. Fernandez, N. Halbwachs, P. Raymond, and C. Ratel. Minimal state graph generation. *Sci. Comput. Program.*, 18(3):247–269, 1992.

J. Le Boudec and P. Thiran. *Network Calculus - A theory of deterministic queuing systems for the internet*, volume 2050 of *Lecture Notes in Computer Science*. Springer, 2001.

Alan Burns and Stewart Edgar. Predicting computation time for advanced processor architectures. In *Proceedings of the 12th Euromicro Conference on Real-Time Systems (EUROMICRO-RTS 2000)*, page 89, 2000.

Paul Caspi, Alain Girault, and Daniel Pilaud. Automatic distribution of reactive systems for asynchronous networks of processors. *IEEE Transactions on Software Engineering*, 25(3), 1999.

Etienne Closse, Michel Poize, Jacques Pulou, Patrick Venier, and Daniel Weil. SAXO-RT: Interpreting Esterel semantic on a sequential execution structure. In Florence Maraninchi, Alain Girault, and Eric Rutten, editors, *Electronic Notes in Theoretical Computer Science*. Elsevier, July 2002. `http://www.elsevier.com/gej-ng/31/29/23/117/53/34/65.5.010.pdf`.

Jean-Louis Colaço, Bruno Pagano, and Marc Pouzet. A conservative extension of synchronous data-flow with State Machines. In *ACM International Conference on Embedded Software (EMSOFT'05)*, Jersey City, NJ, USA, September 2005.

S. Devadas, K. Keutzer, and S. Malik. Delay computation in combinational logic circuits: Theory and algorithms. In *International Conference on Computer-Aided Design*, pages 176–179, 1991.

Edsger W. Dijkstra. GOTO considered harmful. *Communications of the ACM*, 11(3): 147–148, March 1968.

Stephen A. Edwards. An Esterel compiler for large control-dominated systems. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 21(2), February 2002.

Stephen A. Edwards. CEC: The Columbia Esterel Compiler, 2006. `http://www1.cs.columbia.edu/~sedwards/cec/`.

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

Frederic Boussinot. Reactive C: An extension of C to program reactive systems. *Software Practice and Experience*, 21(4):401–428, 1991.

Sascha Gädtke, Claus Traulsen, and Reinhard von Hanxleden. HW/SW Co-Design for Esterel Processing. In *Proceedings of the International Conference on Hardware-Software Codesign and System Synthesis (CODES+ISSS'07)*, Salzburg, Austria, September 2007.

Alain Girault. A survey of automatic distribution method for synchronous programs. In F. Maraninchi, M. Pouzet, and V. Roy, editors, *International Workshop on Synchronous Languages, Applications and Programs (SLAP '05)*, Electronic Notes in Theoretical Computer Science, Edinburgh, UK, April 2005. Elsevier Science.

Alain Girault and Xavier Nicollin. Clock-driven automatic distribution of Lustre programs. In R. Alur and I. Lee, editors, *3rd International Conference on Embedded Software, EMSOFT '03*, volume 2855 of *Lecture Notes in Computer Science (LNCS)*, pages 206–222, Philadelphia, PA, USA, October 2003. Springer-Verlag.

Zonghua Gu. Solving real time scheduling problems with model-checking. In *Embedded Software and Systems*, volume 3820 of *Lecture Notes in Computer Science*, pages 186–197. Springer Berlin / Heidelberg, 2005.

Paul Le Guernic, Thierry Goutier, Michel Le Borgne, and Claude Le Maire. Programming real time applications with SIGNAL. *Proceedings of the IEEE*, 79(9), September 1991.

J. R. Gurd, C. C. Kirkham, and I. Watson. The Manchester prototype dataflow computer. *Communications of the ACM*, 1985.

Olivier Hainque, Laurent Pautet, Yann Le Biannic, and Eric Nassor. Cronos: A separate compilation toolset for modular Esterel applications. In Jeannette M. Wing, Jim Woodcock, and Jim Davies, editors, *World Congress on Formal Methods*, volume 1709 of *Lecture Notes in Computer Science*, pages 1836–1853. Springer, September 1999.

Nicolas Halbwachs. A synchronous language at work: the story of Lustre. In *Third ACM-IEEE International Conference on Formal Methods and Models for Codesign, MEMOCODE'05*, Verona, Italy, July 2005.

Nicolas Halbwachs. Synchronous programming of reactive systems, a tutorial and commented bibliography. In *Tenth International Conference on Computer-Aided Verification, CAV '98*, Vancouver (B.C.), June 1998. LNCS 1427, Springer Verlag.

Nicolas Halbwachs and Pascal Raymond. *A Tutorial of Lustre*, 2001.

Nicolas Halbwachs, Paul Caspi, Pascal Raymond, and Daniel Pilaud. The synchronous data-flow programming language LUSTRE. *Proceedings of the IEEE*, 79(9):1305–1320, September 1991a.

Nicolas Halbwachs, Pascal Raymond, and Christophe Ratel. Generating efficient code from data-flow programs. In J. Maluszyński and M. Wirsing, editors, *Proceedings of the Third International Symposium on Programming Language Implementation and Logic Programming*, pages 1–13207–218. Springer Verlag, 1991b.

D. Harel and A. Pnueli. On the development of reactive systems. *Logics and models of concurrent systems*, pages 477–498, 1985.

David Harel, Hagi Lachover, Amnon Naamad, Amir Pnueli, Michal Politi, Rivi Sherman, Aharon Shtull-Trauring, and Mark Trakhtenbrot. Statemate: A working environment for the development of complex reactive systems. *IEEE Transactions on Software Engineering*, 16(4):403–414, April 1990.

Th. Henzinger and S. Matic. An interface algebra for real-time components. In *Proc. RTAS 2006*, pages 253–266. IEEE Computer Society, 2006.

Cornelis Huizing and Rob Gerth. Semantics of reactive systems in abstract time. In *Proceedings of the Real-Time: Theory in Practice, REX Workshop*, pages 291–314. Springer-Verlag, 1992. ISBN 3-540-55564-1.

Erwan Jahier, Pascal Raymond, and Philippe Baufreton. Case studies with lurette v2. *International Journal on Software Tools for Technology Transfer*, 8(6), November 2006.

Lei Ju, Bach Khoa Huynh, Abhik Roychoudhury, and Samarjit Chakraborty. Performance debugging of Esterel specifications. In *CODES+ISSS*, pages 173–178, 2008.

Gilles Kahn. The semantics of a simple language for parallel programming. In Jack L. Rosenfeld, editor, *Information Processing 74: Proceedings of the IFIP Congress 74*, pages 471–475. IFIP, North-Holland Publishing Co., August 1974.

K. C. Lam and R. K. Brayton. *Timed Boolean Functions. A Unified Formalism for Exact Timing Analysis*. Kluwer, 1994.

E. A. Lee, H. Zheng, and Y. Zhou. Causality interfaces and compositional causality analysis. In *Foundations of Interface Technologies (FIT'05)*, ENTCS. Elsevier, 2005.

Edward A. Lee and David G. Messerschmitt. Synchronous data flow. In *Proceedings of the IEEE*, volume 75, pages 1235–1245. IEEE Computer Society Press, September 1987.

Xin Li. *The Kiel Esterel Processor: A Multi-Threaded Reactive Processor*. PhD thesis, Christian-Albrechts-Universität zu Kiel, Faculty of Engineering, July 2007. `http://eldiss.uni-kiel.de/macau/receive/dissertation_diss_00002198`.

Xin Li and Reinhard von Hanxleden. Multi-threaded reactive programming—the Kiel Esterel Processor. *IEEE Transactions on Computers*, accepted 2010.

Xin Li, Jan Lukoschus, Marian Boldt, Michael Harder, and Reinhard von Hanxleden. An Esterel Processor with Full Preemption Support and its Worst Case Reaction Time Analysis. In *Proceedings of the International Conference on Compilers, Architecture, and Synthesis for Embedded Systems (CASES'05)*, pages 225–236, San Francisco, CA, USA, September 2005. ACM Press. ISBN 1-59593-149-X. doi: http://doi.acm.org/10.1145/1086297.1086327.

Ben Lickly, Isaac Liu, Sungjun Kim, Hiren D. Patel, Stephen A. Edwards, and Edward A. Lee. Predictable programming on a precision timed architecture. In *Proceedings of Compilers, Architectures, and Synthesis of Embedded Systems (CASES'08)*, Atlanta, USA, October 2008.

G. Logothetis and Klaus Schneider. Exact high level WCET analysis of synchronous programs by symbolic state space exploration. In *Design, Automation and Test in Europe (DATE)*, pages 196–203, Munich, Germany, March 2003. IEEE Computer Society.

G. Logothetis, K. Schneider, and C. Metzler. Exact low-level runtime analysis of synchronous programs for formal verification of real-time systems. In *Forum on Design Languages (FDL)*, Frankfurt, Germany, 2003. Kluwer.

Jan Lukoschus and Reinhard von Hanxleden. Removing cycles in Esterel programs. *EURASIP Journal on Embedded Systems, Special Issue on Synchronous Paradigms in Embedded Systems*, 2007.

Avinash Malik, Zoran Salcic, and Partha S. Roop. Systemj compilation using the tandem virtual machine approach. *ACM Trans. Des. Autom. Electron. Syst.*, 14(3):1–37, 2009. ISSN 1084-4309.

Sharad Malik. Analysis of cyclic combinational circuits. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 13(7):950–956, July 1994.

Sharad Malik, Margaret Martonosi, and Yau-Tsun Steven Li. Static timing analysis of embedded software. In *DAC '97: Proceedings of the 34th annual conference on Design automation*, pages 147–152. ACM Press, 1997. ISBN 0-89791-920-3. `http://doi.acm.org/10.1145/266021.266052`.

M. Mendler. Characterising combinational timing analyses in intuitionistic modal logic. *The Logic Journal of the IGPL*, 8(6):821–853, 2000.

Michael Mendler, Reinhard von Hanxleden, and Claus Traulsen. WCRT Algebra and Interfaces for Esterel-Style Synchronous Processing. In *Proceedings of the Design, Automation and Test in Europe (DATE'09)*, Nice, France, April 2009.

A. Metzner. Why model checking can improve WCET analysis. In *Computer Aided Verification*, volume 3114 of *Lecture Notes in Computer Science*, pages 334–347. Springer Berlin / Heidelberg, 2004.

David Park, Jens U. Skakkebaek, Mats P.E. Heimdahl, Barbara J. Czerny, and David L. Dill. Checking properties of safety critical specifications using efficient decision procedures. In *Proceedings of the Second ACM Workshop on Formal Methods in Software Practice*, Clearwater Beach, Florida, March 1998.

Becky Plummer, Mukul Khajanchi, and Stephen A. Edwards. An Esterel virtual machine for embedded systems. In *International Workshop on Synchronous Languages, Applications, and Programming (SLAP'06)*, Vienna, Austria, March 2006.

Dumitru Potop-Butucaru and Robert de Simone. *Optimization for faster execution of Esterel programs*, pages 285–315. Kluwer Academic Publishers, Norwell, MA, USA, 2004. ISBN 1-4020-8051-4.

Dumitru Potop-Butucaru, Stephen A. Edwards, and Gérard Berry. *Compiling Esterel*. Springer, May 2007. ISBN 0387706267.

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

P. Puschner and A. Burns. A review of worst-case execution-time analysis (editorial). *Real-Time Systems*, 18(2/3):115–128, 2000.

Thomas Ringler. Static worst-case execution time analysis of synchronous programs. In *ADA-Europe- 5. International Conference on Reliable Software Technologies*, 2000.

Partha S. Roop, Sidharta Andalam, Reinhard von Hanxleden, Simon Yuan, and Claus Traulsen. Tight WCRT analysis for synchronous C programs. Technical Report 0912, Christian-Albrechts-Universität Kiel, Department of Computer Science, Kiel, Germany, May 2009a.

Partha S. Roop, Sidharta Andalam, Reinhard von Hanxleden, Simon Yuan, and Claus Traulsen. Tight WCRT analysis for synchronous C programs. In *Proceedings of the International Conference on Compilers, Architecture, and Synthesis for Embedded Systems (CASES'09)*, Grenoble, France, October 2009b.

Klaus Schneider. The synchronous programming language Quartz. Internal Report (to appear), Department of Computer Science, University of Kaiserslautern, Kaiserslautern, Germany, 2009. `http://es.cs.uni-kl.de/publications/datarsg/Schn09.pdf`.

Martin Schoeberl. A Java processor architecture for embedded real-time systems. *Journal of Systems Architecture*, 54(1–2), 2008.

J. P. M. Marques Silva and K. A. Sakallah. An analysis of path sensitization criteria. In *Proc. ICCD*, pages 68–72, 1993.

Falk Starke. Executing Safe State Machines with the Kiel Esterel Processor. Diploma thesis, Christian-Albrechts-Universität zu Kiel, January 2009. `http://rtsys.informatik.uni-kiel.de/~biblio/downloads/theses/fast-dt.pdf`.

Falk Starke, Claus Traulsen, and Reinhard von Hanxleden. Executing Safe State Machines on a reactive processor. Technical Report 0907, Christian-Albrechts-Universität Kiel, Department of Computer Science, Kiel, Germany, March 2009.

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

Olivier Tardieu and Stephen A. Edwards. Instanteneous transitions in esterel. In *Proceedings of Model Driven High-Level Programming of Embedded Systems (SLA++P'07)*, Braga, Portugal, March 2007.

Malte Tiedje. Beschreibung des Kiel Esterel Prozessors in Esterel. Diploma thesis, Christian-Albrechts-Universität zu Kiel, Department of Computer Science, January 2008. `http://rtsys.informatik.uni-kiel.de/~biblio/downloads/theses/mti-dt.pdf`.

Malte Tiedje and Claus Traulsen. Designing a reactive processor with Esterel v7. In *Proceedings of the Workshop on Model-Driven High-Level Programming of Embedded Systems (SLA++P'08)*, Budapest, Hungary, April 2008.

Claus Traulsen. The Kiel Reactive Processor—reactive processing beyond the KEP. Presentation at the 14th International Open Workshop on Synchronous Programming (SYNCHRON'07), Bamberg, Germany, November 2007.

Claus Traulsen and Reinhard von Hanxleden. Reactive parallel processing for synchronous dataflow. In *Proceedings of the 25th Symposium On Applied Computing (SAC'10), Special Track Embedded Systems: Applications, Solutions, and Techniques*, Sierre, Switzerland, March 2010.

Claus Traulsen, Jerome Cornet, Matthieu Moy, and Florence Maraninchi. A SystemC/TLM semantics in Promela and its possible applications. In *Proceedings of the 14th Workshop on Model Checking Software (SPIN '07)*, Berlin, Germany, July 2007.

Reinhard von Hanxleden. SyncCharts in C. Technical Report 0910, Christian-Albrechts-Universität Kiel, Department of Computer Science, May 2009. `http://rtsys.informatik.uni-kiel.de/~biblio/downloads/papers/report-0910.pdf`.

Reinhard von Hanxleden, Michael Mendler, and Claus Traulsen. WCRT algebra and scheduling interfaces for Esterel-style synchronous multi-threading. Technical Report 0807, Christian-Albrechts-Universität Kiel, Department of Computer Science, June 2008.

E. Wandeler and L. Thiele. Real-time interfaces for interface-based design of real-time systems with fixed priority scheduling. In *Proc. EMSOFT'05*, 2005.

Andrzej Wasowski. On efficient program synthesis from Statecharts. In *Proceedings of the 2003 ACM SIGPLAN Conference on Language, Compilers, and Tools for Embedded Systems (LCTES'03)*, volume 38, issue 7, June 2003. ACM SIGPLAN Notices.

Reinhard Wilhelm, Jakob Engblom, Andreas Ermedahl, Niklas Holsti, Stephan Thesing, David Whalley, Guillem Bernat, Christian Ferdinand, Reinhold Heckmann, Frank Mueller, Isabelle Puaut, Peter Puschner, Jan Staschulat, and Per Stenström. The worst-case execution-time problem—overview of methods and survey of tools. *ACM Transactions on Embedded Computing Systems (TECS)*, 7(3), 2008.

Li Hsien Yoong, Partha Roop, and Zoran Salcic. Compiling Esterel for distributed execution. In *Proceedings of Synchronous Languages, Applications, and Programming (SLAP'06)*, Vienna, Austria, April 2006.

Li Hsien Yoong, Partha Roop, Valeriy Vyatkin, and Zoran Salcic. A synchronous approach for iec 61499 function block implementation. *IEEE Trans. Comput.*, 58(12): 1599–1614, 2009. ISSN 0018-9340.

Simon Yuan, Sidharta Andalam, Li Hsien Yoong, Partha S. Roop, and Zoran Salcic. STARPro—a new multithreaded direct execution platform for Esterel. In *Proceedings of Model Driven High-Level Programming of Embedded Systems (SLA++P'08)*, Budapest, Hungary, April 2008.

Acknowledgments

This thesis would not have been possible without the help of many people. I thank my "Doktorvater" *Reinhard von Hanxleden* for giving me the opportunity to graduate, for his suggestions and advices, for giving me the freedom to choose and to change my research topic and for encouraging me to keep my topic in the end. I especially thank him for introducing me to the synchronous community and give me the possibility to travel to the annual Synchron meetings and for a research visit at the Verimag lab.

I am very grateful to the students who contributed to the work on reactive processing, these are *Marian Boldt*, *Sascha Gädtke*, *Falk Starke* and *Malte Tiedje*, it was a joy working with you. This thesis would not exist without *Xin Li's* previous work on the Kiel Esterel Processor.

I would in particular like to thank my colleagues *Hauke Fuhrmann* and *Hagen Peters* for their technical, personal, and moral support both at work and beyond. *Miro Spönemann* and *Christian Motika* helped me by removing much of the teaching and administrative tasks in the final phase of my dissertation and by supporting me with to connect of reactive processing with KIELER. I would also like to thank *Tim Grebien* for technical and for not bothering me in the end of my writing phase (by the way, clean up our room).

The contact to other colleagues in from the institute helped me to keep contact to the research in other areas of computer science and to recognize common problems. Therefore, I would like to thank the "Informatik Stammtisch": *Jan Christiansen*, *Sebastian Fischer*, *Fabian Reck*, and *Björn Lüdemann*, as well as, *Jan Täubrich*, *Heiko Schmidt*, and *Jens Schönborn*. Various students from the KIELER team helped to connect reactive processing to their tool, in particular I thank *Christian Schneider* for always fixing the build after I broke it, as well as *Torsten Amende* and *Achim Bleidiessel*.

My work on reactive processing was nicely supported by the synchronous community. Therfore I thank in particular *Michael Mendler*, *Partha Roop*, and *Sidharta Andalam* for various fruitful discussions, as well as *Alain Girault*. I learned a lot on synchronous languages and research during my visit at the Verimag laboratory. For the warm welcome there and and the succesful collaboration I thank *Jérôme Cornet*, *David Stauch*, *Florence Maraninchi*, *Matthieu Moy* and the other members of the synchronous team.

Last but not least I appreciate the help of my family to connect me to the world outside academia. I thank in particular my wife *Imke*, for the constant support and for helping me to actually finish this thesis.