Christian-Albrechts-Universität zu Kiel

Diploma Thesis

# Esterel Compiler for a Synchronous Reactive Processor

Marian Boldt

2007-12-18

Department of Computer Science
Real-Time and Embedded Systems Group

Advised by:
Prof. Dr. Reinhard von Hanxleden
Dipl.-Inf. Claus Traulsen

## Eidesstattliche Erklärung

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

Kiel,

## Abstract

The synchronous language Esterel is well-suited for programming control-dominated reactive systems. It provides non-traditional control structures, in particular concurrency and various forms of preemption, which allow to concisely express reactive behavior. As these control structures cannot be mapped easily onto traditional, sequential processors, an alternative approach that has emerged recently makes use of special-purpose reactive processors.

The *Kiel Esterel Processor* (KEP) is a synchronous reactive processor. It supports exception handling and provides context-dependent preemption handling instructions.

This thesis presents a compiler from *Esterel to KEP assembler* (strl2kasm). The translation of most Esterel statements into KEP assembler is straightforward, especially preemption constructs can be mapped directly. However, it is not trivial to implement the concurrency of Esterel. Concurrency is mapped to multiple threads, yet the compiler must ensure that all data and control dependencies are fulfilled. A priority assignment approach is presented that makes use of an intermediate graph structure, namely the *Concurrent KEP Assembler Graph* (CKAG), that is used to represent the control flow behavior of the KEP assembler program. It is also used to compute and store the concurrent program dependencies that are needed for the priority assignment. The complexity of the assignment algorithm for most programs is linear in the program size. Unlike earlier Esterel compilation schemes, this approach avoids unnecessary context switches by considering actual execution state at run time for each thread. Furthermore, it avoids code replication present in other approaches by implementing consequently the *Write Things Once* (WTO) principle.

# Contents

# List of Figures

# 1 Introduction

*Embedded systems* get more and more popular and today appear in most electrical devices. Thus, it has become an important task to create reliable software for these systems. Since their control flow is often non-standard, languages and processors were designed to implement their behavior explicitly. The properties of such systems and their differences to common applications are now discussed. Therefore, systems in general are considered, which can be divided into three main categories [5]:

- *Transformational Systems*: Computation of an output result from a given input argument, *e. g.*, numerical computations or interpreter/compiler. These systems are also denoted as *standard control flow* applications.
- *Interactive Systems*: The environmental interaction is triggered by the system itself, *i. e.*, the response time affects not the result, *e. g.*, databases and operating systems.
- *Reactive Systems*: These react continuously to inputs from the environment by generating corresponding outputs timely. The result depends on the response time.

Most systems and their programs consist of sub-systems and program parts that belong to different categories. Many embedded systems belong to the class of *reactive systems*. A further definition for reactive systems is given by Edwards *et al.* [18]:

- *Criticality*: They are highly critical, *e. g.*, time-critical, safety-critical.
- *Parallelism*: Distributed components operate often in parallel to achieve the desired behavior.
- *Determinism*: This is obviously desired for critical systems.

To ensure short response times in time-critical systems, it is recommended to support *preemptive* control flow.

The implementation of reactive systems with all the above requirements is no easy task, because the programming paradigms of standard programming languages, like C/C++, are designed to address standard control flow, applicable to implement transformational systems. In particular, a timing model is not a standard language construct and therefore, it has to be implemented by hand. This is a significant obstacle for the implementation of time-critical systems. This leads to a higher complexity on the implementation side, which makes it more difficult. Concurrency is as well hard to express, it is often implemented in a non-deterministic or non-efficient way.

To address these difficulties, the reactive and synchronous languages, like Signal [25], Lustre [26] and Esterel [9, 7], have been developed. They are designed to express reactive control directly in its syntax. These provide a discrete and abstract time model that orders the execution into a chain of ticks. The execution at these ticks is assumed to be instantaneous [10]. Here we consider the synchronous language Esterel, that is both parallel and deterministic. It has a formal mathematical semantics that imposes a strong deterministic program behavior and allows the programmers to develop critical software faster and better.

There exist different strategies to implement an Esterel program. The hardware approach generates combinational circuits from Esterel, *e. g.*, expressed in the hardware description languages VHDL or Verilog [4]. The implementation results in an efficient implementation, but is not flexible because a change in the Esterel source requires a complete re-synthesis. The software approach translates Esterel to a non-synchronous language like C to be executed on standard COTS hardware [10, 16]. This solution is flexible, but the generated code might be slow because standard hardware is not designed to match reactive control flow. Both principles can be combined to a hybrid approach—the hardware-software co-design [3].

Another alternative for synthesizing Esterel has emerged, the *reactive processing* approach. The Esterel program is running on a processor, here the *Kiel Esterel Processor* (KEP) [27], that has been developed specifically for reactive systems. The instruction set of these reactive processors closely resembles the constructs found in Esterel, such as waiting for occurrence of a signal or abortion. Reactive processors provide direct hardware support for reactive control flow, which keeps executables fast and compact and results in lower power consumption compared to traditional architectures.

Apart from efficiency and determinism concerns, another advantage of reactive processors is that due to their comparatively simple structure, *i. e.*, no caches and no pipe lining, and their direct implementation of reactive control flow constructs, it becomes feasible to precisely characterize their timing behavior [39]. In conjunction with the synchronous model of computation, which discretizes time into logical ticks, it is thus possible to derive exact, tight bounds on its *Worst Case Reaction Time* (WCRT) [11], which tells how much time it takes the system to react to the environment. This is important for time-critical reactive systems. The KEP is equipped with a Tick Manager that can provide a constant logical tick rate and also detects internal timing overruns. This can serve to detect hardware failures, reduce jitter, and provides another safeguard—in addition to static analysis—that real-time deadlines are met.

This thesis presents the *Esterel to KEP assembler* (strl2kasm) compiler. The strl2kasm bridges the gap between Esterel and the KEP assembler. This makes it possible to write programs in Esterel and execute them on the KEP. It is also used for a variant of the HW/SW co-design approach that uses the KEP [22, 21], where the software part is KEP assembler generated by the strl2kasm. The compiler is embedded within the *Columbia Esterel Compiler* (CEC) compiler [15] and uses the CEC as front-end to parse Esterel. This is easily possible due to the modular architecture of the CEC. The CEC uses an intermediate data structure, the *GRaph Code* (GRC), to split the Esterel statements into their basic parts using common language constructs. The strl2kasm uses its own representation, the *Concurrent KEP Assembler Graph* (CKAG) to express the control flow of KEP assembler instructions. This different internal intermediate data structures is needed due to the different compilation targets. The CEC generates common soft- and hardware code, whereas the strl2kasm match the reactive parts of Esterel directly onto the KEP assembler. This is possible because the KEP *Instruction Set Architecture* (ISA) closely resembles the Esterel language, *i. e.*, many Esterel statements can be translated straightforward to the KEP ISA.

A compilation approach for sequential Esterel programs and its WCRT were presented [29] that uses the *KEP Assembler Graph* (KAG), a forerunner of the CKAG, that supported only sequential Esterel programs. The strl2kasm now supports concurrent Esterel programs by using the multi-threaded features of the KEP, making further compilation steps necessary, in particular the *priority assignment*.

The KEP supports concurrency by implementing a priority based multi-threading concept. The threads are created by so called PAR instructions that define which instructions belong to the thread in conjunction with the initial thread priority. The PAR instructions match the Esterel concurrency operator, where the concurrent Esterel bodies correspond to the instructions of a thread. Due to the semantics of Esterel, the thread instructions are not independent from each other and have to be executed in some order. *E.g.*, an EMIT S instruction is executed before an according PRESENT S. The execution order is influenced by the threads priorities. The instructions of the thread with the highest priority are executed. Therefore the KEP ISA provides the PRIO instructions. These have an integer argument, a priority, to that its thread is changed to within the KEP scheduling.

The main purpose for the strl2kasm is to set these priorities in a way that the instructions are always executed in the right order. Note that the ordering has to be ensured for all possible signal statuses from the input side. Therefore the strl2kasm compiler can set the initial priorities in the PAR instructions as well as insert PRIO instructions. This goal is achieved by the earlier mentioned *priority assignment*. Its basic idea is to assign each instruction a priority by which the instruction has to be executed, thereby a higher priority is assigned to an EMIT S than to an according PRESENT S. This is called a dependency constraint, any such must be fulfilled. Other constraints belong to the normal control flow, which induces that thread priorities can never grow instantaneously, because an instantaneous priority increase does not guarantee that a thread with a higher priority has not already been executed. The fulfillment of all constraints is the priority assignment. The priority assignment computation is based on a *Depth First Search* (DFS) algorithm that traverses only the instantaneously reachable successors of an instruction. After the assignment, the threads are initialized accordingly, and wherever the assigned priority between instructions differ, a PRIO instruction is inserted.

The strl2kasm supports several optimization techniques. Some are common with other compilers like *dead code elimination* and *constant propagation*. Other target the KEP scheduling principle to solve dependency constraints with appropriate thread identifiers when possible.

The strl2kasm generates efficient assembler code by making use of the KEP scheduling principles and the optimizations resulting from these.

In addition to this introduction chapter, this thesis is organized as follows:

- Chapter 2 describes the synchronous language Esterel and its compilation, as well as the KEP and its instruction set architecture.
- In Chapter 3 the Esterel dismantling is presented. This is a pre-processing step that reduces the amount of Esterel syntax without weakening the expressiveness of Esterel.
- Chapter 4 introduces the CKAG, an intermediate graph structure that is needed during the compilation.
- The creation of the CKAG is described in Chapter 5. This is the first main compilation step.
- The Esterel concurrency is implemented by the *priority assignment* explained in Chapter 6. If necessary, PRIO instructions are inserted.
- Chapter 7 presents several compiler optimizations that are implemented in the strl2kasm.
- The strl2kasm is discussed and validated in Chapter 8 on the basis of the *estbench*, a set of standard Esterel programs.
- The strl2kasm implementation and usage is described in Chapter 9.
- The conclusions of this thesis are presented in Chapter 10 along with possible further work.
- The Appendix shows some bigger CKAG and thread-id tree examples.

Several aspects of the work documented in this thesis were already presented elsewhere. A sequential version of the KEP and its WCRT, based on the KAG, were presented at *CASES'05* [29]. The priority assignment used by the strl2kasm was presented at *ASPLOS'06* [28]. The strl2kasm was also used for an approach of HW/SW co-design using the KEP [22]. The computation of the WCRT has also been implemented in the strl2kasm as a student research project [11], and was presented at *SLA++P'07* [12].

# 2 Basics and Related Work

The specifies of each compiler depend heavily on its source and target language. For the *Esterel to KEP assembler* (strl2kasm) compiler presented here, this is the synchronous language Esterel and the instruction set of the KEP. Before discussing the design issues that depend on these languages, first, some general compiler implementation topics are treated that affect several performance criteria, see Aho *et al.* [1] for a detailed overview on compilers. Many of these regard both the compiler implementation and the properties of the resulting target code.

- *Compiler Speed*: On the one side, the speed of the compilation process is important to generate code in an acceptable amount of time. Therefore the used algorithms are of low complexity or at least have a linear runtime in practice. On the other side, the speed of the target code is even more relevant. Since many Esterel statements are compiled straightforward and efficiently to the KEP assembler, this goal is achieved, see Chapter 8.

- *Code Quality*: The implementation is split into several modules to handle the complexity of the problem. The implementation on the low level is made as robust and readable as possible. Regarding the target code, several optimization techniques are applied to generate efficient code (see Chapter 7).

- *Error Diagnostics*: The compiler itself is designed to document the whole compilation process. This makes it easier to find bugs. The quality of the KEP assembler programs is verified by their execution of testcase scenarios that are generated from the according Esterel source (see Chapter 8). The output traces are compared to the output of an Esterel reference implementation, in this case *EsterelStudio*. A different output indicates an error.

- *Portability*: The *portability* criterion can be divided into *retargetability* and *rehostability*. The retargetability is hard to achieve, because the KEP instruction set is focused on Esterel. However, the strl2kasm is constructed of several modules, see Chapter 9, which allows a high flexibility. The strl2kasm is written in C++, which is a standard and well established programming language with existing compilers on several architectures. It uses only the standard library and depends on software that is already available for several operating systems. Therefore the rehostability should easily possible. Currently it is running under the *Linux* operating system as well as Windows using *Cygwin*.

- *Maintainability*: The implementation language C++ is an object-oriented programming language that allows easy inheritance and data hiding through class

| Esterel Source | Description |
|---|---|
| ; | Sequence operator |
| \|\| | Parallel operator |
| **pause** | The control pauses and restarts in the next instant. |
| **emit** S | Signal S is emitted and becomes present in the current instant. |
| **present** *S* **then**<br>p<br>**else**<br>q<br>**end present** | If signal S is present the then body p is executed otherwise the else body q. |
| **loop**<br>p<br>**end loop** | The body p is infinitely often repeated if not stopped from an outer preemption. Therefore the loop body p has to be non-instantaneous. |
| **suspend**<br>p<br>**when** S | The body p is suspended when signal S occurs. |
| **nothing** | Empty statement |
| **signal** S **in**<br>p<br>**end signal** | A new local signal S is defined with program body p as scope. All global signals of name S are overridden in p. Note that a global S might be present, but the new S will always be absent until an emit S occurs inside p. |
| **trap** *T* **in**<br>...<br>**exit** *T*<br>...<br>**end trap** | The exit T statement terminates, if called, the entire trap body. |

Figure 2.1: Esterel Kernel Statements

data structures. Since the strl2kasm implementation is built of classes with well defined interfaces, these can be replaced by sub-classes for further properties as well as the internal rewrite of class methods.

Next the Esterel language and semantics as well as its compilation are described in detail.

## 2.1 The Esterel Language

The Esterel programming language [9] belongs to the synchronous languages. Its execution is divided into logical *instants*, also called *ticks*. The communication between the environment, different programs and within or across threads occurs via *signals*. Each Esterel program provides a signal interface of *input* and *output* signals; local

```
abort
  p
when S
```

$\leadsto_{ker}$

```
trap T in
  [
    suspend
      p
    when S;
    exit T
  ||
    loop
      pause;
      present S then
        exit T
      end present
    end loop
  ]
end trap
```

(a) abort in kernel statements

```
weak abort
  p
when S
```

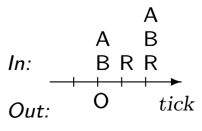
$\leadsto_{ker}$

```
trap T in
  [
    p;
    exit T
  ||
    loop
      pause;
      present S then
        exit T
      end present
    end loop
  ]
end trap
```

(b) weak abort in kernel statements

Figure 2.2: Abort Kernel Statement Representation

```
module abro:

input A,B,R;
output O;

loop
  abort
    [
      await A
    ||
      await B
    ];
    emit O;
    halt
  when R
end loop

end module
```



(a) Esterel code and sample trace

```
INPUT A,B,R
OUTPUT O

[L01] A0:
[L02]       ABORT R,A1
[L03]       PAR 1,A2,1
[L04]       PAR 1,A3,2
[L05]       PARE A4,1
[L06] A2:
[L07]       AWAIT A
[L08] A3:
[L09]       AWAIT B
[L10] A4:
[L11]       JOIN
[L12]       EMIT O
[L13]       HALT
[L14] A1:
[L15]       GOTO A0
```

(b) KEP assembler

*% KEP Execution Trace*
– Tick 1 –
*% In:*
*% Out:*
*% RT: 7*
$\textbf{ABORT}_{L2}$
$\textbf{PAR}_{L3}$ $\textbf{PAR}_{L4}$ $\textbf{PARE}_{L5}$
$\textbf{AWAIT}_{L9}$ $\textbf{AWAIT}_{L7}$
$\textbf{JOIN}_{L11}$
– Tick 2 –
*% In: A B*
*% Out: O*
*% RT: 5*
$\textbf{AWAIT}_{L9}$ $\textbf{AWAIT}_{L7}$ $\textbf{JOIN}_{L11}$
$\textbf{EMIT}_{L12}$ $\textbf{HALT}_{L13}$
– Tick 3 –
*% In: R*
*% Out:*
*% RT: 9*
$\textbf{HALT}_{L13}$ $\textbf{GOTO}_{L15}$ $\textbf{ABORT}_{L2}$
$\textbf{PAR}_{L3}$ $\textbf{PAR}_{L4}$ $\textbf{PARE}_{L5}$
$\textbf{AWAIT}_{L9}$ $\textbf{AWAIT}_{L7}$
$\textbf{JOIN}_{L11}$
– Tick 4 –
*% In: A B R*
*% Out:*
*% RT: 11*
$\textbf{AWAIT}_{L9}$ $\textbf{AWAIT}_{L7}$ $\textbf{JOIN}_{L11}$
$\textbf{GOTO}_{L15}$ $\textbf{ABORT}_{L2}$
$\textbf{PAR}_{L3}$ $\textbf{PAR}_{L4}$ $\textbf{PARE}_{L5}$
$\textbf{AWAIT}_{L9}$ $\textbf{AWAIT}_{L7}$ $\textbf{JOIN}_{L11}$

(c) KEP trace

Figure 2.3: The Esterel ABRO example.

signals are declared by the *signal* statement. At each tick, a signal status is either *present* (emitted) or *absent* (not emitted); the status may not change within a tick.

Esterel statements are either *instantaneous* (also called *transient*), in which case they do not consume logical time, which is referred to as *synchrony hypothesis* [10]); or they are *delayed*, in which case they finish the execution for the current tick. Most statements are transient including, for example, emit, loop, present, or the preemption operators. Delayed statements include pause, (non-immediate) await, and every.

The Esterel *parallelism* is expressed by the concurrency operator, denoted by ||, that groups statements in concurrently executed threads. The parallel terminates when all its branches have terminated. Wether concurrent threads are transient or delayed depends on the thread bodies and the signal environment during execution.

Esterel offers two types of abort preemption constructs. An *abortion* kills its body when an abortion trigger occurs. The *strong* abortion statement kills its body immediately (at the beginning of a tick), while *weak* abortion lets its body receive control for a last time (abortion at the end of the tick). A *suspension* freezes the state of a body in each instant when the trigger event occurs.

Consider the standard ABRO example in Figure 2.3 (a). It is an example that is minimal by using all the Esterel's reactive features. The program body consists of a loop that encloses an abort over a parallel. The await statements are waiting concurrently for signals A and B. Each await terminates when the specific signal occur. If both signals A and B have occurred then the parallel terminates and the output signal O is emitted. The signal R resets the loop body and this reactive computation starts again.

Esterel also offers an exception handling mechanism via the trap/exit statements. An exception is *declared* with a trap scope, and is *thrown* with an exit statement. An exit T statement terminates immediately the corresponding trap T statement and weakly aborts the trap body. However, there are complications when traps are nested or when the trap scope includes concurrency. The following rules apply: if one thread raises an exception and the corresponding trap scope includes multiple threads, then all the threads are weakly aborted; if concurrent threads execute multiple exit statements in the same tick, the outermost trap takes priority. The execution of multiple exit statements belonging to the same trap is equivalent to a single exit statement.

All Esterel statements can be translated into a small set of kernel statements. In Figure 2.1, an overview of the Esterel kernel statements is given. These statements have the full semantically expressiveness as Esterel and are minimal in this behavior. This Esterel sub-set is used to define the semantics of Esterel [6].

The abort statements are not part of the kernel statements. They can be translated to kernel statements as described in Figure 2.2. Their behavior is matched by a trap containing a present test that is executed each instant concurrently to the abort body. If the signal is present then the trap is terminated by an exit; the strong abort is additionally enclosed by a suspend statement to match the immediate stop. However, the number of kernel statements that is needed to represent the abort statements is huge compared to one (abort). Therefore, to be efficient, the KEP ISA supports them directly by the ABORT instruction.

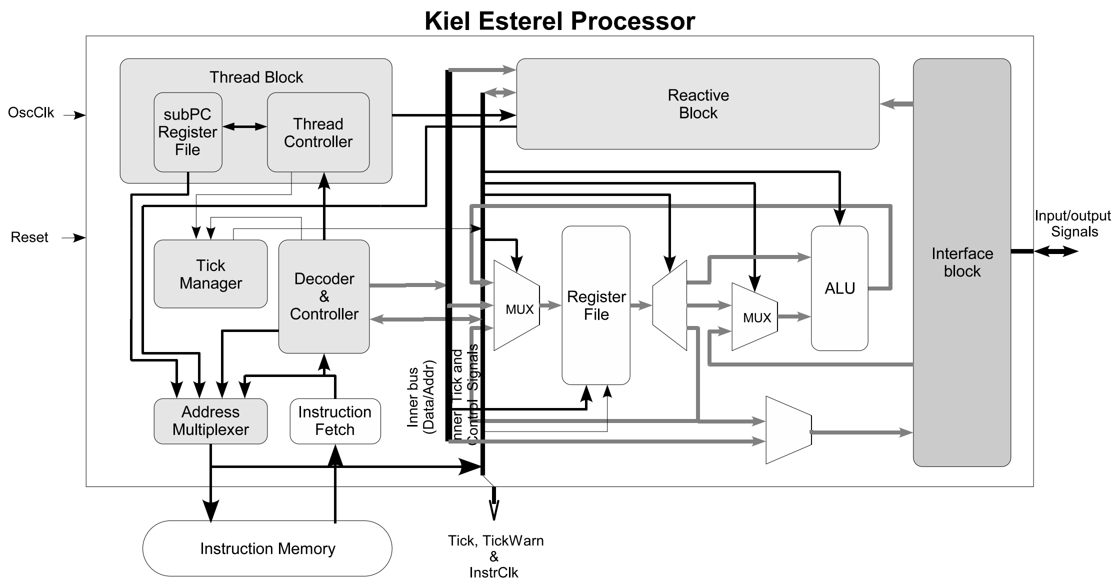


Figure 2.4: The Kiel Esterel Processor

The KEP supports a superset of the kernel statements directly. However, there remain other Esterel statements that are not handled directly, but have to be transformed into equivalent statements that are supported. This is made as a preprocessing step denoted as *esterel dismantling* (see Chapter 3).

Due to the mixture of preemption and concurrency, it is not trivial to determine whether an Esterel program is correct. The usual criteria its *constructiveness* [5]. It can be determined by a translation to combinational circuits to perform *ternary logic* [31] analysis on it. Here, we assume that the source programs are acyclic and this implies that they are constructive. However, the strl2kasm does not make a constructiveness check.

There exist multiple compilers to translate Esterel to VHDL and C. The compilation to standard architectures relies either on the simulation of netlists, or a static schedule is determined and the code is sequentialized accordingly. The Estudio implementation as well as the *Columbia Esterel Compiler* (CEC) [15] are such Esterel compilers. They use an intermediate graph structure, the GRC [34], that splits the Esterel statements into its basic parts. Due to the architecture of the KEP, this is not done in the strl2kasm compilation. This is also the reason why we use a different intermediate format, namely the CKAG that can represent the control flow of all KEP instructions.

Next, the KEP and its *Instruction Set Architecture* (ISA) is introduced.

| Esterel Source | KEP Assembler | Notes |
|---|---|---|
| **emit** *S* [(*val*)] | EMIT *S* [, {#*data*\|*reg*}] | Emit (valued) signal *S*. |
| **present** *S* **then**<br>p<br>**else**<br>q<br>**end present** | PRESENT *S*, *elseAddr*<br>P<br>GOTO *endAddr*<br>*elseAddr*:<br>Q<br>*endAddr*: | Jump to *elseAddr* if *S* is absent. No GOTO instruction is inserted when the *else* body is empty. |
| [weak] **abort**<br>p<br>**when** [immediate, *n*] *S* | [LOAD _COUNT,*n*]<br>[W]ABORT[I] *S*, *endAddr*<br>P<br>*endAddr*: | To delay a preemption for *n* ticks is done by setting the built-in variable _COUNT. |
| **suspend**<br>...<br>**when** [immediate, *n*] *S* | [LOAD _COUNT,*n*]<br>SUSPEND[I] *S*, *endAddr*<br>...<br>*endAddr*: | |
| **trap** *T* **in**<br>...<br>**exit** *T*<br>...<br>**end trap** | *startAddr*:<br>...<br>EXIT *exitAddr startAddr*<br>...<br>*exitAddr*: | Exit from a trap, *startAddr*/*exitAddr* specifies trap scope. Unlike GOTO, check for concurrent EXITs and terminate enclosing \|\|. |
| **pause** | PAUSE | Await for a signal. AWAIT TICK is equivalent to PAUSE. |
| **await** [immediate, *n*] *S* | [LOAD _COUNT, *n*]<br>AWAIT [I] *S* | |
| **sustain** *S* [(*val*)] | SUSTAIN *S* [, (#*val*\|*reg*)] | Sustain (valued) signal *S*. |
| **halt** | HALT | Halt the program. |
| **nothing** | NOTHING | No operation. Sometimes needed to distinguish address labels. |
| **loop**<br>...<br>**end loop** | *addr*:<br>...<br>GOTO *addr* | Jump to *addr*.<br>The loop body has to be non-instantaneous. |
| **[**<br>$p_1$<br>\|\|<br>⋮<br>\|\|<br>$p_n$<br>**]** | PAR *prio*$_1$, *startAddr*$_1$, *id*$_1$<br>...<br>PAR *prio*$_n$, startAddr$_n$, *id*$_n$<br>PARE *endAddr*<br>*startAddr*$_1$:<br>P$_1$<br>*startAddr*$_2$:<br>⋮<br>*startAddr*$_n$:<br>P$_n$<br>*endAddr*:<br>JOIN | 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.<br>Behind the *endAddr* label, the corresponding *join node* occurs, which is executed at the end of each instant the parallel is active. A JOIN instruction is executed a second time when it is part of a nested parallel. |
| | PRIO *prio* | The current thread priority is set to *prio*. This instruction has no direct Esterel counterpart. |

Figure 2.5: Overview of the Esterel syntax and how these Esterel statements are compiled to the KEP instruction set.

## 2.2 The Kiel Esterel Processor (KEP)

The *Kiel Esterel Processor* (KEP) [38] is a reactive processor. It was designed by Xin Li [27], therefore we follow his description:

Neither traditional processors nor classical programming languages have structures or statements/instructions to handle corresponding Esterel statements efficiently. Hence, the implementation of the Esterel semantics on commercial off-the-shelf (COTS) processors is problematic since it must be simulated. Therefore, an Esterel-based design proves its efficiency on model description and validation, but can hardly enhance the implementation performance or reduce resource usage.

The KEP is designed to handle Esterel statements directly and efficiently, *i. e.*, it is an *Application Specific Instruction-set Processor* (ASIP) that targets Esterel programs. Notable features of the KEP include the following:

- The KEP is the first reactive processor which employs a multi-threaded architecture for directly handling concurrency. This strategy uses resources efficiently and easily scales up to very high degrees of concurrency.

- The KEP contains a full-custom *reactive core*, whose instruction set and data path have been tailored exclusively for the processing of Esterel code. Hence, all types of Esterel preemption, delays, and exceptions, can be handled by KEP very efficiently.

- The KEP also includes an interface block for handling Esterel input, output and local types of pure and valued signals. Furthermore, testing the presence and values of signals across logical instants (corresponding to Esterel's pre operator) are also directly supported.

- Throughout the development of the KEP, scalability has been considered, hence the allowed number of signals, the maximum thread number, the nesting depth of preemption primitives, and other design parameters are fully configurable.

- Unlike other reactive processing approaches, the KEP ISA is *complete* in that it allows a direct mapping of all Esterel statements onto KEP assembler. All the Esterel kernel statements, including delay, preemption, concurrency and exception handling, are implemented directly and semantically accurately by the KEP, and they can be freely combined and nested as defined by the Esterel semantics. However, it can also make unrefined processing approaches fairly costly. The KEP ISA therefore not only supports common Esterel statements directly, but also takes into consideration the statement context. Providing such a *refined* ISA further minimizes hardware usage while preserving the generality of the language.

Advantages of the KEP compared with traditional processors include:

**Performance** As the instruction set and data path have been developed specifically for Esterel execution, the Esterel module can be executed fairly fast on KEP. This benefits two key aspects of system performance, *i. e.*, the *Worst Case Reaction Time* (WCRT) and *Average Case Reaction Time* (ACRT).

**Memory** Because most typical Esterel statements can be expressed directly with just a single KEP instruction, an Esterel program executed on the KEP has very low instruction and data memory usage.

**Power Usage** For controller programming, the main goal of Esterel, the control signals tend to be more often absent than present [7]. Due to the architecture of the KEP, very few instruction cycles are needed for executing a *blank event*, which corresponds to the condition of all signals being absent.

**Logic Area** The KEP offers a novel light-weight thread model, *i. e.*, the multi-threaded architecture, to implement Esterel concurrency efficiently. This characteristic significantly reduces its logic resource usage for implementing a practical (industry scale) Esterel module.

**Predictability** The KEP is not designed to optimize (average) performance for general purpose computations, and hence does not have a hierarchy of caches, pipelines, branch predictors, etc. This leads to a simpler design and execution behavior and further implies that control-flow is preserved while compiling Esterel into machine code, and that the execution platform has a very predictable timing behavior.

In summary, the KEP is an efficient reactive processor for handing practical Esterel modules, and appears to be very competitive with other implementations.

Figures 2.5 and 2.6 give an overview over the KEP ISA. The KEP ISA resembles the Esterel syntax, therefore many Esterel statements are transformed straightforward to their KEP counterparts. Nevertheless, not all Esterel statements are directly supported within the KEP syntax. Therefore, the next chapter describes how these are handled.

| **Esterel Source** | **KEP Assembler** | **Notes** |
|---|---|---|
| **input** *I* [: *type*]<br>**output** *O* [: *type*] | INPUT[V] *I*<br>OUTPUT[V] *O* | *type* $\in$ {*integer*, *boolean*}<br>I/O-statements are part of the interface: no instruction cycles are needed. |
| **var** *x* : *type* **in** | VAR *X* | *type* $\in$ {*integer*, *boolean*}<br>The VAR instructions are also part of the interface, even if the Esterel var can occur anywhere. Temporary registers needed for computations are also declared by VAR. |
| **signal** *S* **in** . . . **end** | SIGNAL *S* | Initialize a local signal *S*. |
| sigdecl *S* := val : type | SETV *S*, (#val \| reg) | The signal value of $S$ is set (sigdecl $\in$ {input,output,signal}). |
| x := x + 1<br>x := x - 1<br>x := x * 2 | ADD X,#1<br>SUB X,#1<br>MUL X,#2 | Other supported register operations are ORR,ANDR,XORR. |
| **if** x > y **then**<br>p<br>**else**<br>q<br>**end if** | CMP X,Y<br>JW G, *elseAddr*<br>P<br>GOTO *endAddr*<br>*elseAddr*:<br>Q<br>*endAddr*: | Jump to *elseAddr* if the JW compare for G fails. No GOTO statement when the *else* body is empty. Other compare operations are GE,L,LE,EE,NE matching *greater equal*, *lower (equal)*, *equal* and *unequal*. |
| x := f(5,y) | LOAD _TMP0,#5<br>LOAD _TMP1,Y<br>CALL F<br>LOAD X,_TMP0<br>. . .<br>F:<br>. . .<br>RETURN | The arguments are loaded to function interface registers _TMPi, as after the CALL the result. To address label F the function body of $F$ is compiled. |
|  | CLRC | Clear the carry bit. |
|  | SETC | Set the carry bit to '1'. |
|  | SR[C] | Shift right. Use also the carry bit by SRC. |
|  | SL[C] | Shift left. |
|  | NOTR | Bit inversion. |

Figure 2.6: KEP Instruction Set: Interface and Computational Expressions

# 3 Preprocessing: Esterel Dismantling

This chapter describes an Esterel to Esterel transformation, called *Esterel dismantling*, which is very similar to the transformation of Esterel statements to kernel statements in the Esterel Primer [7]. The Esterel programs $p$ are translated to semantically equivalent Esterel programs $p'$, whereby $p'$ is part of a syntactical subset of Esterel:

$$p \rightsquigarrow_{dis} p' \quad \text{whereby} \quad [\![p]\!] = [\![p']\!].$$

This subset has the same expressiveness as Esterel, but uses less and simpler statements. Although many statements are dismantled into their kernel statements, the strl2kasm dismantles to a Esterel subset that is a superset of the Esterel kernel statements, because the KEP assembler syntax directly supports some non kernel statements like abort.

This transformation allows the integration of Esterel statements that are not directly supported by the KEP, like the every statement or complex variations of the await, and reduces hereby the amount of syntactical constructs that have to be handled by the compiler in later compilation steps.

The dismantling technique makes the strl2kasm flexible in the use of the input Esterel syntax, it obtains the downward compatibility of older Esterel programs with outdated syntax like the do watching statement, which is replaced by an abort as described in the Esterel Language Primer [7]. It might provide also the upward compatibility to future syntax extensions, provided that the CEC parser is also updated and an appropriate Esterel translation exists, so that the following compilation steps are not affected. However, possible new language constructs that expand the expressiveness of Esterel cannot be handled by dismantling and therefore would effect all compilation steps.

Note that the dismantling is a recursive process, on the one hand the bodies of complex statements are also dismantled, on the other hand some dismantled statements might be dismantled again, until they reach an atomic statement level.

## 3.1 Module/Run Dismantling

Esterel programs consists of modules. A module can be instantiated within another module by using the run statement. Recursive or mutually recursive module instantiation is forbidden [7] so it is possible to replace each run statement by its module body to get a program with only one module, the main module. This module expanding technique is implemented within the CEC [15] and the strl2kasm will make use of it as the first action after parsing.

```
loop
  p
each S
```

$\rightsquigarrow_{dis}$

```
loop
  do
    p
  upto S
end loop
```

$\rightsquigarrow_{dis}$

```
loop
  abort
    p;
    halt
  when S
end loop
```

(a) loop each dismantling

```
every S do
  p
end every
```

$\rightsquigarrow_{dis}$

```
await S;
loop
  p
each S
```

$\rightsquigarrow_{dis}$

```
await S;
loop
  abort
    p;
    halt
  when S
end loop
```

(b) every dismantling

```
do
  p
watching S
timeout
  q
end timeout
```

$\rightsquigarrow_{dis}$

```
abort
  p
when S do
  q
end abort
```

(c) dismantling do watching

```
every immediate S do
  p
end every
```

$\rightsquigarrow_{dis}$

```
await immediate S;
loop
  abort
    p;
    halt
  when S
end loop
```

(d) every immediate dismantling

```
do
  p
upto S
```

$\rightsquigarrow_{dis}$

```
abort
  p;
  halt
when S
```

(e) dismantling do up to

```
repeat n times
  p
end repeat
```

$\rightsquigarrow_{dis}$

```
var R_COUNT : integer in
  R_COUNT := n;
  trap REPEAT in
    loop
      if (R_COUNT ≤ 0) then
        exit REPEAT
      end if;
      p;
      R_COUNT := R_COUNT−1
    end loop
  end trap
end var
```

(f) repeat dismantling

Figure 3.1: The dismantling in (c) and (e) describes how the outdated statements do watching and do up to are handled respectively. The other figures describe the dismantling of the *temporal loops* loop each and every and the *finite loop* repeat. Note that the signal expression in loop each has to be non-immediate. All these Esterel loop variants are described in the Esterel Primer [7].

Note that a module that is not instantiated by a run statement will not be expanded. So it is possible in a program of multiple modules that there are still multiple modules even though the module expanding is performed, in such a case the unused modules will be ignored. Nevertheless the following compilation steps of the strl2kasm expect a single module program.

## 3.2 Dismantling of Temporal and Finite Loops

The loop each statement is a temporal loop, its body is started initially and restarts when the loop each signal expression occurs. After the termination of its body the loop halts waiting for the signal expression. This behavior is matched by a combination of a simple loop and a do up to over the loop each body, see Figure 3.1 (c).

The delay cannot be immediate, otherwise the abort would be instantaneous and infinitely often restarted by the loop. Such an instantaneous loop is not allowed in Esterel. The strl2kasm throws an error message when the loop each signal expression has the immediate attribute.

As described in the Primer [7], the every statement is the second type of temporal loop. The difference to the loop each is that its signal expression is initially awaited before its body is started.

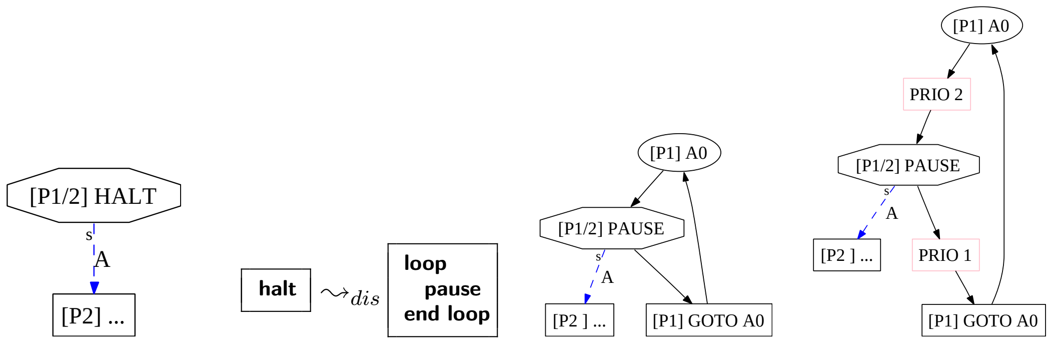
It is also possible that the delay is immediate, in that case the await placed in front is also immediate, but for the following loop each the immediate has to be removed, otherwise the loop body would be instantaneous, as described before.

Unlike loop and temporal loops the repeat statement executes its body only for a finite number of times, but the body nevertheless has to be non-instantaneous. How often the repeat body is executed is specified by an integer expression. If this expression is lower or equal to zero, the repeat statement terminates instantaneously. Determining the positiveness is in general hard to compute and might be statically not possible, so it is not allowed to have the repeat as loop body.

The basic idea of the repeat dismantling is to put the repeat body into a loop. To make this possible the repeat body has to be non-instantaneous. A counter register of name R_COUNT is initialized before the loop starts. At the beginning of the loop it is tested whether the count is lower or equal to zero and if true, the loop terminates. This is realized by putting the loop within a simple trap. At the end of the loop the counter is incremented by one. In Figure 3.1 (f) the repeat dismantling is described. The dismantling of nested repeat statements uses variables R_COUNT_$i$ with $i \in \mathbb{N}_0$ to distinguish the different variable counts.

## 3.3 Priority Dismantling

Under some circumstances it is necessary to dismantle the Esterel statements halt and await, even though they are directly supported by the KEP. Due to the KEP concurrency model, which uses multi-threading via priorities to ensure the correct



(a) assignment cannot be realized (b) dismantling halt (c) scheduling possible via PRIO instructions

Figure 3.2: This example explains why the priority dismantling is necessary to ensure the KEP multi-threading via priorities. In (a) the HALT instruction has a priority assignment of [P1/2], which means that it should have priority of one and in the next instant its priority must be increased to two, but only if signal $A$ occurs, otherwise its priority has to be one again. This cannot be realized via PRIO instructions, as explained in the text. So we dismantle the halt, see (b), and we can insert PRIO instructions to get the correct behavior, see (c).

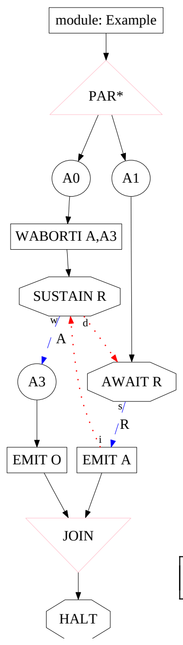
execution order according to the semantics of Esterel, the so called priority assignment is performed, see Chapter 6, which tries to fulfill all signal dependencies. After the assignment, appropriate PRIO instructions are inserted to realize the priority assignment.

Not all statements are executed instantaneously, some might be active for several instants. These statements are called *delayed statements*. During these instants, such a statement might have different priorities, depending on the current signal environment, but once assigned, the HALT and AWAIT would hold their priority for all following instants. It is only possible to have two priorities for an instruction at maximum, the first is for the current instant and the second, possibly higher priority, if necessary, for all following instants. The priority assignment takes this into account by assigning an additional priority for the next instant called *prio next*, which becomes valid after the current instant is executed and the next one is started. Only the PAUSE instruction allows to implement this behavior as explained in the following. So the solution is to dismantle all delayed statements onto a level, where the only *delayed* statements are PAUSE instructions.

Figure 3.2 illustrates the problem. The priority assignment might have computed for the delayed instruction HALT a priority of one and a next priority of two, denoted by [P1/2]. The next value depends on the signal A: only when A occurs and the HALT is aborted, the next instruction, which must have for some reason a priority of two, is executed and so the HALT must have a priority of at least two. So the priorities

```
module Example:

input I;
output O;

signal A, R in
  [
     weak abort
       sustain R
     when immediate A;
     emit O
   ||
     await R;
     emit A
  ]
end signal

end module
```

(a) Esterel source



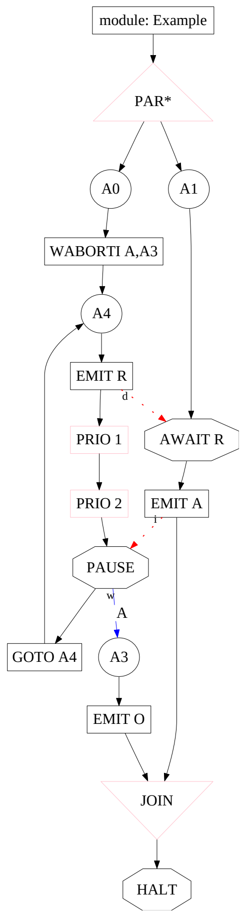
(b) priority cycle

```
sustain R
```

$\rightsquigarrow_{dis}$

```
loop
  emit R;
  pause
end loop
```

(c) dismantling sustain



(d) no priority cycle

Figure 3.3: This example shows a priority cycle caused by a sustain statement. Note that in (b) the AWAIT instruction is normally dismantled, but to make the dependency cycle easier to illustrate, it is retained. This cycle is solved by dismantling the sustain, so it possible to insert PRIO instructions in between the writer and reader part to run them in different priorities.

of the HALT should be always one, except the last instant, when it is aborted, but this point of time might vary and depend on the signal content. Nevertheless, as mentioned, the execution of the PRIO 2 before the HALT would result in a higher priority of two for all the following instants, violating the assignment. If PRIO 2 is not executed, the HALT will always have the priority of one, which also violates the assignment. It is in general not possible to schedule a KEP assembler program with non-PAUSE delayed instructions via PRIO instructions. By dismantling the halt to its kernel statements (b), as described in the Esterel Primer [7], the resulting KEP program can be scheduled. As seen in (c), the insertion of a PRIO 1 and PRIO 2 instruction can now ensure the desired behavior.

Note that if a delayed statement has to be executed after compiling with the same priority for all instants, the dismantling is not necessary. This can be ensured for statements that are part of the main thread. The strl2kasm uses an optimization technique, called *collapsing* (see Section 7.2), to use yet again the HALT, AWAIT and SUSTAIN instructions if possible. If no PRIO instructions were inserted within a dismantled instruction, then this instruction runs always with the same priority and the collapsing will undo the dismantling.

For the same reason as halt and await, the sustain has to be dismantled, because this statement is also delayed and can be active for several instants, whose number may vary. But there is yet another reason, why the sustain must be dismantled, namely to eliminate potential dependency cycles. Even if the problem described before is solved by enlarging the KEP assembler syntax to allow delayed statements to run in different priorities without being dismantled, the sustain has still to be dismantled. The SUSTAIN S instruction influences the status of signal S (is a writer of S) and must be executed before instructions that depend on S's status are executed, *e. g.*, the PRESENT S instruction is a reader of S. In this behavior the SUSTAIN is a writer, but it can also be a reader to another signal, when occurring within an according strong ABORT scope to that signal, because of the delayed control flow. Being a writer and a reader at the same time, whereby writers must have potentially high and readers low priorities, can result in a dependency cycle. In Figure 3.3 (b) the SUSTAIN R instruction/node is a writer to the AWAIT R and a reader of the EMIT A, because the SUSTAIN R occur in the scope of the WABORT A,A3. The writer-reader relation is denoted by dotted lines in the graph. The priority of a writer has to be greater than the of its reader, so the SUSTAIN R priority must be greater than the of AWAIT R and on the other hand the priority of EMIT A has to be greater than the of SUSTAIN R. Since priorities cannot increase instantaneously within a thread (see the *control flow constraints* in Section 6.2), the following inequation has to be solved in order to assign a schedule:

$$prio(\text{SUSTAIN R}) \ < \ prio(\text{EMIT A}) \ \leq \ prio(\text{AWAIT R}) \ < \ prio(\text{SUSTAIN R}).$$

Because $prio(\text{SUSTAIN R}) \not< prio(\text{SUSTAIN R})$ holds true, it is not possible to solve the inequation, a so called program cycle is found.

By dismantling the sustain statement, none of the resulting statements will be reader and writer at the same time. The writing part corresponds to the EMIT instruction and the reading part to the PAUSE. Within their execution order the EMIT comes first, so it is possible to avoid the potentially dependency cycle, by inserting PRIO instructions in between. The writer instruction can now run with a greater priority than the reader instruction, which was not possible when these instructions were bounded together within the SUSTAIN.

In the example in Figure 3.3 (d) the resulting priority constraints become:

$$prio(\mathsf{PAUSE}) \ < \ prio(\mathsf{EMIT\ A}) \ \leq \ prio(\mathsf{AWAIT\ R}) \ < \ prio(\mathsf{EMIT\ R}).$$

This inequation can be solved, because $prio(\mathsf{PAUSE}) \ < \ prio(\mathsf{EMIT\ R})$ is possible and is realized by a PRIO instruction, which lowers the priority. Note that the second PRIO instruction, which increases the priority, takes effect in the next instant, where the EMIT R instruction must have its higher priority again.

Another example of a *writer-reader statement* is the valued emit of a non-literal value, because the value might be a reader, *e. g.*, an emit S,?T statement is a writer to S and a reader to T. The basic idea to solve this problem is to split the writing and reading parts, as done in the sustain case. The writing part is the signal emission and the reading part the assignment of the signal's value, so the dismantled result would look like emit S; "S := ?T". However such an Esterel statement, which assigns a signal value without emitting this signal at the same time does not exist, at least not in Esterel v5. But the KEP assembler instruction set has such an instruction, the SETV instruction, so the separation of writer and reader is performed during the compilation on the KEP level, see Section 5.2.1, into EMIT S followed by SETV S,?T.

## 3.4 Simplification of Complex Signal Expressions

The Esterel signal expressions, which occur in present, await, abort and suspend statements, can have several non-simple forms. In the following three different types of non-simple expressions are presented.

The signal expressions of await, abort and suspend could be delayed by a preceding positive integer expression, called the *delay*. The specific statement is delayed by that factor, waiting that its signal expression is active accordingly often, before it continues. Note that the simple case is equivalent to a delay of one, whereby a 'delay of zero' is called *immediate* and is inverse to a delay; such an immediate expressions is not per se dismantled, because these are directly supported by the KEP assembler instruction set. The present has no delayed expressions, due to the fact that it is always executed instantaneously.

Another type of signal expression are the *Boolean operations*, which are part of the first order predicate calculus, consisting of the binary and and or operations and the unary not operation with signals as atomic expressions.

These kind of expressions can be combined, a short overview of the signal expres-

```
module abro:

input A,B,R;
output O;

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

end module
```

(a) original ABRO program

$\leadsto_{dis}$

```
module abro:

input A,B,R;
output O;

loop
  abort
     [
        await A
      ||
        await B
     ];
     emit O;
     halt
  when R
end loop

end module
```

(b) after dismantling loop each

$\leadsto_{dis}$

```
module abro:

input A,B,R;
output O;

loop
  abort
     [
       abort
       halt
       when A
     ||
       abort
       halt
       when B
     ];
     emit O;
     loop
       pause
     end loop
  when R
end loop

end module
```

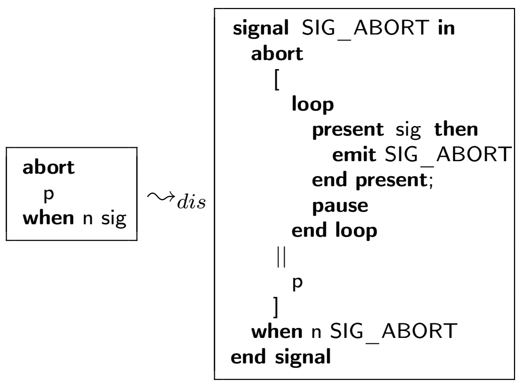
(c) after dismantling the await and halt statements

$\leadsto_{dis}$

```
module abro:

input A,B,R;
output O;

loop
  abort
     [
       abort
         loop
           pause
         end loop
       when A
     ||
       abort
         loop
           pause
         end loop
       when B
     ];
     emit O;
     loop
       pause
     end loop
  when R
end loop

end module
```

(d) dismantling completed

Figure 3.4: The dismantling of the ABRO example. First the loop each is dismantled, then the await statements and the first halt, after that the two halt statements resulting from the await dismantling are again dismantled. Then the dismantling is completed, because no more dismantling rules are applicable.

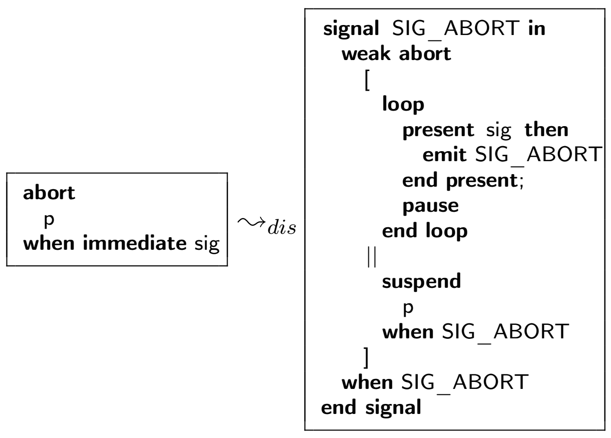


```
abort
  p
when n sig
```

$\rightsquigarrow_{dis}$

```
signal SIG_ABORT in
  abort
    [
      loop
        present sig then
          emit SIG_ABORT
        end present;
        pause
      end loop
    ||
      p
    ]
  when n SIG_ABORT
end signal
```

(a) delayed abort signal expression



```
abort
  p
when immediate sig
```

$\rightsquigarrow_{dis}$

```
signal SIG_ABORT in
  weak abort
    [
      loop
        present sig then
          emit SIG_ABORT
        end present;
        pause
      end loop
    ||
      suspend
        p
      when SIG_ABORT
    ]
  when SIG_ABORT
end signal
```

(b) immediate abort signal expression

Figure 3.5: The dismantling of Boolean Operations in abort signal expressions is shown, first if the signal expression is delayed by integer $n$ and second when it is immediate.

sion grammar:

$$\begin{aligned} \text{atomic signals: } S &= S_1 \mid S_2 \mid \ldots \mid A \mid B \mid \ldots \\ \text{Boolean operation: } sig &= S \mid \texttt{not } sig \mid (sig \texttt{ or } sig) \mid (sig \texttt{ and } sig) \\ \text{[delayed] signal expression: } dsig &= sig \mid n\ sig \qquad \text{where } n \in \mathbb{N}. \end{aligned}$$

The next complex form of signal expression is the *case expression*, which are enumerations of signal expressions The signal expressions in turn are no case expressions, but might be Boolean operations and/or delayed, if the relevant statement allow delays. The cases are considered in the order they appear in the program. The present, abort and await statements allow case expressions, the suspend does not.

Not all combinations are allowed in each Esterel statement, e. g., the present case expression consists only of *sig* expression and not *dsig*, because it is always executed instantaneously and therefore cannot be combined. On the other hand *dsig* expressions are allowed in await case and abort case statements. For more details about the Esterel grammar, especially about the Esterel signal expressions, see [9].

The signal delay is compiled into the KEPregister operation LOAD _COUNT,$n$ to initialize the KEPdelay count _COUNT with the specific delay $n$, see Section 5.2 for further details. Case Expressions might contain an unlimited number of cases and Boolean operations could be arbitrarily nested, therefore an assembler syntax of fixed bit size cannot match such expressions in general.

The complex Esterel expressions are dismantled to appropriate statements with atomic expressions, if necessary, as described in the following.

```
abort
  p
when
  case s1 do p1
  ...
  case sn do pn
end abort
```

$\leadsto_{dis}$

```
abort
  ...
    abort
      p
    when sn do
      pn
    end abort
  ...
when s1 do
  p1
end abort
```

(a) dismantling abort case

```
await
  ...
  case si [do pi]
  ...
  case immediate sj [do pj]
  ...
end await
```

$\leadsto_{dis}$

```
trap AWAIT_CASE in
  trap AC1 in
    ...
      trap ACn-1 in
        trap ACn in
    %% immediate signal tests
          ...
          present si
            exit ACj
          end present;
          ...
          loop
            pause;
              present s1 then
                exit AC1
              end present;
              . . .
              present sn then
                exit ACn
              end present;
          end loop
        end trap;
        pn
        exit AWAIT_CASE
      end trap
      pn-1;
      exit AWAIT_CASE
    ...
  end trap;
  p1
end trap
```

(b) dismantling await case

```
present
   case S1 do p1
      ...
   case Sn do pn
   else q
end present
```

$\leadsto_{dis}$

```
present S1 then
  p1
else
  ...
    else
      present Sn then
        pn
      else
        q
      end present
    end present
  ...
end present
```

(c) dismantling present case

Figure 3.6: The dismantling of Esterel statements with case expressions. (a): The abort cases are nested and held the preemption order of the signal expressions $s_i$. (b): The await cases are tested each instant in a loop, which exits according to nested traps. (c): The present case results in the nesting of several simple present statements, the if case is analog dismantled to simple ifs.

### 3.4.1 Case Expressions

Case expressions occur in conditional statements, namely present and if, and in await and abort constructs. In the following the dismantling of all case statements is described in detail.

The present case is semantically equivalent to an else if like expression:

present ... else present ... else present ... end,

and is accordingly dismantled into nested non-case present statements, see Figure 3.6 (c).

According to the Esterel semantics the present case expressions are tested in sequence from first to last as long as an expression is fulfilled, then the according bodies are executed or the next statement in sequence to the present case if no body exists. This behavior is matched by $n$ statements for $n$ cases, whereby each present statement is the else body of the previous present statement, except for the first, which has no predecessor. The last and most inner else body is the default body of the present case statement, see Figure 3.6 (c). If a one case of a present case expression has no body, it is dismantled to a present-else statement with no *then* body, the *else* body remains as previously explained.

The single case version of the present case is not dismantled, because it is equivalent to the present ... then ... [else ...] end version of the present. So the dismantling takes only place for the multiple cases. Note that the simple present statement is represented in the *Abstract Syntax Tree* (AST), after being parsed, as a single case present case statement. Therefore the dismantling process, which replaces a (multiple) present case through several nested simple presents, will result in nested single case present case statements.

The if case statement is very similar to the present case, they distinguish in the type of conditional expression, the if case tests for a standard Boolean expression instead of the Boolean signal expression as explained above. Apart from this, its dismantling works the same way and needs no further explanation.

The await case statement waits simultaneously with all its signal expressions till one occurs, then it stops and executes at least one do body. That one of the first active signal expression is executed, if it has no do body, no do body is executed at all for this await case. The dismantling will take this into account. The await case could be dismantled to an abort case with a halt as body, but for performance issues, it is separately dismantled.

The await case behavior is matched by a loop with a pause statement as body and several signal testing via present statements to this loop, see Figure 3.6 (b). The loop is exited by several exit statements of related trap statements, which are placed around the loop, one trap for each case. They are nested from outer to inner according to their case sequence from first to last. Their symbol names are named with AC followed by an integer for distinction. The case bodies are placed behind their specific trap statements. An additional trap of name AWAIT_CASE[_$i$] is positioned around the whole dismantle expression to exit the expression after the execution of a case body has taken place. If no body is specified for a given case, no trap label is needed to jump for, the control flow can start directly at the next statement, so an exit statement is created to the most outer trap label AWAIT_CASE[_$i$] and no own trap AC_$j$ is created for it. So it does not matter whether the cases have do bodies or not, this additional trap is always needed. The number of used traps statements is bounded to be at most the number of cases plus one, if each case has a body, and to be at least one if there are no case bodies at all.

The abort case prioritizes the various signal expressions according to their sequence within its case expression. The nesting of abort also allows prioritizing of signal expressions, the outer more an abort the higher the preemption. Thus the dismantling process converts the abort case into multiple nested abort statements, see Figure 3.6 (a).

### 3.4.2 Boolean Signal Expressions

Boolean signal expressions occur in the signal reading statements: await, abort, suspend and present. The Boolean signal expressions of the non-instantaneous statements await, abort and suspend can also be delayed or immediate. The present is instantaneously executed and therefore has no delay. The KEP does not support Boolean signal expressions directly in its instruction set architecture, so they have to be simplified to simple atomic expressions.

The present statement cannot be dismantled in general without duplicating one of its bodies, thus violating the *Write Things Once* (WTO) principle. The nesting of a present-or expression leads to a duplication of the *then* body, the present-and to a duplication of the *else* body. This may increase the code size exponentially for accordingly nested Boolean signal expressions of the present. Therefore expressions in present statements are handled later during the creation of the KEP assembler instructions and not at the Esterel level (see Section 5.2.1 and Figure 5.3).

The await statements that contain Boolean signal expression are handled the same way as the simple await with atomic signal expressions, they are dismantled to an abort statement of the same Boolean signal expression with a halt statement as body. This is done both for the delayed or immediate expressions.

The dismantling of abort statements with Boolean signal expressions is split in two cases, the non-zero and the zero delay case, both can be seen in Figure 3.5. In both cases a new local signal *SIG_ABORT* is created to replace the original signal expression *sig* of the abort. If *sig* occurs, signal *SIG_ABORT* should be present to abort the body $p$, this is realized by a parallel extension of $p$, where each instant is tested, whether *sig* is present, and if present, signal *SIG_ABORT* is emitted. A delay $n$ of *sig* is transferred to *SIG_ABORT* without the need for further modifications.

In case of an immediate *sig*, the abort is made weak to allow the immediate abortion from within the (now) weak abort statement. The semantic of a weak abort recommends that all instantaneously executable statements are executed, especially $p$, even if *SIG_ABORT* is present, which would violate the former behavior of an strong abortion. This problem is fixed by putting $p$ into a suspend with trigger *SIG_ABORT*.

# 4 The Concurrent KEP Assembler Graph (CKAG)

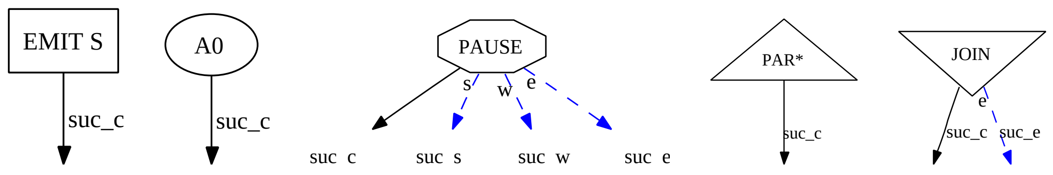
During the compilation from Esterel to KEP assembler, an intermediate graph structure is used that represents the control flow of the resulting KEP assembler. This structure, called the *Concurrent KEP Assembler Graph* (CKAG), is a directed graph with KEP assembler instructions as nodes and several edge types to match the normal composition of instructions, denoted by *control flow edges*, as well as preemptive control flow, denoted by *preemption edges*.

The graph structure is named *concurrent*, because it allows the representation of concurrent control flow by using so called *fork nodes*. This is an advancement in comparison to its predecessor version, the KAG [29], which was designed for sequential programs only. The CKAG additionally provides *join nodes* to join again the control after being forked, whereby the *fork nodes* and *join nodes* represent the PAR/PARE and JOIN instructions of the according KEP assembler program, respectively.

The CKAG is required to realize the *priority assignment* [28], needed to ensure the parallel semantics of Esterel, as well as for the *Worst Case Reaction Time* (WCRT) analysis [12]. The optimizations performed by the compiler like *dead code elimination* and the *instruction collapsing*, (see Chapter 7) are realized on the CKAG structure, too. It would be possible to implement these computations directly on the KEP assembler instructions, which are a list. However, it would be much less efficient to find the label addresses of, *e. g.*, GOTO, PRESENT and ABORT instructions, because the next instruction in the control flow is in general not its successor in the instruction list. This is even worse for the preemptive control flow. In the CKAG the control flow successors are directly and efficiently accessible. Nevertheless the KEP assembler instructions and its CKAG are closely related and a change in the CKAG implies a change in the KEP assembler and vice versa. *E.g.*, if a node in the CKAG is removed, the according instruction has to be removed from the KEP instructions.

Each time PAR/PARE instructions are created the control flow is split into several sub-threads. As this can happen recursively, the resulting thread structure might be complex. To easily identify semantically dependent statements that might be executed concurrently, the subthread hierarchy induced by the *fork nodes* is made abstract as a tree structure (see Section 4.2). This structure can also be used to optimize the use of the thread-id values defined by PAR (see Section 7.5).

Before describing the graph building process from Esterel source, the CKAG with all related structures is defined and explained in detail. First are all node and edge types of the CKAG presented, followed by the according thread-id tree and some conditions about the scopes of all KEP signal, register and address names.



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

Figure 4.1: Nodes and edges of the Concurrent KEP Assembler Graph (CKAG).

We define the CKAG structure as a directed graph with nodes $N$ and different types of edges $E$ and $P$:

**Definition 4.1** (Concurrent KEP Assembler Graph (CKAG)). *A CKAG $C$ of a KEP program $p$ is given by*

$$C = C(p) = (N, E, P)$$

*with node set $N$, edge set $E \subseteq N \times N$ and preemption edges $P$. Each preemption edge is a tuple of preemption type $k \in K := \{s, w, e\}$ and signal symbol $s \in \mathbb{S}$, where $\mathbb{S}$ is defined as the set of signal symbols. The preemption type indicates whether the preemption results from strong abort, weak abort or an exit, and the signal symbol by which signal the preemption is triggered.*

$$P \subseteq N \times (K \times \mathbb{S}) \times N.$$

Note that an exit preemption never occurs for signals, but for trap labels, which are handled like signals in the graph.

The CKAG represents the control flow of a KEP program, it is required to have a control flow source node, from which the program starts. This node is called the *root* of a CKAG and represents all INPUT/OUTPUT instructions and all other instructions executed at program start, like the initialization of the valued signal _TICKLEN.

It is —at least initially— required that there is only one node without a parent, the root node. In addition to this a CKAG should be coherent, excluding solitary *join nodes*, which are still needed. If one of these two properties are not met, it can be enforced by removing all nodes and edges that are not reachable from the root node. Such a generated CKAG remains semantically equivalent, because the deleted nodes corresponded to unreachable KEP instructions.

## 4.1 Nodes and Edges

The CKAG distinguishes *transient nodes* and *label nodes*, which represent instantaneous execution, *delay nodes*, which represent instructions that may hold for more than one tick, and *fork* and *join nodes*, which represent concurrency (see Figure 4.1). An overview about all CKAG node types and the according KEP instructions they represent:

**Definition 4.2** (CKAG Nodes)**.**

***T***: *Transient Nodes. This includes* EMIT*, conditional* PRESENT/JW*, all variations of* ABORT*,* SUSPEND*, and computational* ADD*, instructions, shown as a box.*

***L***: *Label Nodes are like transient nodes, except that they only hold address labels and no real KEP instructions. they are displayed in an oval form.*

***D***: *Delay Nodes (octagons), which correspond to delayed KEP instructions (*PAUSE*,* AWAIT*,* HALT*,* SUSTAIN*).*

***F***: *Fork Nodes (triangles), corresponding to* PAR/PARE *instructions. At least two* PAR *instructions and always one* PARE *are associated with a fork node.*

***J***: *Join Nodes (inverted triangles). They contain* JOIN *instructions only.*

***N***: *The set of all nodes $N$ is partitioned into the five different type of nodes:*

$$N \ := \ D \ \dot{\cup} \ T \ \dot{\cup} \ L \ \dot{\cup} \ F \ \dot{\cup} \ J.$$

Each *fork node $f$* has exactly one corresponding *join node $j$* and the other way round, which can referred by $f.join$ and $j.fork$ respectively:

$$f.join := j \ \text{ and } \ j.fork := f.$$

This property is ensured during the creation of each fork-join pair, see the parallel creation Figure 5.1 (e). A simple conclusion is that the sets of *fork* and *join nodes* always have the same cardinality: $|F| \ = \ |J|$.

The KEP assembler instructions of each node $n \in N$ are referred by $n.stmt$ respective $n.stmts$ to indicate that $n$ contains multiple instructions. As convention the KEP instructions are used to further specify the nodes beyond the node types, *e. g.*, is $n$ called *[weak] abort node* or *goto node*, if $n.stmt$ equals a [W]ABORT or GOTO instruction respectively. The delay and *label nodes* are thereof not excluded: a *delay node $d$* is called *pause node* if $d.stmt =$ PAUSE and a *label node $l$* is named *trap label* if the address $l.stmt$ is an EXIT address.

Each abort and suspend node $n$, *i. e.*, $n.stmt =$ [L|T][W]ABORT[I]/SUSPEND[I], defines an instruction scope from $n.stmt$ to an address label *addr* within the KEP program. We define $n.end \in L$ as the *label node* containing the abort label *addr*, *i. e.*, $n.end.stmt \ = \ addr$, and $n.scope \subseteq N$ as the nodes containing all instructions of the scope: $n.scope := \{n.stmt, \ldots, n.end.stmt\}$.

For a (weak) abort node $n$ and each node $d \in D \cap n.scope$, there exists a preemption edge from $d$ to $n.end$, *i. e.*, $(d, n.end)_{(w/s,S)}$, where $S$ is the according abort signal of $n.stmt$. The node $n.end$ is called a *preemption successor* of $d$. The CKAG edge sets $E$ and $P$ implicate this and other types of node successors as described in the following.

**Definition 4.3** (Node Successors). *Given a node $n$, we define $n.suc_c \subseteq N$ as the set its of successors in $E$ or the image of $E$ under $n$, named control flow successors of $n$:*

$$n.suc_c \;:=\; E[n] \;=\; \{m \mid (n,m) \in E\}.$$

*The preemption successors as successors in $P$ are analogously defined for all preemption types $k \in \{s, w, e\}$:*

$$n.suc_k \;:=\; P[n]_{(k,.)} \;:=\; \{m \in V \mid \exists s \in \mathbb{S} : (n,m)_{(k,s)} \in P\}.$$

*Furthermore all successors $n.suc$ of $n \in N$ are defined by:*

$$n.suc \;:=\; E[n] \,\cup\, P[n].$$

Successors reached via preemption edges are $n.suc_s$ for strong aborts, $n.suc_w$ for weak aborts, and $n.suc_e$ for exit exceptions. These edges are represented as dashed edges, in contrast to solid edges of the control flow successors, marked with small tail labels $s$, $w$ and $e$, respectively. See Figure 4.1 for an overview about all edge types respective successors. Note that according to the semantics of Esterel preemption edges occur only from *delay nodes*, because a preemption either takes place at the beginning or the end of an execution, when a pause is reached. In contrast, all transient and *label nodes* have control flow successors only. 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 $n.end \in n.suc_c$.

Given $d \in D$, the *weak abort successors* $d.suc_w$ and the *exit successors* $d.suc_e$ are the nodes to which control can be transferred immediately, that is, when entering $d$ at the end of a tick, from $d$ to via an abort or trap start node.

The *strong abort successors* $d.suc_s$ are the nodes to which control can be transferred after a delay, that is, when restarting $d$ at the beginning of a tick, from $d$ to via an abort node. If $d \in D$ and $d \in n.scope$ for some strong abort node $n$, it is $n.end \in d.suc_s$. 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 $d$ is entered) for the preemption to take place.

For $f \in F$, $f.suc_c$ includes the nodes corresponding to the beginnings of the forked threads. If $n$ is the last node of a concurrent thread, $n.suc_c$ might include the join node for the corresponding JOIN instruction.

As in Esterel, where no instantaneous loops are allowed, we forbid the existence of an *instantaneous cycle* in the CKAG. To determine whether Esterel statements are instantaneously executed is explained in detail by Tardieu and de Simone [36]. They also point out that an exact analysis of instantaneous reachability has NP complexity, this holds true for the CKAG respective KEP, so we make an conservative approach of a cycle. First the sequential case is defined:

**Definition 4.4** (Sequential Instantaneous Successors). *Given a node $n \in N \backslash (F \cup J)$, its instantaneous successors are defined by:*

$$n.suc_{inst} := \begin{cases} n.suc_c & : n \in T \cup L \\ n.suc_w \cup n.suc_e & : n \in D \end{cases}$$

*A node vector $p = (p_0, \ldots, p_{l-1}), n_i \in N \setminus (F \cup J), l \in \mathbb{N}$ is called sequential instantaneous path of length $l$, if they form an instantaneous successor chain:*

$$p \textit{ sequential instantaneous path } := \forall p_i, i = 0, \ldots, l-2 : \; n_{i+1} \in n_i.suc_{inst}.$$

Note that *transient nodes*/*label nodes* have no preemption successors and its control flow successors are always instantaneous. The instantaneous successors of *delay nodes* are never control flow successors, but all preemption successors ($n.suc_w \cup n.suc_e \subseteq P[n]$), as the preemption edges of types $w$ and $e$ are instantaneously executed.

Given a sequential instantaneous path $p$, if $p$ contain a node $n$ more than once, then $p$ forms a sequential instantaneous cycle and is called itself *(sequential) cycle*. Such a cycle can easily detected by a so called *instantaneous DFS* over the CKAG. This is a DFS algorithm that uses only instantaneous successors for its traversal. The algorithm starts its DFS at the root node and at all *delay nodes*, so are the control flow successors of *delay nodes* not traversed until the DFS starts at them. The WCRT analysis [12] is such an algorithm, which cannot be computed or would be infinite, if an instantaneous cycle occur within the CKAG.

A non-trivial task of the CKAG structure is to properly define the different types of possible control flow for non-sequential nodes, *i. e.*, fork and *join nodes*, since it depends on their sub-thread bodies, whether they are instantaneous or not, or both. This applies also for their node successors, if a fork-join $(f, j)$ is instantaneously executed, then the nodes $j.suc_c$ are instantaneous successors of $(f, j)$. Given a strong abort node $n$, if $(f, j)$ is part of the abort scope $n.scope$, all *delay nodes* $d$ within the $(f, j)$ scope lead to an abortion of it. The according non-instantaneous successors $d.suc_c$ of $d$ are in consequence also non-instantaneous successors of $(f, j)$. The weak abort case leads also to successors of a fork-join, but it might be again varies, whether they are instantaneous or not.

The possible control flow successors of a KEP parallel construct respective their fork and *join nodes* are defined hereafter.

**Definition 4.5** (Fork-Join Successors). *Given a fork node $f \in F$, $j := f.join$, with the according fork scopes defined as $f.sub \subseteq N$ and $f.sub_i \subseteq N$ containing all nodes of $(f, j)$ and the sub-thread $i$ respectively.*

*The parallel abort preemption successors are derived from the preemption edges of the delay nodes in scope $f.sub$, whose control flow continues outside of this scope:*

$$\forall k \in \{s, w\} : \; j.suc_k := \bigcup_{d \in f.sub} d.suc_k \;\setminus f.sub$$

*All parallel successors, instantaneous and non-instantaneous, are noted with $(f,j).suc$:*

$$(f,j).suc \ := \ j.suc_c \cup j.suc_s \cup j.suc_w \cup j.suc_e$$

A fork-join successor $n$ of $(f,j)$ is called *instantaneous*, if $n$ is instantaneously reachable from $f$. This is the case, when it is instantaneously reachable by all its sub-threads over normal control flow edges or an exit or weak abort preemption is instantaneously reachable, and therefore terminates $(f,j)$ instantaneously. It is called *delayed*, if $n$ is a strong abort successor of $j$ or a preemption successor, whose preemption might take place after a delay. If a *delay node* exists in the scope of $(f,j)$, then are additionally the control successors of $j$ be delayed. See the definition:

**Definition 4.6** (Instantaneous/Delayed Fork-Join Successors)**.** *Given a fork-join pair $(f,j)$, the set $(f,j).suc_{inst} \subseteq (f,j).suc$ of instantaneous fork-join successors of $(f,j)$ are defined as follows:*

$$\begin{aligned} n \in (f,j).suc_{inst} \ := \ & (n \in j.suc_c \ \wedge \ \forall n_i \in f.suc_c \ \exists \textit{inst. path from } n_i \textit{ to } j) \ \vee \\ & (n \in j.suc_w \ \wedge \ \exists d \in D \ \ \exists \textit{inst. path from } f \textit{ to } d : n \in d.suc_w) \ \vee \\ & (n \in j.suc_e \ \wedge \ \exists n' \in N \ \ \exists \textit{inst. path from } f \textit{ to } n' : n'.stmt = \mathsf{EXIT} \wedge n \in n'.suc_e) \end{aligned}$$

*The delayed fork-join successors $(f,j).suc_{delayed} \subseteq (f,j).suc$:*

$$\begin{aligned} n \in (f,j).suc_{delayed} \ := \ & n \in j.suc_s \quad \vee \\ & (n \in j.suc_c \ \wedge \ \exists d \in f.sub \cap D) \quad \vee \\ & (n \in j.suc_w \ \wedge \ \exists d \in D \ \ \exists \textit{delayed path from } f \textit{ to } d : n \in d.suc_w) \ \vee \\ & (n \in j.suc_e \wedge \exists n' \in N \ \ \exists \textit{delayed path from } f \textit{ to } n' : n'.stmt = \mathsf{EXIT} \wedge n \in n'.suc_e) \end{aligned}$$

*In general these sets are not distinct:*

$$(f,j).suc_{inst} \ \cap \ (f,j).suc_{delayed} \ \neq \ \emptyset \quad \textit{(general case)}.$$

Note that these definitions are conservative, *i. e.*, an instantaneously successor might be instantaneously reachable, but could also be executed delayed and the other way round. Only if a successor is exclusively in $(f,j).suc_{inst}$ and $(f,j).suc_{delayed}$ it will, when executed, be executed instantaneously and delayed according to $f$ respectively.

The definition above is recursive, because the occurring paths might again contain a fork-join pair. If so, those must also be instantaneous and delayed, that means they have an instantaneous and delayed successor respectively. In this meaning these paths are called *parallel instantaneous path* and *parallel delayed path*.

As already mentioned, we assume that the given program does not have cycles. In the notion of parallel paths, a CKAG is not allowed to have an instantaneous parallel path that forms a cycle, *e. g.*, an instantaneous fork-join $(f,j)$ and the existence of an instantaneous path from $j$ to $f$ would lead to an irregular CKAG with an infinite and therefore invalid WCRT.

## 4.2 KEP Thread-Id Tree

This section introduces the *KEP thread-id*, a data structure that is used to hold all information of a KEP thread and its relations to other threads within a CKAG. The set of all KEP thread-ids form a tree. Each node $n$ in a CKAG has a KEP thread-id, which is referenced by $n.threadid$. All KEP thread-ids of a CKAG $C$ are denoted with $\mathbb{T}$ and $\mathbb{T}_C$, respectively.

The KEP threads are identified by an integer value, which has to be unique at runtime to ensure a correct KEP multi-threading. To match the KEP thread-id to the KEP threads, we define the *KEP thread-id value*, which is referenced by $t.id$ for each thread $t$. The $.id$ operation projects the CKAG thread structure to the KEP assembler, *i. e.*, threads are identified by the KEP as integer id values without the need to know the exact hierarchy of the threads. Note that two different threads can have the same id value, if they are never executed at the same time, so in general: $t_1.id = t_2.id \not\Rightarrow t_1 = t_2$. Therefore the $.id$ function is not injective and we cannot use the thread id values instead of the KEP thread id. This behavior will later be used to minimize the use of thread-id values, see Section 7.5.3.

Each KEP thread-id has a position within the *KEP thread-id hierarchy*, which is defined by its parents and children. The parent KEP thread-id is referenced by $.parent$ and exists for all thread-ids except the *main thread*, which is the root of the thread-id structure. We denote the main thread's parent thread with the standard bottom symbol $\perp$ to handle undefined expressions.

Given a KEP thread-id $t$, its children are denoted by $t.children$. These children were all created within $t$, *i. e.*, their exists for each a PAR instruction respective *fork node* were it is derived from that has the thread-id $t$. In general there are several *fork nodes* in $t$, so two children could originate from different PAR* constructs respective *fork nodes*. We denote the children's *fork nodes* as $t.forks$. The children of a thread-id can now be described as the union of all the thread-id children of the *fork nodes*:

$$t.children \;=\; \dot{\bigcup}_{f \in t.forks} f.subthreads.$$

The information from which *fork node* a thread-id is derived is important to define the two KEP thread-id relations *concurrent* and *sequence*. But first we define the notion of a *thread-id path* or *thread path*, which is used to define the *sub-thread* relation.

**Definition 4.7** (KEP Thread-Id Path)**.** *A vector of KEP thread-ids* $p = (p_0, \dots, p_{n-1})$ *is called KEP thread-id path, short thread path, if they form a chain via the parent relation:*

$$\forall i = 1, \dots, n-1 : p_i = p_{i-1}.parent.$$

*The set of all thread paths (of a given CKAG C) is denoted with* $\mathbb{P}$ *and* $\mathbb{P}_C$*, respectively, and the length of a thread path* $p$ *with* $|p|$*.*

To make meaningful statements about the KEP thread-ids, we define two axioms: a thread-id is never its own child, and paths have a finite length, *i. e.*, they have no cycle. It could be proven by taking the Esterel syntax into account, but at this point, they are independent of Esterel and therefore axioms.

**Axiom 4.8** (KEP Thread-Id Axioms).

$$\begin{aligned}(\textit{child-parent irreflexivity}) \quad & \forall t \in \mathbb{T}, c \in t.children : c \neq t \\ (\textit{cycle free}) \quad & \forall t \in \mathbb{T}, p \in \mathbb{P} :\ p = (t, \dots, t) \ \Rightarrow\ |p| = 1\end{aligned}$$

Next the thread relations beginning with the *sub-thread relation* are defined, followed by conclusions and theorems about them.

**Definition 4.9** (Sub-Thread Relation). *$t_1$ is a sub-thread of $t_2$, denoted $t_1 \leq t_2$ (or $t_2 \geq t_1$), if a path of threads exist from $t_1$ to $t_2$:*

$$t_1 \leq t_2 \ :\Leftrightarrow\ \exists(p_0, \dots, p_{n-1}) \in \mathbb{P} :\ p_0 = t_1 \ \wedge\ p_{n-1} = t_2.$$

*A derived relation is the real sub-thread relation that additionally demands that the threads are different:*

$$t_1 < t_2 \ :\Leftrightarrow\ t_1 \leq t_2 \ \wedge\ t_1 \neq t_2$$

The definition of the sub-thread relation allows now a more elegant definition of the previous defined fork scopes, also called sub-graphs, for $f \in F$:

$$\begin{aligned} f.sub &= \{\, n \in N \mid n.threadid \leq f.threadid\}, \\ f.sub_i &= \{\, n \in N \mid n.threadid \leq f.child_i.threadid\}. \end{aligned}$$

Some properties that follow directly from the basic definition are given now:

**Lemma 4.10** (Conclusions from the Sub-Thread Definition). *Some simple characteristics of the (real) sub-tread relation:*

$$\begin{aligned} (i) \quad & \forall t_1, t_2 \in \mathbb{T} : t_1.parent = t_2 \ \Rightarrow\ t_1 < t_2 \\ (ii) \quad & \forall p = (p_0, \dots, p_{n-1}) \in \mathbb{P} : p_0 < \ \dots\ < p_{n-1} \\ (iii) \quad & \forall t_1, t_2 \in \mathbb{T} : t_1 < t_2 \ \Rightarrow\ t_1 \leq t_2 \end{aligned}$$

*Proof.* (i) Given thread $t_1$ with its parent $t_2$, the vector $(t_1, t_2)$ is a thread path. Threads $t_1$ and $t_2$ are different, because of the first thread axiom.

(ii) Each path consists of child-parent pairs, with (i) follows the proposition.

(iii) Trivial.

□

**Theorem 4.11** (Partial Sub-Thread Order). *The sub-thread relation is partial order, i. e., the following three properties must hold for all $t_1, t_2, t_3$ in $\mathbb{T}$:*

$$\begin{aligned} (\textit{reflexivity}) \quad & t_1 \leq t_1 \\ (\textit{antisymmetry}) \quad & t_1 \leq t_2 \ \wedge\ t_2 \leq t_1 \ \Rightarrow\ t_1 = t_2 \\ (\textit{transitivity}) \quad & t_1 \leq t_2 \ \wedge\ t_2 \leq t_3 \ \Rightarrow\ t_1 \leq t_3 \end{aligned}$$

*Proof.* (reflexivity): Given thread $t_1$, the 1-dimensional vector $(t_1)$ is trivially a thread path from $t_1$ to $t_1$ according to the sub-thread definition.

(antisymmetry): Given sub-threads $t_1, t_2$ with $t_1 \leq t_2$ and $t_1 \leq t_2$, so according to the definition exist thread paths $p = (t_1 = p_0, \ldots, p_{n-1} = t_2)$ and $q = (t_2 = q_0, \ldots, q_{m-1} = t_1)$. This combined results in a path from $t_1$ over $t_2$ to $t_1$ again, whose length is one, because of the cycle-free axiom. The equality of $t_1$ and $t_2$ follows.

(transitivity): Given sub-threads $t_1, t_2, t_3$ with $t_1 \leq t_2$ and $t_2 \leq t_3$, so according to the definition exist thread paths $p = (t_1 = p_0, \ldots, p_{n-1} = t_2)$ and $q = (t2 = q_0, \ldots, q_{m-1} = t_3)$. The interconnected thread path $(t_1 = p_0, \ldots, t_2, \ldots, q_{m-1} = t_3)$ of $p$ and $q$ forms a chain from $t_1$ to $t_3$ according to the sub-thread definition. □

**Definition 4.12** (Concurrent Relation). *Two threads $t_1$ and $t_2$ are concurrent, denoted $t_1 \mid\mid t_2$, if they can be executed in parallel:*

$$t_1 \mid\mid t_2 :\Leftrightarrow \exists t_i \geq t_1, t_j \geq t_2 \, \exists f \in ForkNodes : t_i, t_j \in f.children.threadid \land t_i \neq t_j$$

The sub-thread relation is not total, because concurrent threads are incomparable:

$$\forall t_1, t_2 \in \mathbb{T} : t_1 \mid\mid t_2 \Rightarrow \neg(t_1 \leq t_2 \land t_2 \leq t_1).$$

Therefore the sub-thread relation is in general no order, only if no concurrent threads exist, *i. e.*, the underlying Esterel program is sequential, it is an order, but in that case only the main thread exists.

**Theorem 4.13** (Concurrent Sub-thread Monotony). *The concurrent relation is monotone as to the sub-thread relation:*

$$\forall t_i, t_j, t_1, t_2 \in \mathbb{T} : \ t_i \mid\mid t_j \land t_1 \leq t_i \land t_2 \leq t_j \Rightarrow t_1 \mid\mid t_2.$$

*Proof.* Given concurrent threads $t_i$ and $t_j$ with sub-threads $t_1 \leq t_i$ and $t_2 \leq t_j$. Per definition there exist threads $t'_i \geq t_i$, $t'_j \geq t_j$ and a *fork node* $f$ with $t'_i, t'_j \in f.sub-threads$. Because of the transitivity of the sub-thread relation follows $t_1 \leq t'_i$ and $t_2 \leq t'_j$, so they are concurrent according to the definition (same $f$). □

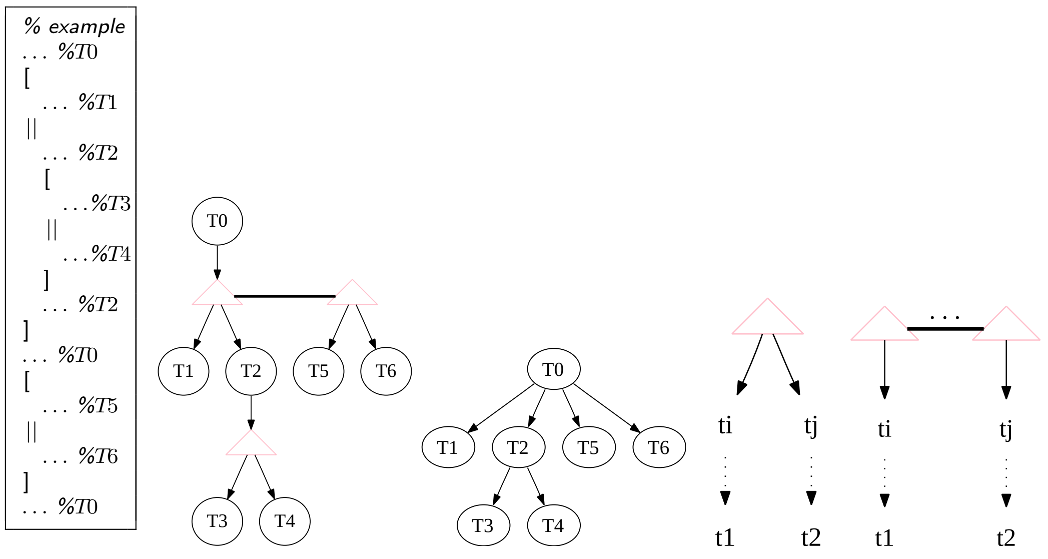
**Definition 4.14** (Least Common Fork). *Given two concurrent threads $t_1$ and $t_2$, a common fork of them is a fork node $f$, whose id is higher within the sub-thread hierarchy:*

$$t_1, t_2 \leq f.threadid$$

*Their least common fork $f'$ is defined as the common fork with the least thread-id according to the sub-thread relation:*

$$\forall f \in CommonForkNodes_{(t_1,t_2)} : f'.threadid \leq f.threadid$$

The *least common fork* exists for all concurrent thread-id pairs, because the main thread is always a *common fork*. The least common fork is well defined thus unique and equal to the fork node $f$ in the definition of the concurrent relation.



```
% example
... %T0
[
  ... %T1
||
  ... %T2
  [
    ...%T3
  ||
    ...%T4
  ]
  ... %T2
]
... %T0
[
  ... %T5
||
  ... %T6
]
... %T0
```

(a) Esterel source, KEP thread-id tree and sub-thread relation (b) $t_1 \mid\mid t_2$ concurrent relation (c) $t_1$ ; $t_2$ sequence relation

Figure 4.2: The KEP thread-id relations: An example KEP thread-id tree and its sub-thread relation is shown in (a). The according Esterel source is schematically presented with accordingly annotated thread-id values. The graphical representations of the *concurrent* and *sequence* relation can be seen in (b) and (c) respectively, whereby the dotted lines represent the sub-thread relation.

**Definition 4.15** (Sequence Relation)**.** *Two threads $t_1$ and $t_2$ are called to be in sequence to each other, if they are in no sub-thread relation and not concurrent:*

$$t_1 \ ; \ t_2 \ :\Leftrightarrow \ \neg(t_1 \leq t_2 \ \vee \ t_2 \leq t_1 \ \vee \ t_1 \mid\mid t_2)$$

*Note that this relation is derived from the sub-thread and the concurrent relation defined before.*

If KEP thread-ids that are neither in sub-thread nor in concurrent relation, they are called to be in sequence. That means they are derived by different *fork nodes*, see Figure 4.2 (c), and therefore cannot be scheduled at the same time. The sub-threads of sequentially defined parallel statements are in sequence relation to each other, but they are also in sequence if the sub-threads are part of conditionally different PRESENT or JW body scopes. The meaning can be described by being not instantaneously active during the KEP thread scheduling.

See Figure 4.2 for an overview about all KEP thread-id relations. There is a schematically described Esterel source, whose three KEP thread-id relations of the resulting KEP thread-id tree are all non-empty. The sub-thread relation is never empty, because the main thread exists always and the sub-thread relation is reflexive. The sub-thread relation of the example is graphically presented in Figure 4.2 (a) without its reflexive and transitive edges, just the parent-child edges are shown.

This principle can be generalized:

**Theorem 4.16** (Reflexive-Transitive Hull Equivalence)**.** *Given KEP thread-ids $\mathbb{T}$ that are coherent by the parent-child relation. If $\mathbb{T}$ is non-trivial, i. e., $|\mathbb{T}| > 1$, then is the reflexive-transitive hull of its parent-child relation identical with the according sub-thread relation.*

*Proof.* $\subseteq$: With Conclusion 4.10 (i) and (iii) is the parent-child relation part of the sub-thread relation and the sub-thread relation is reflexive and transitive as proven in Theorem 4.11.
$\supseteq$: Given sub-threads $t_1 \leq t_2$, $t_1 \neq t_2$, according to the sub-thread definition exists thread path $p \in \mathbb{P}$ composed of parent-child pairs, so is $(t_1, t_2)$ element of the transitive hull. If $t_1 = t_2$, then $(t_1, t_2)$ is part of the reflexive hull, because each thread has a child or a parent, when $\mathbb{T}$ consists of more than the main thread and is coherent. □

In Figure 4.2 (b) and (c) are the general definitions of the KEP thread-id concurrent and sequence relation shown, respectively. They differ by the *fork nodes*, where the KEP thread-ids are derived from: the concurrent relation requests the existence of a common *fork node* and the sequence relation of two different *fork nodes*.

The according relations of the example are:

$$\begin{aligned}
||_{\text{example}} &= (\{T1\} \times \{T2, T3, T4\}) \cup \{(T3, T4)\} \cup \{(T5, T6)\} \\
&= \{(T1, T2), (T1, T3), (T1, T4), (T3, T4), (T5, T6)\} \quad \text{and} \\
;_{\text{example}} &= \{T1, T2, T3, T4\} \times \{T5, T6\} \\
&= \{(T1, T5), (T1, T6), (T2, T5), (T2, T6), (T3, T5), (T3, T6), (T4, T5), (T4, T6)\}.
\end{aligned}$$

As mentioned in Theorem 4.13 is the concurrent relation inherited to its children, so is the complete sub-tree of $T2$ concurrent to $T1$. A similar statement can be made about the sequence relation: the sub-threads of sequential *fork nodes* are all in sequence to each other at a time with two different forks. In the example are two forks defined within the main thread $T0$, their sub-threads are $T1, T2, T3, T4$ and $T5, T6$ respectively, their cross product as a result is part of the sequence relation and failing of additional sequentially defined *fork nodes* it is the full sequence relation.

The id values in the example are denoted by $T\langle\text{id}\rangle$ and are made unambiguous for reasons of simplifications, but in general it is allowed for sequential thread-ids to have the same thread-id value. For the sub-thread and the concurrent relation rules exist that the KEP thread-id values must fulfill to lead to a correct KEP assembler program. These are aggregated by the notion of the *KEP thread-id tree* and are defined in the following, Note that the example described before is already a KEP thread-id tree.

**Definition 4.17** (KEP Thread-Id Tree)**.** *A set of KEP thread-ids $\mathbb{T}$ form a KEP thread-id tree, if he following properties hold:*

*(i)a There is exactly one main thread $m$:*

$$\exists! m \in \mathbb{T} : m.parent \ = \ \perp,$$

*whereby $\perp$ indicates that the main thread has no parent, which therefore is undefined.*

*b $\mathbb{T}$ is be coherent (and cycle-free).*

*(ii) Concurrent threads have different thread-id values:*

$$\forall t_1, t_2 \in \mathbb{T} : t_1 \ || \ t_2 \Rightarrow t_1.id \neq t_2.id.$$

*(iii) Real sub-threads have greater thread-id values than their parents threads:*

$$\forall t_1, t_2 \in \mathbb{T} : t_1 \ < \ t_2 \Rightarrow t_1.id > t_2.id$$

*It follows that the main thread has the lowest thread-id value and this is zero, and the other threads have all thread-id values greater zero.*

*(iv) The id value hierarchy of concurrent KEP thread-ids inherits to their sub-trees:*

$$\forall t_i, t_j, t_{i'}, t_{j'} \in \mathbb{T}, t_i.id < t_j.id : t_{i'} \ \leq \ t_i \wedge t_{j'} \ \leq \ t_j \Rightarrow t_{i'}.id < t_{j'}.id$$

These properties are necessary to match to the KEP thread structure and to ensure the schedulability of KEP assembler programs. Therefore it is recommended that all KEP thread-ids $\mathbb{T}$ of a CKAG are KEP thread-id trees. The first properties (ia) and (ib) ensure together with the cycle-free axiom that $\mathbb{T}$ forms a tree of KEP thread-ids. Property (ii) is needed to distinguish active threads during the scheduling, because if they are concurrent they might be concurrently active. Note that (ii) can not be fulfilled if $\mathbb{T}$ contains a cycle. Property (iii) guarantees that all sub-threads block their parent threads during the scheduling, although they might have the same priority, because parent threads have to wait until all their sub-threads have finished. This and the tree property (iv) are later explained in more detail in the *priority assignment* Chapter 6. There are the scheduling problems discussed and it is illustrated how the priority assignment solves them with the help of the tree properties.

## 4.3 Symbol Scopes

The KEP assembler instructions use several kinds of name strings in their syntax: *signal names*, *variable names* and *address names*. Signal names are used for KEP signals that are defined by INPUT, OUTPUT and SIGNAL instructions and are used in EMIT instructions and several other signal instructions. Variable names are needed

```
var X := 1 : <type> in
  ...
end;
...
var X := 2 : <type> in
  ...
end;
```

```
VAR X

LOAD V,#1
...
LOAD V,#2
...
HALT
```

(a) sequential variable scope

```
[
   var X := 1 : <type> in
     ...
   end;
 ||
   var X := 2 : <type> in
     ...
   end;
]
```

```
VAR X,X_0

    PAR 1,A0,1
    PAR 1,A1,2
    PARE A2,1
A0:
    LOAD X,#1
    ...
A1:
    LOAD X_0,#2
    ...
A2:
    JOIN
    HALT
```

(b) parallel variable scope

Figure 4.3: Variable Names could be reused, if they occur in distinct thread bodies of non-concurrent thread-ids, in (a) name $X$ is reused. If concurrent, another name is used, see (b), where the additionally new name $X_0$ occur. This renaming avoids easily the problem of registers used by threads in parallel.

to represent KEP registers in KEP register instructions, *e.g.*, the valued EMIT or data operations like ADD and MUL. They are globally defined by VAR instructions at program start, even though in there Esterel exist local variables. Each KEP address has an unique address name, defined by the name followed by a colon. They are used as target label by GOTO instructions and to mark the scopes/bodies of ABORT and PAR.

The KEP is case-insensitive. This has to be respected during the creation of new names as well as when using the names resulting from the Esterel source program. As convention the strl2kasm uses internally names only with capital letters, these letter names are the representatives of the string relation *equality modulo letter case*. Esterel is on the other hand case-sensitive, so different names in Esterel could be equal by the KEP:

$$\texttt{"name"} \neq_{\text{Esterel}} \texttt{"Name"} \quad \text{vs.} \quad \texttt{"name"} =_{\text{KEP}} \texttt{"NAME"} =_{\text{KEP}} \texttt{"Name"},$$

whereby $=_{\text{Esterel}}$ and $=_{\text{KEP}}$ are defined as the case-sensitive Esterel and the case-insensitive KEP string comparison, respectively. To distinguish them by the KEP, one name has to be renamed, *e.g.*, from "NAME" to "NAME_0", resulting in two different KEP symbol names:

$$\texttt{"NAME"} \neq_{\text{KEP}} \texttt{"NAME\_0"}.$$

This could lead to the unpleasant situation that input or output signals are renamed, which results in a changed interface on the KEP side. The strl2kasm will throw a warning each time an interface name is changed.

The use of a new name for each new local signal and variable declared in an Esterel source program would be very inefficient in general, because the body scopes might be only a small fraction of the program and are unused in the rest of the program. The solution is to reuse the names and the according signals and registers when the scopes are distinct, *i. e.*, they do not overlap each other. The signals and variables remain different, even though they will use the same name, the strl2kasm therefore uses unique *KEP symbols*. The KEP signals are matched by *signal symbols* and KEP registers by *variable symbols*. Each symbol $s$ has a name $s.name$ that might be used by another symbol of the same type. So it is possible to distinguish different symbols of the same name to not *e. g.*, create falsely a dependency between an EMIT S instruction of one signal scope and a PRESENT S of another different scope. For global signals, namely INPUT and OUTPUT signals, such a problem never occurs.

In addition to distinct bodies as mentioned before, it is demanded that bodies of the same name should not be concurrent. Otherwise it would be possible that despite of distinct bodies, the same signal or variable name is in use for different signals and variables respectively. This leads to the notion of *namespace* to avoid this problem:

**Definition 4.18** (Symbol Namespace)**.** *The namespace of a symbol name $n$ is a set of KEP thread-ids $T_n \subseteq \mathbb{T}$, that contains no concurrent KEP thread-ids:*

$$\forall t_1, t_2 \in T_n : \ \neg(t_1 \mid\mid t_2).$$

Figure 4.3 explains the reuse (a) and the renaming (b) of the variable symbol name $X$ respectively. In (a) the Esterel statement bodies are distinct and part of the same thread, in particular are their KEP thread-ids not concurrent, so name $X$ could be reused. The KEP thread-ids are concurrent in (b) and instead of $X$ is the next free name of type $X_i$, namely $X_0$, used for the second thread. Note that the two different KEP variable symbols remain different, independent of whether they use the same name or not.

Internal registers and signals do not need to be extra defined, they are built-in and therefore there symbols are called *built-in symbols*. Example *built-in registers* are _COUNT and _TICKLEN to define delays and the KEP ticklength respectively. The Esterel tick signal is matched by the built-in signal TICK. The names of built-in symbols are *keywords*, the other symbols are not allowed to carry such names and must be renamed. This problem does not occur for register built-in symbols, because their names start always with an underline and Esterel names have to start with a letter.

The names of Esterel statements, like emit, are keywords in Esterel and therefore cannot be used as signal or variable names in Esterel, as well as by the KEP, because it has a similar syntax, *e. g.*, name EMIT is part of the KEP syntax. But note that the names $\{\mathrm{Emit}, \mathrm{eMit}, \mathrm{emIt}, \mathrm{emiT}, \mathrm{EMit}, \mathrm{eMIt}, \ldots, \mathrm{eMIT}, \mathrm{EMIT}\}$, 15 in number, are all allowed in Esterel, but equals EMIT in KEP as mentioned earlier and therefore must be renamed, *e. g.*, to EMIT_0 if yet free. Other KEP keywords that are not directly based on Esterel syntax, as CMP, JW, ADD are also renamed, if used as symbol name.

# 5 Constructing the CKAG

The CKAG is built from Esterel source via a structural translation by traversing recursively over its AST, which is generated by the CEC [15]. While the Esterel statements are compiled to KEP Assembler (KASM) instructions, the corresponding CKAG is built by creating a node for each instruction, which will be inserted into the graph. The kind of the node depends on the kind of the instruction: for instantaneously executed instructions a *transient node*, for address labels a *label node*, for non-instantaneous executed instructions a *delay node* and for concurrency a *fork node* respectively *join node*.

A node typically contains exactly one instruction, except *label nodes* containing only address labels and *fork nodes* containing one PAR instruction for each child thread initialization and a PARE instruction. While the instructions form an instruction list, which constitutes the later assembler program, their control flow behavior is matched by the nodes in the CKAG.

When a *delay node* is created, additional preemption edges are added according to the (weak) abortion/exception context. *E.g.*, in Figure 5.1 (c) a preemption edge from a PAUSE *delay node* to a *label node* of a strong abort is added because the *delay node* is to be within the body of this strong abort. The *delay nodes* contain mostly PAUSE instructions, since more complex potentially non-instantaneous instructions like AWAIT, SUSTAIN and HALT are —at least initially— dismantled into kernel statements. This is described further in Section 3.3. The creation of the concurrency nodes fork and join is shown in Figure 5.1 (e) and described later in more detail.

The graph building transformation starts at the AST root, which correspond to the Esterel *modules*. Due to the Esterel dismantling, described in Chapter 3, the number of modules is one, holding the main module only with the program as body. If there are after all more than one module, the first is taken as main module. This might be the case when there is an unused module in the program, which cannot be expanded, or the module expanding was not performed at all leading to a compilation error, if a run statement is called. In the following, we assume a correct AST with only one module.

Before the program body is transformed, the Esterel I/O interface is compiled to KEP instruction. This is represented by the root node *C.root* of the resulting CKAG $C$, which at this early building stage consists of this node with only empty edge sets. An Esterel input/output statement is easily transformed into a KEP INPUT/OUTPUT instruction, whereby the according signal symbols are created and renamed if necessary as mentioned in the previous section. Other Esterel interface statements, like sensor, are not supported by the KEP and have to be replaced by input/output statements. See Figure 5.1 (a) to have an overview about how the graph building starts.

Valued I/O signals are defined by an additional V for "valued", namely with INPUTV/OUTPUTV instructions. The according signal value initializations are made by SETV instructions and, if no initialization value is specified, a default value of zero is assigned together with a warning that a variable is not initialized. It might be correct to have no initialization value, when the according signal value is never used until the signal value has been emitted. If the value is accessed yet before set, caused by a wrong Esterel program, the default initialization ensures at least a deterministic program behavior instead of a sudden break of the execution. Thus adding default values leads to a more robust assembler code.

If the module body, *i. e.*, the Esterel program, starts with the declaration of local signals, which therefore are in fact global, their transformed SIGNAL[V] counterparts can be added as an optimization to the KEP interface and then be executed before the program starts. This technique might lower the WCRT of the final KEP program.

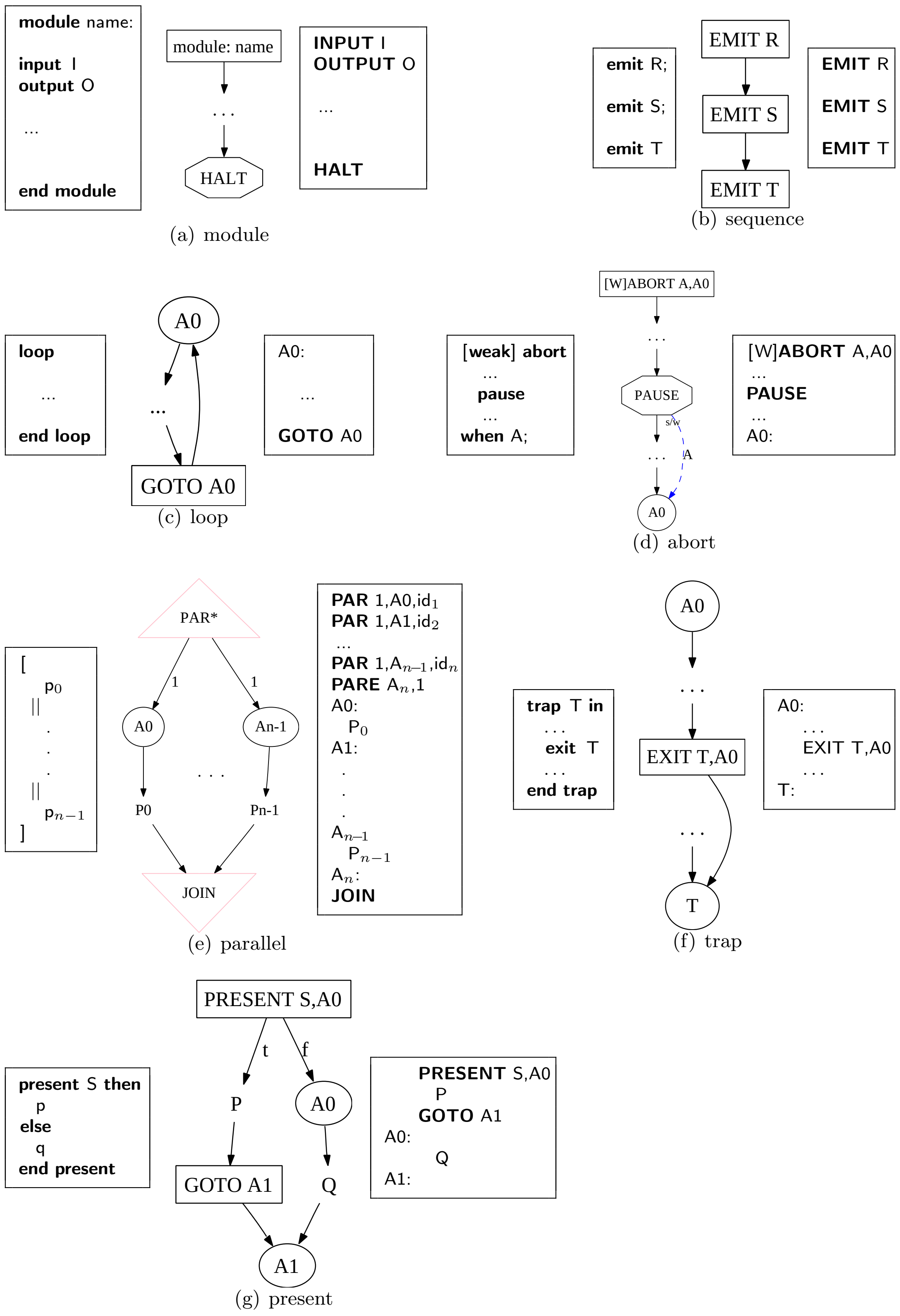
To terminate the program correctly (see Figure 5.1 (a)), a *delay node* with a HALT instruction is added at the end of the transformation. The Esterel halt statements are all dismantled as described in Section 3.3, so no HALT instructions exists in the KEP assembler, at least before performing the collapse optimization of Section 7.2. An exception is the HALT at the of the program, which never will make any priority related problems during the scheduling: its control flow will never aborted and can remain at a constant priority, and in any case it is part of the main thread. The extra termination node might be superfluous, if the program already terminates for all possible input traces. That might be the case, when *e. g.*, the whole program is enclosed by a loop or is a parallel statement list, where a sub-thread on its part never terminates. In such a case the HALT is not added, the ABRO program is an example for that.

Each definition of variables by var statements in Esterel results in VAR instructions, which do not remain local, instead they are globally defined like the INPUT/OUTPUT instructions and therefore belong also to the interface respective root node. As mentioned in the previous section, the variable names might be reused, if the current scope is not concurrent to another one, in which the specific name is already used.

Although the Esterel modules of the AST have a body of Esterel statements, a module is not counted as a *complex statement*, because it cannot be nested. *I.e.*, the bodies contain on their part no module statement, since the modules are handled only once at the beginning. Nevertheless, seen from a parser perspective, the body of the main module is in no way different from the bodies of complex statements.

Esterel distinguishes two different types of statement relations, the Esterel *sequence* and *parallel*, denoted by ; and ||. The sequential statement list is compiled by transforming the statements from the first to the last into a KEP instruction sequence respective CKAG node sequence. This is shown in Figure 5.1 (b). If a statement does not terminate, its statement successors are ignored. Such statements are loop and exit, this way the CKAG remains coherent.

Figure 5.1 (e) depicts the use of KEP concurrency. If threads are defined to be



```
module name:

input  I
output O

 ...

end module
```

```
INPUT I
OUTPUT O

 ...

HALT
```

(a) module

```
emit R;

emit S;

emit T
```

```
EMIT R

EMIT S

EMIT T
```

(b) sequence

```
loop

   ...

end loop
```

```
A0:

   ...

GOTO A0
```

(c) loop

```
[weak] abort
   ...
  pause
   ...
when A;
```

```
[W]ABORT A,A0
...
PAUSE
...
A0:
```

(d) abort

```
[
   p0
  ||
   .
   .
   .
  ||
   pn−1
]
```

```
PAR 1,A0,id1
PAR 1,A1,id2
 ...
PAR 1,An−1,idn
PARE An,1
A0:
  P0
A1:
  .
  .
  .
An−1
  Pn−1
An:
JOIN
```

(e) parallel

```
trap T in
  . . .
  exit T
  . . .
end trap
```

```
A0:
   . . .
   EXIT T,A0
   . . .
T:
```

(f) trap

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

```
     PRESENT S,A0
       P
     GOTO A1
A0:
       Q
A1:
```

(g) present

Figure 5.1: CKAG Building (Esterel source, CKAG, KEP assembler)

in parallel by Esterel, then for each of them a default KEP thread-id value and a start address is assigned via PAR instructions. Each thread gets the initial priority 1 during creation, which might be changed during priority assignment [28] described in Chapter 6. Determining the end of the whole parallel, the PARE defines an *end address*. Within the CKAG the PAR and the PARE instructions are encapsulated by their *fork node*. Compiling an Esterel parallel changes also the current KEP thread-id tree by creating a new sub-thread each time the statement list of an Esterel sub-thread gets transformed. This newly created thread-id is now used for the following creation of nodes. The KEP thread-id structure is not changed by the transformation of all other statements, except for the start, where the main thread with id zero is created.

The children respective control successors of a *fork node* are the start labels defined by PAR instructions. The label defined by PARE indicates the end of the last sub-thread scope, but is never a jump address to execute KEP instructions, so no *label node* is created for this address. The insertion of a PARE *label node* would violate the property that a CKAG should have only one root, since it has no parent.

Many Esterel statements are, at least in its simple forms, directly transformed to their KEP counterparts. *E.g.*, is emit transformed to EMIT, nothing to NOTHING and pause to PAUSE. If they were not be dismantled, then halt, await and sustain would be translated to HALT, AWAIT and SUSTAIN respectively. The translation of more complex statements that have statement bodies, loop, abort, suspend, trap, present and if, is explained in the next section.

## 5.1 Compiling Complex Statements

The building process of complex statements is made recursively, *i. e.*, their bodies are transformed to a sub-graph according to the current environment. The environment persists of preemption successors that are added and erased by starting and ending the build process of the bodies of the preemption statements (weak) abort and trap. Also a body is always built according to the current KEP thread-id $t$, *i. e.*, a newly created node $n$ belongs to $t$: $n.threadid := t$, whereby the current KEP thread-id changes when parallel statement list occur as mentioned earlier. Nevertheless when the body building is completed, the former environment is restored, especially is no potential sub-thread id is used for the continuing building. Next is the compilation of each complex type of statement discussed in detail.

The compilation of the Esterel loop statement and its body is shown in Figure 5.1 (b). The loop body is translated recursively, ahead of the loop a start address A0 and at the end the corresponding GOTO A0 instruction are inserted. In the CKAG the behavior of restarting the loop body is reflected by the edge between the GOTO node and address label. The potentially following statements of the current sequential statement list will be ignored and not transformed, anyhow might building go further through preemption, as explained next.

The building of a (weak) abort statement is presented in Figure 5.1 (d). After the

[W]ABORT instruction is created and inserted into the CKAG, the abort body is handled recursively. This generates preemption edges from all occurring delay nodes to the abort end. This is done for all the preemption statements, namely abort, weak abort and trap. For all of these instructions a preemption edge with the according type and symbol is added. In this example (Figure 5.1 (d)) at least one edge for each delay node is added with preemption type $s$ respective $w$ and signal symbol $A$. How many other edges are added depends on the current environment, *i. e.*, how deeply the different preemption instructions are nested into each other .

If the (weak) abort is *immediate*, then a [W]ABORTI instruction is created instead and in case of the strong abort the abort node is additionally connected with the abort label by a control flow edge. Note that the ABORTI instruction depends on its abort signal, each instruction that emits this signal instantaneously has to be executed previously, the abort node has become a reader to its abort signal.

The abort statement might contain a so called *do body*, *i. e.*, when the abortion takes place, this body has to be executed first. This is realized by inserting the KEP instructions respective the body sub-graph at the abort label. In case that the abort signal does not occur, an additional GOTO is created, which points to an address after the do body.

The Esterel trap transformation is described in Figure 5.1 (f). All according exit statements within the trap scope are translated to EXIT instructions with a control flow edge to its trap label node in the CKAG. Additionally to the trap label a start trap label is created, here named with $A0$, which is used by the EXIT instruction to determine the trap start in order to decide whether different EXIT instructions are nested or concurrent.

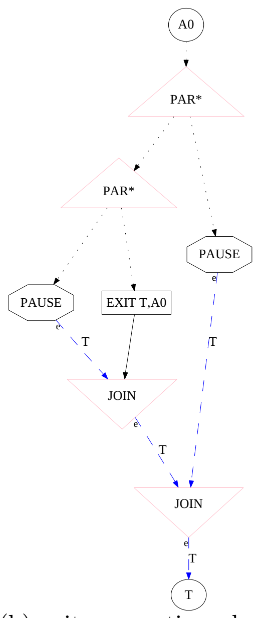
If an EXIT instruction is part of a sub-thread, which itself is part of the according trap scope (and not vice versa), then concurrent threads are also terminated when called. All sub-thread instructions that are instantaneously executable are executed. The control flow stops at the end of the according thread or at a PAUSE instruction, followed by the according JOIN instructions. According to this, exit preemption edges are created for all PAUSE instructions leading to specific *join nodes*, as shown in Figure 5.2. The *join nodes* are executed from inner to outer, also represented as exit preemption edges, this time between *join nodes*. All delayed instructions derived from the trap body depend on the according EXIT instructions. This will be relevant for the priority assignment explained in the next Chapter 6.

Similar to the *do* body of the abort statement, the trap could have a *handle*, which is likewise managed by creating according instructions and nodes at the *trap* label (node).

Esterel distinguishes two types of conditionals, present for signal and if for standard Boolean expressions. Both contain up to two bodies, a *then* and an *else* body. Which one is executed will be determined by the accordingly created PRESENT respective CMP and JW instructions. Figure 5.1 (g) pictures the creation of the PRESENT. The conditional jump of the PRESENT is matched by two control flow edges in the CKAG, annotated with $t$ and $f$ depending on whether the condition was $true$ or $false$. The edge to the *else address*, defined by PRESENT, here $A0$, is labeled with

```
trap T in
  ...
  [
   .
    .
     .
      [
        ...
        exit T
        ...
      ]
     .
    .
   .
  ];
  ...
end trap
```

(a) trap over concurrent Esterel statements



(b) exit preemption edge hierarchy

```
A0:
      ...
      PAR 1,A_i0,id
      ...
      PAR 1,A_i(n-1),id
      PARE A_in
A_i0:
      ...
      PAR 1,A_j0,id
      ...
      PAR 1,A_j(k-1),id
      PARE A_jk
A_j0:
      ...
      EXIT T,A0
      ...
A_jk:
      JOIN
      ...
A_in:
      JOIN
      ...
T:
```

(c) according KEP assembler instructions

Figure 5.2: If an EXIT terminates concurrent KEP instructions, this is displayed by exit preemption edges directed to the according *join nodes*, that form on their own a join preemption hierarchy ended by the most outer *join node* having an exit preemption edge to the trap label.

$f$, and therefore the *else* body is built at this label. The *then* body is transformed directly after the PRESENT node. Behind the *else* instructions is a GOTO inserted to skip the following *else* body instructions. This might be unnecessary, when no *else* body exists at all, or the control of the *then* case remains in a loop. If no *then* body is specified (present ... else ... end), the *true* edge leads directly from the PRESENT to the *else* body skipping GOTO.

The compilation of the if uses the JW instruction, instead of PRESENT. The register operation CMP is previously performed to test one of the following criteria by JW: L, LE, G, GE, EE and NE. These are the KEP pendants of lower/greater (equal) and equal/unequal, see Section 2.2. *E.g.*, the if expression ($X > 5$), which tests whether the register $X$ has a value greater five, is translated to CMP $X$,#5 to compare $X$ with five, followed by JW G,$\langle else_a ddr\rangle$, to test the compare onto greater by G and jump accordingly.

Both, the present and if, might contain arbitrarily nested Boolean expressions. These result in a more complicated testing procedure and are explained in the next section and Figure 5.3.

The Esterel local signal declaration by signal contains a body, but the KEP provides no constructs to limit the scope of a local KEP signal. Instead this is handled by the compiler by freeing the signal name for further use after the signal body has been transformed. In contrast to the VAR instructions, which are always added to the interface, the newly created SIGNAL[V] instructions are inserted into the KEP assembler respective CKAG. The signal name has to be a free, respecting the scope rules of Section 4.3.

## 5.2 Handling Expressions

Esterel expressions appear in several Esterel statements of several types: integer expressions in computations or as a delay of a signal expression, signal expressions in abort and present, expressions of type Boolean in if statements. Since the KEP cannot handle complex expressions in statements by its instruction set architecture, the strl2kasm compiler has to make the computations explicit.

The Esterel expressions are pre-processed to so called *KEP expressions* and then inserted into the according KEP assembler instruction. If the expression is not a literal or atomic, and therefore does not comply with the KEP assembler syntax, it is further processed to an instruction that contains only atomic expressions. In this sense the KEP expressions are used as low-level intermediate data structure. To denote the KEP expressions, we use capital letters and, if applicable, the according register operation names. In addition to this the polish notation is applied, *e. g.*, the Esterel expression $(X + 5)$ is transformed to the KEP expression (ADD $X$ 5).

As first step, the compiler replaces all constants specified in the Esterel program by its literal values. Therefore they must all be defined in the Esterel source. If not, the compiler will throw an error message and stop. Simultaneously the binary Esterel operations that are *associative* are made flat, therefore the KEP expression structure supports over beyond *unary* and *binary* operations of an *associative operation*, *e. g.*, the expression $(2 + (X + 5))$ is flattened to (ADD 2 X 5).

Next, if possible, the occurring literal values are aggregated according to their specific operation. *E.g.*, the KEP expression (ADD #2 X #5) is computed to (ADD 7 X). This step is known as *constant propagation* [2]. This is a recursive process beginning from more operations by a DFS algorithm to the higher ones. Sub-operations might be in this way replaced by a literal allowing a further evaluation. *E.g.*, the KEP expression (ADD #1 X (MUL #2 #3)) is first reduced to (ADD #1 X #6), which makes the application of ADD possible and results in (ADD #7 X).

A further evaluation of KEP expressions could take advantage of the behavior of *neutral* and *dominant elements*. Given a binary operation *op*, $e$ is a neutral element of *op* if it leaves all other operands $x$ unchanged: $(e\ op\ x) = x$. So we can skip these elements in KEP expressions. The data operations ADD and MUL and the Boolean operations AND and OR have the neutral elements #1 and #0 respective true and false, whereby true/false are represented by #1 and #0. The AND polymorphically used in signal operations has also a neutral element, namely the TICK signal. For example, the expression (ADD $?V$ #0 $X$) can be reduced to (ADD ?V $X$).

```
PRESENT (NOT S),E
```

$\rightsquigarrow_{build}$

```
        PRESENT S,A0
        GOTO E
A0:
```

(a) NOT expressions

```
PRESENT (AND S₁ ... Sₙ),E
```

$\rightsquigarrow_{build}$

```
PRESENT S₁,E
...
PRESENT Sₙ,E
```

(b) AND expressions

```
PRESENT (OR S₁ ... Sₙ),E
```

$\rightsquigarrow_{build}$

```
        PRESENT S₁,E₁
        GOTO A0
E₁:
        . . .
Eₙ₋₂:
        PRESENT Sₙ₋₁,Eₙ₋₁
Eₙ₋₁:
        GOTO A0
        PRESENT Sₙ,E
A0:
```

(c) OR expressions

Figure 5.3: The compilation of all three possible Boolean operations that could occur in PRESENT instructions. Note that the signals $S_i$ might be itself again a Boolean expression.

A dominant literal $d$ is the opposite of an neutral element. If applied the result is always $d$, independent of other operands: $(d\ op\ x) = d$. The MUL operation has the dominant value #0, the Boolean AND/OR have #0 respective #1, and the TICK signal dominates the OR signal operation. *E.g.*, (MUL $?V$ #0 $X$) can be reduced to #0. Note that not for all operations a neutral or dominant value exists, *e. g.*, ADD has no dominant integer value.

## 5.2.1 Compiling Signal Expressions

Signal expressions are used in signal statements when occurring in abort and suspend they could be *delayed* or *immediate*, in present only Boolean operations are allowed. The KEP syntax directly supports immediate instructions, marked by an additional I. Therefore KEP expressions are never immediate, but the assembler instructions are. The same for delays of signal expressions, they are directly transformed to a LOAD register operation to initialize the built-in signal _COUNT with the delay expression. The delay expression might be a complex operation that might be compiled to several other register instructions, see the next Section 5.2.2.

Boolean signal operations do not exist in abort statements, at least at this compilation step, because the Esterel dismantling procedure of Chapter 3 handles them. So if there is a Boolean abort signal expression in the original program, it is now present in a present statement together with an atomic expression in the abort, see Figure 3.5. How Boolean present operations are handled is explained next, whereby these expressions are processed to possibly associative KEP expressions as explained before.

$$\begin{tikzcd}\boxed{\textbf{emit}\ \mathsf{S(3)}} \arrow[r, squiggly, "build"'] & \boxed{\textbf{EMIT}\ \mathsf{S,\#3}}\end{tikzcd}$$

(a) literal

$$\begin{tikzcd}\boxed{\textbf{emit}\ \mathsf{S(?T)}} \arrow[r, squiggly, "build"'] & \boxed{\begin{array}{l}\textbf{EMIT}\ \mathsf{S}\\ \textbf{SETV}\ \mathsf{S,?T}\end{array}} \arrow[r, squiggly, "build"'] & \boxed{\begin{array}{l}\textbf{EMIT}\ \mathsf{S}\\ \textbf{LOAD}\ \mathsf{REGO,?T}\\ \textbf{SETV}\ \mathsf{S,REG0}\end{array}}\end{tikzcd}$$

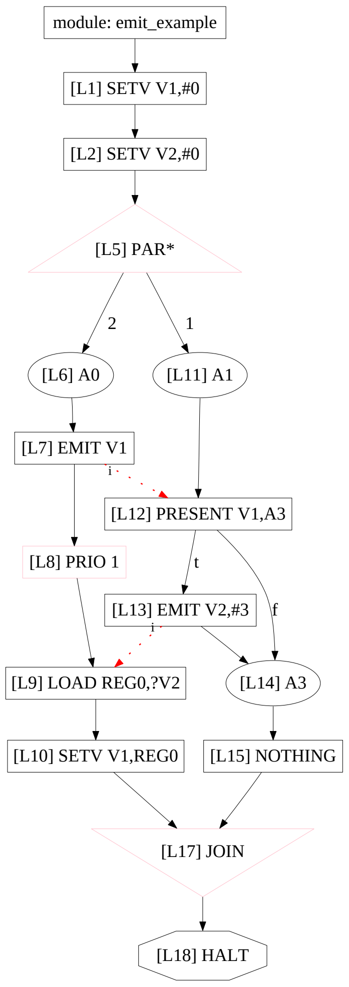
(b) signal value

$$\begin{tikzcd}\boxed{\textbf{emit}\ \mathsf{S(?T+3)}} \arrow[r, squiggly, "build"'] & \boxed{\begin{array}{l}\textbf{EMIT}\ \mathsf{S}\\ \textbf{SETV}\ \mathsf{S,(}\textbf{ADD}\ \mathsf{?T\ \#3)}\end{array}} \arrow[r, squiggly, "build"'] & \boxed{\begin{array}{l}\textbf{EMIT}\ \mathsf{S}\\ \textbf{LOAD}\ \mathsf{REGO,?T}\\ \textbf{ADD}\ \mathsf{REG0,\#3}\\ \textbf{SETV}\ \mathsf{S,REGO}\end{array}}\end{tikzcd}$$

(c) data operation

```
module emit_example:

input  A;

output V1 : integer ;
output V2 : integer ;


[
   emit V1(?V2)
  ||
   present V1 then
     emit V2(3)
   end present
]

end module
```

```
INPUT A
OUTPUTV V1,V2
VAR REG0
EMIT _TICKLEN,#14

[L01,T0,P2]          SETV V1,#0
[L02,T0,P2]          SETV V2,#0
[L03,T0,P2]          PAR 2,A0,1
[L04,T0,P2]          PAR 1,A1,2
[L05,T0,P2/2]        PARE A2,2
[L06,T1,P2]     A0:
[L07,T1,P2]          EMIT V1
[L08,T1]             PRIO 1
[L09,T1,P1]          LOAD REG0,?V2
[L10,T1,P1]          SETV V1,REG0
[L11,T2,P1]     A1:
[L12,T2,P1]          PRESENT V1,A3
[L13,T2,P1]          EMIT V2,#3
[L14,T2,P1]     A3:
[L15,T2,P1]          NOTHING
[L16,T0,P2]     A2:
[L17,T0,P1]          JOIN
[L18,T0,P1/1]        HALT
```



(d) Priority Dismantling on KEP level

Figure 5.4: Several valued EMIT transformations: An EMIT with a literal value remains unchanged (a), as a signal value has to be stored in a register in order to use its value (b) and in (c) is a computational expression split into register operations before its result can be stored and emitted. In (d) is an example program shown, where the break between emitting and value setting parts avoid a program cycle.

Three types of Boolean signal expression operations are distinguished by the Esterel present statement, the not, and and or, corresponding to the KEP NOT, AND and OR operations. Hence the KEP assembler syntax only supports atomic signals. These operations are further simplified to even atomic PRESENT instructions, as shown in Figure 5.3. The NOT can easily consumed by switching then *then* and *else* body during the building process. However, if it is not at the top of the signal operation, the switching is realized by a GOTO as seen in Fig. 5.3 (a). In Fig. 5.3 (b) the PRESENT AND operation is shown. This is simply translated to a chain of PRESENTS instructions of the operands $S_i$, all leading to the same *else* label $E$, when the PRESENT test fails. The OR operation, explained in Fig. 5.3 (c), needs more instructions, since for the first $n-1$ operands each time a GOTO instruction is created. These lead all to the *then* case body at a newly created label, here $A0$, after the last PRESENT test. The *else* labels $E_i$, $i = 0, \ldots, n-1$, all lead to the next PRESENT test. When the last one also fails, the overall *else* label $E$ is reached and the *else* body executed.

Although the signal expressions of emit are always atomic, according to the Esterel semantics, there could be a complex data operation value when the emit signal is valued. The emitting of a simple literal value can be applied by adding the value directly after the signal of the EMIT instruction (Figure 5.4). Such a literal value might originate from a data expression by performing the pre-processing steps described before, especially from evaluation.

A KEP data expression might contain the signal values of other valued signals, the computation therefore depend on these signals. We call those valued signal *readers* to such signals. The other way round the EMIT instruction must always be executed, before the corresponding PRESENT instructions are called, so it is always a *writer* to its signal. Other computations might also need the emitted signal value. This property of a valued EMIT of being writer and reader at the same time might lead to priority scheduling problems, as described in Section 3.3 in case of the SUSTAIN.

The same solution is possible for the valued EMIT. The writing part is separated to a simple non-valued EMIT followed by a SETV instruction, containing the EMIT value expression, which is the reading part, to set the signal value accordingly. This can be seen as a *priority dismantling on KEP level*. This principle is used when emitting *e. g.*, another signal value, see Figure 5.4 (b), or computation, see (c). In both cases the reading part, here $?V$, is following the EMIT. Nevertheless is the SETV a writer to the signal value and other statements might depend on it.

A more complex example is shown in Figure 5.4 (d). Two valued emissions of valued signals $V1$ and $V2$ occur, $V1$ emits the signal value of $V2$ and $V2$ emits a value of #3 when $V1$ is active. So the emissions depend on each other: $V1$ must be active to emit the correct signal value of $V2$, and to emit $V1$, signal $V2$ has already been active. After the EMIT $V1$, $?V2$ is split as described before. The conflict is solved, hence the EMIT $V1$ runs with a higher priority as the SETV assignment part. See also the according CKAG: a PRIO instruction is inserted in between, lowering the priority behind the EMIT. The resulting execution trace is as follows: when the threads were created, $T1$ has the higher priority of 2 and therefore executes the EMIT $V1$ and PRIO 1 instructions. So $V1$ is now active and the control switches to thread $T2$, which has a higher thread-id value. The *then* body of PRESENT $V1$, namely EMIT $V2$, $\#3$, gets executed. After $T2$ has finished, thread $T1$ is again executed and the signal value of $V1$ is set to $?V2$.

## 5.2.2 Compiling Data Expressions

Data expressions are of type integer or Boolean and can occur as the delay of a signal expression, in variable/register assignments and in if comparisons. The KEP currently does not support floating point operations, but the compilation scheme described here in the following would be the same. This section presents the compilation of data operations. The atomic data expressions, namely literals, signal values and registers, remain unchanged.

The building of a computation is made according to some result register. Given a KEP operation *op*, its first argument is loaded by the KEP LOAD register instruction. For each remaining operand an *op* register instruction is created to perform the operand on the result register. *E.g.*, the KEP expression (ADD $X$ $\#2$ $Y$) is transformed to instruction LOAD REG0, $X$ followed by ADD REGO, $\#2$ and ADD REGO, $Y$. The register REGO is a temporary result register that is freed after use. In general these registers are REG$i$, $i \in setN$, and have to respect the symbol namespace described in Section 4.3. Such an expression might be used as a value of a valued signal, see Figure 5.4 (c). Before being freed the REG0 register is used to set the signal value.

A special case is a register assignment, where the result register is part of the computation. Then no temporary register is needed and the result register has not to be initialized. *E.g.*, the Esterel statement $x\ :=\ x + 1$ is translated to ADD $X, \#1$.

The data expressions might be arbitrarily nested. Building these is done like the evaluation step, namely via DFS. Consider the KEP expression (ADD (MUL $\#2$ $X$) (MUL $\#3$ $?V$)): first the sub-expression (MUL $\#2$ $X$) is built by instructions LOAD REG0, $\#2$ and MUL REG0, $X$, and then replaced by REG0, resulting in (ADD REG0 (MUL $\#3$ $?V$)). Likewise (MUL $\#3$ $?V$) is built and replaced by REG1. In the last step expression (ADD REG0 REG1) is realized with result REG0 by building instruction ADD REG0,REG1, hereafter the name REG1 is freed. Note that REG0 is not initialized at the end, because LOAD was already applied as the previous MUL were built.

Esterel supports the use of external *functions* written in a host language. The strl2kasm compiler supports these functions, if their code is already transferred to KEP assembler, with the CALL instruction. The arguments of a function call are loaded to the so called *function interface registers* _TMP$i$ by LOAD instructions. After CALL is executed, the result again is hold by these registers. Note that the _TMP$i$ registers are built-in registers like _COUNT.

The if statement is very similar to the present, but contains *Boolean (data) expressions*, instead of signal expressions, with all kind of comparative operations: greater/lower (equal) and (in)equality tests. The basic comparison is already explained in the previous Section 5.1 with the focus of building the *then* and *else* bodies.

The building of Boolean signal expressions is shown in Figure 5.3. It works as for if statements, except that now are three instructions needed instead of one PRESENT: the first to initialize a temporary register REG0 by the specific value, which then is compared to true by CMP REG0,#1 before the conditional jump is built by JW EE,$\langle else_a ddr\rangle$. The comparative expressions are integrated into this principle by setting a result register by JW according to the comparison.

All KEP instructions, especially the register operations that are used to compile the Esterel expressions as described before, are limited to 16 bit integers. The KEP provides a built-in register to use integer values up to 32 bit, namely _UINT32REG. Given a 32 bit integer number $i \in \mathbb{N}$ in a register operation *reg_op reg*, $\#i$, the 32 bit value $i$ is replaced by register _UINT32REG. This results in the following instruction instead: *reg_op reg*, _UINT32REG. The value $i$ is previously loaded to register _-UINT32REG by the DEF32 instruction: DEF32 $\#i$. To ensure the correct use of 32 bit values, the compiler checks all register operations for integer values and replace them accordingly if needed.

# 6 Priority Assignment

A main problem for compiling Esterel onto the KEP is the need to manage the thread priorities to ensure the correct execution order. If an instruction depends on other instructions then those must be executed prior and thus have a higher priority. Therefore these priorities have to be assigned properly during creation and further execution. In the KEP setting, this is not merely a question of efficiency, but a question of correct execution. The computation of these priorities is called *priority assignment* [28].

Priorities are assigned during the creation of a KEP thread by PAR instructions and by a particular PRIO instructions. Due to the non-linear control flow, it is still possible that a given instruction may be executed with varying priorities; in principle, the architecture would allow a fully dynamic scheduling. However, it is assumed that the given Esterel program can be executed with a statically determined schedule, which requires that there are no cyclic signal dependencies. This is a common restriction, imposed for example by the Esterel v7 [19] and the CEC [15] compilers. Note that there are also Esterel programs that are causally correct (*constructive* [8]), yet cannot be executed with a static schedule and hence cannot be directly translated into KEP assembler using the approach presented here. However, these programs can be transformed into equivalent, acyclic Esterel programs [30], which can then be translated into KEP assembler. Hence, the actual run-time schedule of a concurrent program running on the KEP is *static* in the sense that if two instructions that depend on each other, such as the emission of a certain signal and a test for the presence of that signal, are executed in the same logical tick, they are always executed in the same order relative to each other, and the priority of each instruction is known in advance. However, the run-time schedule is *dynamic* in the sense that due to the non-linear control flow and the independent advancement of each program counter, in general it cannot be determined in advance which code fragments are executed at each tick. This means that the thread interleaving cannot be implemented with simple jump instructions. Instead a run-time scheduling mechanism is needed that manages the interleaving according to the priority and actual program counter of each active thread.

This leads to the *priority assignment*. Each KEP instruction respective CKAG node are one or two priorities assigned, dependent on the type and behavior of the control flow:

**Definition 6.1** (Node Priorities). *For all nodes $n \in N$ the according priority* **n.prio** $\in \mathbb{N}$ *is defined as the priority that the thread n.threadid should be running with, when executing n.*

*For delayed nodes $n \in D \cup J$ additionally a priority* **n.prionext** $\in \mathbb{N}$ *is defined, the priority the according instruction has to be executed with, after the delay.*

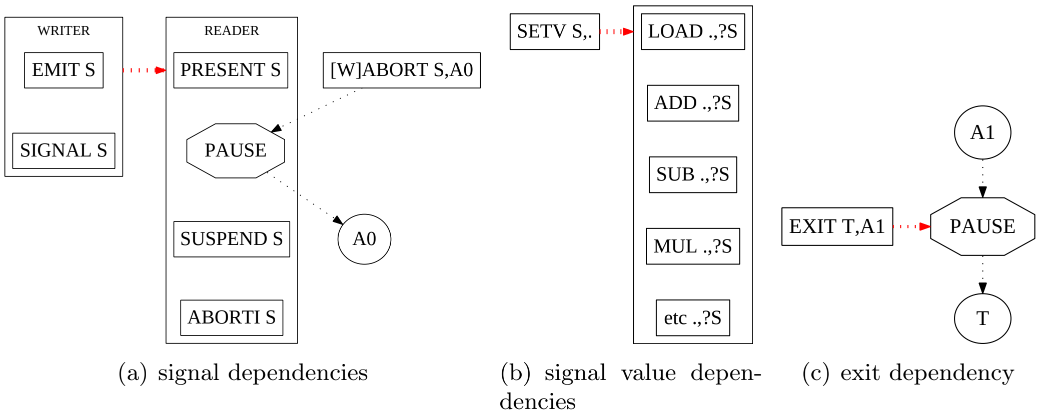
Given a *join node* $j$, the assigned priority *j.prionext* represents the priority that thread *j.threadid* must have after the complete parallel construct $(j.fork, j)$ has terminated non-instantaneously. If a parallel is always instantaneously executed, no next priority is needed for the specific *join node*. That is the case, when the delayed successors of $j$ are empty, see Section 4.1.

The aggregate of computed priorities *n.prio* and *n.prionext* is named as the *priority assignment* of a given CKAG. The priority assignment is in graphical representations denoted with preceded "P" for all nodes $n \in N$. Thus, their according priorities *n.prio* and *n.prionext* have the following form: $P\langle n.prio\rangle$ respective $P\langle n.prio\rangle/\langle n.prionext\rangle$.

To obtain a more general understanding of how the priority mechanism influences the order of execution, recall that at the start of each tick, all enabled threads are activated and subsequently scheduled according to their priorities. Furthermore, each thread is assigned a priority upon its creation by a PAR instruction. Once a thread is created, its priority remains the same, unless it changes its own priority with a PRIO instruction. In that case it keeps that new priority until it executes yet another PRIO instruction, and so on. Neither the scheduler nor other threads can change its priority. Note also that a PRIO instruction is considered instantaneous. The only non-instantaneous instructions, which delimit the logical ticks and are also referred to *delayed* instructions, are the PAUSE instruction and derived instructions, such as AWAIT, SUSTAIN and HALT, although these were yet dismantled and does not occur at this compilation step. This mechanism has a couple of implications:

- At the start of a tick, a thread is resumed with the priority with which it was terminated. This could be the last thread's PRIO instruction executed during the preceding ticks, or the priority it has been created with if it has not executed any PRIO instructions. In particular, if we must set the priority of a thread to ensure that at the beginning of a tick the thread is resumed with a certain priority, it is not sufficient to execute a PRIO instruction at the beginning of that tick; instead, we must already have executed that PRIO instruction in the preceding tick.

- A thread is executed only if no other active thread has a higher priority. Once a thread is executed, it continues until a delayed instruction is reached, or until its priority is lower than that of another active thread or equal to that of another thread with higher *id* value. While a thread is executing, it is not possible for other inactive threads to become active; furthermore, while a thread is executing, it is not possible for other threads to change their priority. Hence, the only way for a thread's priority to become lower than that of other active threads is to execute a PRIO instruction that lowers its own priority below that of other active threads.



(a) signal dependencies (b) signal value dependencies (c) exit dependency

Figure 6.1: The types of *signal dependencies*. The PAUSE instruction is only a reader, when part of an ABORT respective trap scope, see (a) and (c). The SETV instruction manipulates a signal value, so it is only writer of signal value manipulating instructions.

## 6.1 Signal Dependencies

This section defines the *dependencies*, introduced for the priority dismantling in Section 3.3, in detail. KEP instructions that manipulate a signal or its value are called *writer* or *dependency source*. Instructions that use a signal (value) are called *reader* or *dependency sink*. Which instructions are writer or reader is defined in the following.

**Definition 6.2** (Writer and Reader Set). *The set of writers $W_s \subseteq N$ respective readers $R_s \subseteq N$ is defined for a given a signal symbol $s \in \mathbb{S}$ as follows:*

$$\begin{aligned} W_s &:= \{n \in N \mid n.stmt = \textsf{EMIT}\ s,\ \textsf{SIGNAL}\ s,\ \textsf{SETV}\ s,.,\ \textsf{EXIT}\ s\} \\ R_s &:= \{n \in N \mid n.stmt = \textsf{PRESENT}\ s,\ \textsf{SUSPEND}\ s,\ \textsf{ABORTI}\ s,\ reg\_op\ .,?s,\ \textsf{PAUSE}\}, \end{aligned}$$

*where $reg_op \in \{$LOAD, ADD, SUB, MUL, ANDR, ORR, XORR, CMP$\}$ and the PAUSE instructions in $R_s$ are part of an abort or trap scope, i. e., it exists an abort or trap node $n$ with PAUSE $\in n.scope$ according to signal $s$.*

*The writers and readers of a CKAG are denoted with $W$ and $R$, and are defined as the unions about all $W_s$ and $R_s$, respectively:*

$$W := \dot{\bigcup}_{s\in\mathbb{S}} W_s \qquad \textit{and} \qquad R := \dot{\bigcup}_{s\in\mathbb{S}} R_s.$$

A tuple of writer and reader form regarding to the same signal a *signal dependency*. Relevant for the scheduling are only dependencies between concurrent instructions, therefore they are defined accordingly:

**Definition 6.3** (Signal Dependency). *The set $\mathbb{D}_s \subseteq W_s \times R_s \subseteq N \times N$ of concurrent writer-reader tuples of signal symbol $s \in \mathbb{S}$ is the set of signal dependencies of s:*

$$\forall S \in \mathbb{S}: \; \mathbb{D}_S := \{(w, r) \in W_S \times R_S \mid w.threadid \mid\mid r.threadid\},$$

*where it is required that* SETV *writers have only signal value readers:*

$$\forall (w, r) \in \mathbb{D}_S: \; w.stmt = \textsf{SETV}\, S \;\Rightarrow\; r.stmt = \textsf{reg\_op}\; .,?S.$$

*To indicate that a dependency $(w, r) \in \mathbb{D}_S$ belongs to signal $S$, it is denoted as $(w, r)_S$.*

*The set of all signal dependencies $\mathbb{D} \subseteq W \times R$ is the union of all $\mathbb{D}_S$:*

$$\mathbb{D} := \dot{\bigcup}_{S \in \mathbb{S}} \mathbb{D}_S.$$

*The readers of a node n are called the* dependency successors *of n and written as n.dep:*

$$n.dep := \{r \in N \mid (w, r) \in \mathbb{D}\}.$$

All possible combinations of writer and reader instructions that form signal dependencies are shown in Figure 6.1. All writers are on the left in (a) and (b) and form a dependency with the readers on the right, denoted by a dotted (red) line. An abort node is only a reader if it is immediate. A PAUSE instruction is a reader only if it is in the scope of an abort node. This is denoted in the figure by dashed (blue) edges between abort and corresponding abort label nodes xthat represent the enclosing ABORT construct. The PAUSE might also be a reader to trap signals of EXIT instructions (c), here the PAUSE is part of a trap scope.

Note that no instructions/nodes of the main thread are part of a signal dependency: The main thread is never concurrent to another thread, because all other threads are sub-threads of the main thread. Therefore all writer and reader nodes have a KEP thread-id different from the main thread, with an id value greater than zero.

Non-concurrent dependencies are not defined as dependencies, even though their writers must be executed prior, too. For sequential instructions the execution order is fixed, as mentioned, therefore it is not possible to affect their order by priorities. If they have a wrong order, the Esterel source is invalid. We assume acyclic Esterel programs on the input side of the compiler. The strl2kasm performs no analysis for Esterel correctness, this has to be done at the Esterel level before the strl2kasm is used.

In the definition of writers $W$ and readers $R$ occur no PAR, PARE and JOIN instructions. Therefore fork and *join nodes* are never writers or readers and thus never part of a dependency.

It turns out that analogously to the distinction between *prio* and *prionext*, we must distinguish between dependencies that affect the current tick and the next tick:

**Definition 6.4** (Delayed and Immediate Dependency Successors). *A dependency $(w, r) \in \mathbb{D}_S$ is called* delayed *if r is a delay node in between the scope of a strong* *abort node over $S$, i. e., for reader $r$ exists a strong abort successor $n'$ belonging to signal $S$. Therefore the set of delayed dependencies is defined accordingly:*

$$\forall S \in \mathbb{S}: \ \mathbb{D}_{S,d} := \{(w,r) \in \mathbb{D}_S \mid \exists (r,n')_{(s,S)} \in \mathbb{P}\}.$$

*The delayed dependency successors of node $n \in N$ are defined analogously to $n.dep$:*

$$n.dep_d := \{r \in N \mid (w,r) \in \mathbb{D}_{.,d}\}.$$

*The remaining successors are called* immediate dependency successors *and the according dependencies are called* immediate dependencies*:*

$$n.dep_i := n.dep \setminus n.dep_d \qquad \textit{and} \qquad \mathbb{D}_{S,i} := \mathbb{D}_S \setminus \mathbb{D}_{S,d}.$$

Note that only dependencies that have a *delay node* $d \in D$ as reader can be delayed, due to the fact that only *delay nodes* have strong abort preemption successors. The other way round are dependencies $(w,r) \in \mathbb{D}$ always immediate, if $r \in N \setminus D$.

A CKAG is not allowed to have cycles in the graph induced by the instantaneous successors of its nodes, as mentioned in Chapter 4. This correlates with the ban of instantaneous loops in Esterel and allows a proper WCRT analysis [12] on the CKAG. This non-cycle requirement is now extended by taking the dependencies into account: a program is considered *cyclic* if the priority assignment algorithm presented in this chapter fails. This leads to the following definition of a *program cycle*:

**Definition 6.5** (Program Cycle). *An Esterel program is cyclic if the corresponding CKAG contains an instantaneously executable cyclic node path $p = (n_0, \ldots, n_k) \in N^k$, $k \geq 2$ with $n_0 = n_k$, whereby also the dependency successors are included:*

$$\begin{aligned} \forall i = 0, \ldots, k-1 \quad : \quad & p_{i+1} \in p_i.suc_{inst} \cup p_i.dep_i \ \vee \\ & p_{i+1} \in p_i.dep_d \ \Rightarrow \ p_{i+2} \in p_{i+1}.suc_{delayed} \ \vee \\ & p_{i+1} \in p_i.suc_{delayed} \ \Rightarrow \ p_i \in p_{i-1}.dep_d, \end{aligned}$$

*that means for $p$ that all successors are instantaneous or dependency related, where after delayed dependencies only delayed successor follow, and vice versa.*

Note that this definition is conservative, because the definition of the instantaneous and delayed successors of fork-*join node* pairs is made conservatively. However, what exactly constitutes a cycle in an Esterel program is not obvious and there exist different, but accepted, definition of cyclicality at the language level. Esterel compilers that require acyclic programs differ in the programs they accept as 'acyclic': the CEC accepts some programs that the v5 compiler rejects and vice versa [34].

Due to the fact that Esterel has a loop construct, the CKAG in general contains cycles, non-instantaneous cycles or non-program cycles, which are *valid*. The question is how the constraints allow an assignment for such valid cycles. The answer are the *prionext* priorities. Consider a (valid) cycle $(d, n_1, \ldots, n_{k-1}, d)$ that starts and ends with a *delay node d*. The induced constraint chain starts and end with priorities *d.prionext* and *d.prio* respectively, thus the priorities can decrease alongside the constraint chain from *d.prionext* to *d.prio*.

In the absence of a program cycle, as defined here, the priority assignment should be able to fulfill all *constraints* derived from node and dependency successors, as defined in the next section.

## 6.2 Priority Constraints

The task of the priority assignment algorithm is to compute a priority assignment that respects the Esterel semantics as well as the execution model of the KEP. The algorithm computes for each reachable node $n$ in the CKAG the priority $n.prio$ and, for nodes in $D \cup J$, $n.prionext$. According to the Esterel semantics and the observations made in Chapter 4, a correct priority assignment must fulfill the following constraints, starting with the *dependency constraints* implicated by the signal dependencies:

**Definition 6.6** (Dependency Constraints)**.** *An ordering $>_{id}$ between assigned priorities of concurrent nodes $n, m \in N$ is defined, called "thread-id greater", that respects the KEP scheduling mechanism by taking the KEP thread-id values of n and m into account:*

$$\begin{aligned} n.prio >_{id} m.prio \;\; &:= \;\; (n.prio \;>\; m.prio) \;\;\vee \\ &\phantom{:=}\;\; (n.prio \;=\; m.prio \;\wedge\; n.threadid.id \;>\; m.threadid.id). \end{aligned}$$

*The instantaneous and delayed dependencies indicate each an instantaneous and delayed dependency constraint respectively:*

$$\begin{aligned} \forall (w,r) \in \mathbb{D}_i \;\; &: \;\; w.prio >_{id} r.prio \\ \forall (w,r) \in \mathbb{D}_d \;\; &: \;\; w.prio >_{id} r.prionext \end{aligned}$$

*Because all dependencies must hold, the writers priority must be greater than the maximum priority of its dependency successors:*

$$\forall n \in N : n.prio >_{id} max_{>_{id}} (\{m.prio \mid m \in n.dep_i\} \cup \{m.prionext \mid m \in n.dep_d\}).$$

Why is the KEP scheduling behavior reflected by the $>_{id}$ relation? First, $>_{id}$ is defined between concurrent nodes as only these can be active concurrently, and second is the greater KEP thread-id value relevant when the priorities are equal, as same as the KEP scheduling works. The $>_{id}$ greater relation can be summarized as 'if not greater in priority than in the thread-id value' relation.

A node might have no dependency successors. This is trivially the case for non-writer instructions, but also for writers that have no appropriate reader. Note that dependencies exists always according to some signal symbol, and both, writer and reader instruction, refer to this signal. If only a writer exists of some signal, no dependencies exist for them and they all have an empty set as dependency successors. The $>_{id}$ -maximum priority of an empty set is defined as the minimum priority, defined as one. The $>_{id}$ relation is *total* (defined on concurrent nodes!), *i. e.*, $n >_{id} m \ \vee \ m >_{id} n$ is true for all concurrent nodes $n, m \in N$, because concurrent nodes must have different KEP thread-id values as defined by the thread-id tree properties in Section 4.2.

The *instantaneous control flow constraints* derive from a node's instantaneous successors:

**Definition 6.7** (Instantaneous Control Flow Constraints). *Within a logical tick, a thread's priority cannot increase:*

$$\forall n \in N \ \forall m \in n.suc_{inst}: \ n.prio \geq m.prio.$$

*An equivalent definition:*

$$\forall n \in N : n.prio \geq max\{m.prio \mid m \in n.suc_{inst}\}.$$

*When the execution starts with a new tick at a delayed instruction, its former delayed successors are in that instant instantaneous and the priorities should also not increase:*

$$\forall n \in N : n.prionext \geq max\{m.prio \mid m \in n.suc_{delayed}\}.$$

This constraint is needed because the increase of a thread priority has only an effect for the possibly next instant, not for the current instant. If a thread is scheduled, than it has the highest priority of active threads, by a yet higher priority it remains the thread with the maximum priority.

Note that for delayed nodes $n$, the definition does not forbid the increase of the *prionext* priority, *i. e.*, $n.prionext > n.prio$ can be assigned without violating the constraints. As mentioned, this is important to schedule valid cyclic programs.

The dependency constraints are between concurrent instructions according to the dependency definition. In contrast, the instantaneous control flow constraints are only among non-concurrent instructions, so it is possible to distinguish these constraints by their KEP thread-ids.

**Theorem 6.8** (Non-Concurrency of Control Successors). *The control flow successors in a CKAG are always among nodes with non-concurrent KEP thread-id:*

$$\forall n \in N, \ m \in n.suc: \ \neg(n.threadid \mid\mid m.threadid).$$

*Proof.* All edges are created during the building process by processing the specific Esterel bodies one by one, especially the bodies of a parallel statement list. Edges between nodes deriving from different concurrent bodies therefore do not exist. □

Implicit concurrent instructions are executed in some order with according control flow successors, but this is done by the KEP scheduling mechanism and not matched by edges in the CKAG. In general concurrent nodes could always be a successor of each other, this behavior is adequatly expressed by the concurrency relation.

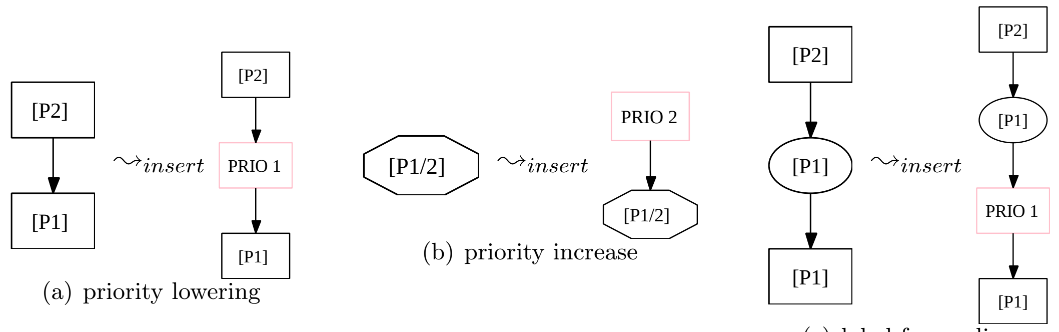
```
procedure main()
  forall  $n \in N$ do
   $n.prio := -1$
  $V_{prio} := \emptyset$
  $V_{prionext} := \emptyset$
  $N_{ToDo} := n_{root}$
  while $\exists n \in N_{ToDo} \setminus V_{prio}$ do
   getPrio($n$)
  forall  $n \in ((D \cup J) \cap V_{prio}) \setminus V_{prionext}$ do
   getPrioNext($n$)
end
```

```
function getPrioNext($n$)
  if  $n.prionext = -1$ then
   if  ($n \in V_{prionext}$) then
     error(Cycle detected!)
   $V_{prionext} \cup= n$
   if  $n \in D$ then
    $n.prionext :=$
        prioMax($n.suc_c \cup n.suc_s$)
   elseif  $n \in J$ then
    $n.prionext :=$
        max($n.fork.prio$,
            prioMax($(n.fork, n).suc_{delayed}$))
   end
  end
  return $n.prionext$
end
```

```
function prio[Next]Max($M$)
  $p := 0$
  forall  $n \in M$ do
   $p :=$ max($p$, getPrio[Next]($n$))
  return $p$
end
```

```
function getPrio($n$)
  if  $n.prio = -1$ then
   if  ($n \in V_{prio}$) then
     error(Cycle detected!)
   $V_{prio} \cup= n$
   if  $n \in D$ then
    $n.prio :=$ prioMax($n.suc_w$)
    $N_{ToDo} \cup= n.suc_c \cup n.suc_s$
   elseif  $n \in F$ then
    $n.prio :=$ prioMax(
        $n.suc_c \cup (n, n.join).suc_{inst}$)
    $N_{ToDo} \cup= (n, n.join).suc_{delayed} \cup$
        $n.join.prio$
   elseif  $n \in T$ then
    $n.prio :=$ max(
      prioMax($n.suc_c$),
      prioMax($n.dep_i$) + 1,
      prioNextMax($n.dep_d$) + 1)
   end
  end
  return $n.prio$
end
```

Figure 6.2: Algorithm to compute the priority assignment.

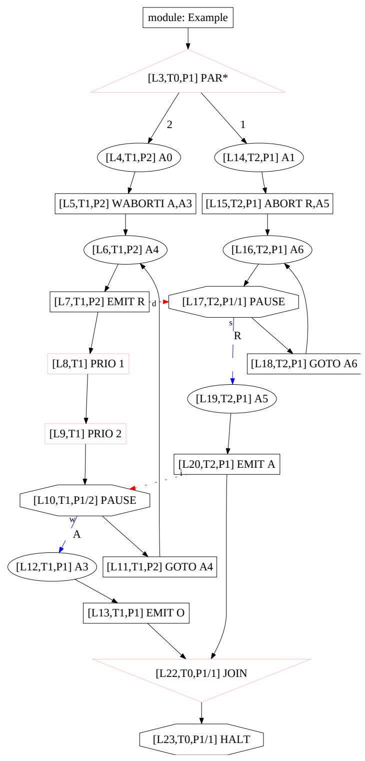


(a) priority lowering (b) priority increase (c) label forwarding

Figure 6.3: Insert PRIO instructions according to the priority assignment.

```
module Example:

input  I;
output O;

signal  A, R in
  [
     weak abort
       sustain  R
     when immediate A;
     emit O
   ||
     await R;
     emit A
  ]
end signal

end module
```

(a) Esterel source



(b) priority assignment

```
%%% Esterel Module: Example

INPUT I
OUTPUT O
SIGNAL A,R
EMIT _TICKLEN,#12

[L01,T0,P1]            PAR 2,A0,1
[L02,T0,P1]            PAR 1,A1,2
[L03,T0,P1]            PARE A2,1
[L04,T1,P2]      A0:
[L05,T1,P2]            WABORTI A,A3
[L06,T1,P2]      A4:
[L07,T1,P2]            EMIT R
[L08,T1]               PRIO 1
[L09,T1]               PRIO 2
[L10,T1,P1/2]          PAUSE
[L11,T1,P2]            GOTO A4
[L12,T1,P1]      A3:
[L13,T1,P1]            EMIT O
[L14,T2,P1]      A1:
[L15,T2,P1]            ABORT R,A5
[L16,T2,P1]      A6:
[L17,T2,P1/1]          PAUSE
[L18,T2,P1]            GOTO A6
[L19,T2,P1]      A5:
[L20,T2,P1]            EMIT A
[L21]            A2:
[L22,T0,P1/1]          JOIN
[L23,T0,P1/1]          HALT
```

(c) resulting KEP assembler program

Figure 6.4: An *priority assignment* example, starting with the Esterel source (a) the assignment is performed on the CKAG and when finished PRIO instructions are inserted (b), see also the resulting KEP program (c). The priorities of the assignment are shown as [P⟨*prio*⟩(/⟨*next*⟩)].

## 6.3 Assignment Algorithm

The fulfillment of all constraints is achieved by the *priority assignment algorithm*. The algorithm computes a minimum assignment, *i. e.*, the assigned priorities are as high as needed to fulfill the constraints, but as low as possible:

$$n.prio(next) := min\{p \in \mathbb{N} : \ \forall m \in n.suc_{constraints} : \ p \geq / >_{id} m.prio\},$$

whereby $n.suc_{constraints}$ summarize all the constraint successors described before. The control flow constraints are fulfilled by assigning at least the maximum successor priority. The priorities must also consider dependency constraints, which can lead to an increase of the priority. In fact dependencies are the only reason why the priority can get higher than the overall minimum. Given the dependency $(w, r)$, the constraint $w.prio >_{id} r.prio$ must be fulfilled. This can be realized by assigning the same priority when the writers thread-id value is greater, *i. e.*, $w.prio = r.prio$. When the thread-id value is lower, the priority is increased by one:

$$\forall r \in w.dep : \ w.threadid.id < r.threadid.id \ \Rightarrow \ w.prio := r.prio + 1.$$

Priority $r.prio + 1$ is in case of a lower thread-id value the lowest priority with $w.prio >_{id} r.prio$.

If a node has no constraints at all, neither control flow nor dependency, the overall minimum priority is assigned as priority. These nodes are typically *delay nodes*, where the instantaneous control flow stops.

The priority assignment algorithm is shown in Figure 6.2. It starts in the main() procedure, which, after some initializations, in line 8 calls getPrio() for all nodes that must yet be processed. This set of nodes, given by $N_{ToDo} \setminus V_{prio}$ (for "Visited"), initially just contains the root of the CKAG. After *prio* has been computed for all reachable nodes in the CKAG, a forall loop computes the *prionext* priorities for reachable delay/*join nodes* that have not been computed yet.

Procedure getPrio() first checks whether it has already computed *n.prio*. If not, it then checks for a recursive call to itself (lines 3/4, see also Theorem 6.9). The remainder of getPrio() computes *n.prio* and, in case of delay and fork nodes, adds nodes to the $N_{ToDo}$ list. Similarly getPrioNext() computes *n.prionext*.

**Theorem 6.9** (Termination). *For a valid, acyclic Esterel program, getPrio() and getPrioNext() terminate. Furthermore, they do not generate a "Cycle detected!" error message.*

*Proof.* Procedure getPrio() produces an error (line 4) if it has not computed *n.prio* yet (checked in line 2) but has already been called (line 3) earlier in the call chain. This means that it has called itself via one of the calls to prioMax() or prioNextMax() (via getPrioNext()). An inspection of the calling pattern yields that an acyclic program in the sense of Definition 6.5 cannot yield a cycle in the recursive call chain. □

**Theorem 6.10** (Fulfillment of Constraints). *For a valid, acyclic Esterel program, the priority assignment algorithm computes an assignment that fulfills the constraints of Definitions 6.6 and 6.7.*

*Proof.* First observe that, apart from the initialization in main(), each $n.prio$ is assigned only once. Hence, when prioMax() returns the maximum of priorities for a given set of nodes, these priorities do not change any more. Therefore, the fulfillment of Constraint 6.6 can be deduced directly from getPrio. □

**Theorem 6.11** (Linearity). *For a CKAG with nodes $N$ and edges $E$ and $P$, the computational complexity of the priority assignment algorithm is $\mathcal{O}(|N| + |E| + |P|)$.*

*Proof.* Apart from the initialization phase, which has cost $\mathcal{O}(|N|)$, the cost of the algorithm is dominated by the recursive calls to getPrio(). The total number of calls is bounded by $|E| + |P|$. With an amortization argument, where the costs of each call are attributed to the callee, it is easy to see that the overall cost of the calls is $\mathcal{O}(|E| + |P|)$. □

Note also that, while the size of the CKAG may be quadratic in the size of the corresponding Esterel program in the worst case, it is in practice (for a bounded abort nesting depth) linear in the size of the program. This results in an algorithm that is effectively complexity linear in the program size.

## 6.4 Realizing the Priority Assignment

After priorities have been computed for each reachable node in the CKAG, we must generate code that ensures that each thread is executed with the computed priority. This task is presented in Figure 6.3 by the *priority insertion* algorithm. First the default priorities of the PAR instructions are replaced by the *fork node* child priorities to initialize the threads according to the assignment.

Each time the priority changes in the assignment, *i. e.*, a node $n \in N$ has a successor $m \in n.suc$ with a different assigned priority ($n.prio \neq m.prio$), PRIO instructions are inserted accordingly. In fact $n.prio > m.prio$ must hold true and is simply realized by inserting a PRIO instruction with the lower priority of $m.prio$ in between. When the priorities $n.prio$ and $n.prionext$ of a delayed node $n \in N$ differ, a PRIO instruction with an increased priority of $n.prionext$ is inserted. This time the priority can grow, *i. e.*, $n.prio < n.prionext$, and in that case the PRIO $n.prionext$ has to be inserted before $n$, see Figure 6.3 (b). As mentioned earlier, the increasing the thread priority after the delay would have no effect on the KEP scheduling, at least in current instant. This would violate the assignment.

If the priority $d.prio$ of a *delay node* $d$ is lower than one of its parent's priority, but the *prionext* is again increasing, then two PRIO instructions are inserted, see the example in Figure 6.4. At program line L8 two PRIO instructions are inserted. First the PRIO $d.prio$ instruction is inserted, followed by the PRIO $d.prionext$. A reversed order would make the higher assigned priority $d.prionext$ obsolete, because directly following the lower *d.prio* would be set. Hence, first the changes of control flow successors in the assignment are realized, see the algorithm graphically described in Figure 6.3, followed the realization of higher assigned *prionext* priorities. Note that the realization of a lower *prionext* value is still done behind the delayed node.

The compiler suppresses PRIO instructions for the main thread, because the main thread never runs concurrently to other threads. Also no PRIO instructions are inserted at the end of a thread, although the priority assignment indicates a successor with a lower priority, as the PRIO instruction would have no effect.

The insertion of PRIO instructions must respect the behavior of the instructions to hold main invariants of the CKAG. *E.g.*, consider the insertion of a PRIO instruction in the middle of a GOTO and its label. This has no effect on the assembler side, and it also destroys the CKAG control flow behavior and therefore the CKAG does not longer match the control flow of the corresponding KEP assembler. The insertion of PRIO instructions before abort labels does not realize the priority assignment as well. The solution to such problems is to forward the insertion to the label's successor, see Figure 6.3 (c). This is always possible, because label nodes have never more than one child node. This process is recursive, *i. e.*, if the successor of a label is again a label, this label is also forwarded and so on.

The insertion of a PRIO instruction can always be forwarded over nodes whose instruction is no reader. In particular, it can be forwarded across writer nodes, because it is never a problem to execute a writer with a higher priority than the assignment has computed. To avoid the need for multiple PRIO instructions, it should not be forwarded to multiple children, but only to a single child. As mentioned before, *label nodes* have this property and are additionally never readers. Another instruction with the same behavior are the NOTHING instructions that might be needed to distinguish labels from each other. As an optimization the forwarding is applied to NOTHING instructions. If the control flow of the NOTHING stops in the following at a JOIN, the PRIO instruction is not needed at all. This case is characteristic, because the NOTHING often distinguishes PARE labels from other labels, *e. g.*, preemption labels of abort and trap scopes. Other more advanced optimizations are explained in the next chapter.

# 7 Compiler Optimizations

This chapter describes the compiler optimizations of the strl2kasm compiler. Optimizations in general are performed to optimize the use of machine resources, like program space or register count, and program behavior, like execution time and response time. *E.g.*, the later described CKAG *collapsing* optimizes both, space and time, because it combines several instructions to less, but semantically equivalent, instructions that are executed in less cycles.

Other optimization techniques target the limits of the KEP assembler syntax: the KEP instructions have a limited size and so are also all instruction parameters limited, especially the thread priorities and KEP thread-id values. So it is a goal to minimize the maximum priority and thread-id value of a given program, see the following Sections 7.4 and 7.5.3 respectively.

Yet another optimization is applied on algebraic expressions to reduce the use of instructions and registers via *propagation propagation* [2] during the CKAG building as already mentioned. A *peephole optimization* [1] suppresses directly the creation of needless instructions during the building process, *e. g.*, instruction LOAD X,X assigning a register value it already has.

Some of these techniques belong to specific compilation steps and have to be performed according to these steps, *e. g.*, the minimization of signal dependencies makes only sense before the priority assignment is performed, and the instruction collapsing must be performed after the priority assignment.

The dead code removal, which is described in the next section, can be made during all compilation steps. However it should be performed first, so that also the other optimizations can benefit from it.

## 7.1 Dead Code Removal

Dead or unreachable code is program code which will never be executed, and therefore can be removed without changing the semantic of the program. In general it is hard to determine whether an instruction is reachable or not. The dead code removal optimization can result in smaller CKAG and KEP assembler program.

Syntactically and semantically it is allowed to have such code within a program. However, it often indicates a bug, because writing unreachable code is normally not intended during software development, at most temporarily for debugging purposes. Especially if a program is used in safety critical applications and therefore has to be bug-free as possible, the compiler should throw a warning when detecting dead code. In this sense the dead code removal is not only an optimization technique, it is also a method of detecting program errors on the Esterel side.

On the Esterel level, there exists very easily detectable dead code, like statements following a loop or an exit statement. Such code is automatically ignored by the CKAG building process by setting the graph builder to the so called *sleep* state, see Chapter 5. The same way statements are ignored that follow a concurrent statement list, and (in KEP) after a JOIN instruction. If one of the sub-thread bodies contains a loop, it never terminates by non-preemptive control flow. Nevertheless the control might go further through the preemption of ABORT and EXIT instructions at their appropriate labels.

The above dead code removal is performed on the Esterel level during the generation of the CKAG. A dead code analysis can be performed additionally on the CKAG by searching for non-root nodes that have no parents. Such a node is apparently not reachable and can be removed together with its children if they become orphans and so on. The CKAG is created as a connected graph with only one root node, namely the interface, so the described method would not change the CKAG, because no node will be removed. There some analysis must be conducted first, which, *e. g.*, removes preemption edges of signals that never occur for some reason. The preemption edges of all output, local and trap signals that have no according writer are unreachable. This problem is in general more difficult, especially if input signals are involved, which need no writer instruction to be active. Such an approach could result in orphan nodes making the search and remove for such nodes useful, because the CKAG is built coherently at the beginning.

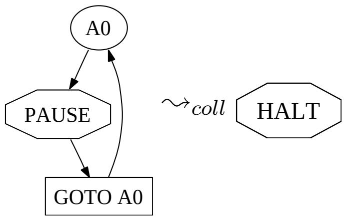
## 7.2 KEP Collapsing

This section describes how to replace specific KEP instructions by semantically equivalent but more efficient instructions. This technique can be seen as the opposite of the Esterel dismantling described in Chapter 3. Especially the three types of Esterel statements halt, sustain and await, that are dismantled initially, as described in Section 3.3, can now possibly be collapsed back to the KEP instructions HALT, SUSTAIN and AWAIT[I].

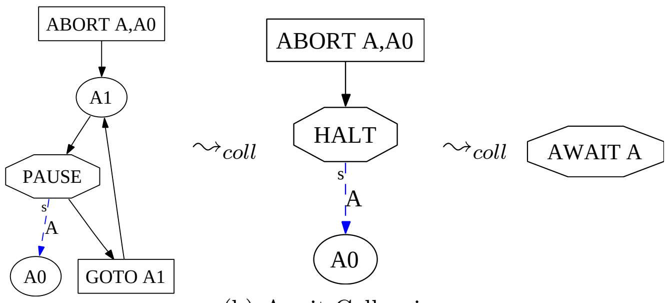
Figure 7.1 describes which pattern has to be found and by which instruction they were replaced. This optimization technique must be performed after the priority assignment, which is described in the previous Chapter 6, to ensure the schedulability. If it is not necessary to insert priority instructions, the pattern will be left unchanged and can be collapsed.

The HALT pattern consists of a PAUSE node surrounded by a GOTO loop. This is the simplest pattern. The SUSTAIN pattern contains an additional EMIT within the loop. If the EMIT is valued, the resulting SUSTAIN is too with the same value. More complicated valued EMIT's which need more instructions to perform a register computation do not match the SUSTAIN pattern and will not be collapsed.

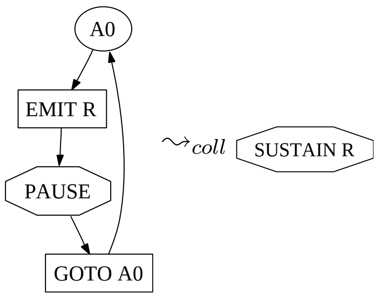
The AWAIT pattern consists of a HALT instruction which is the body of an ABORT instruction. To find these kind of pattern the HALT collapsing has to be performed first. This pattern has an immediate variant: the AWAITI pattern. The according



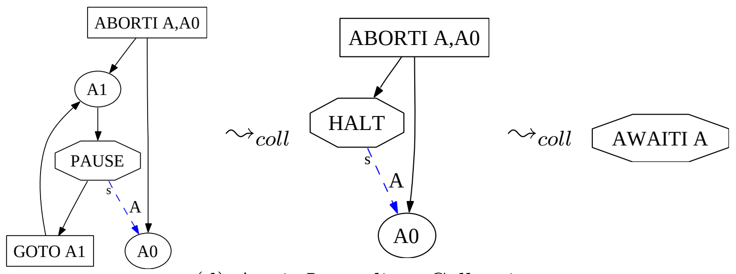
(a) Halt Collapsing



(b) Await Collapsing



(c) Sustain Collapsing



(d) Await Immediate Collapsing

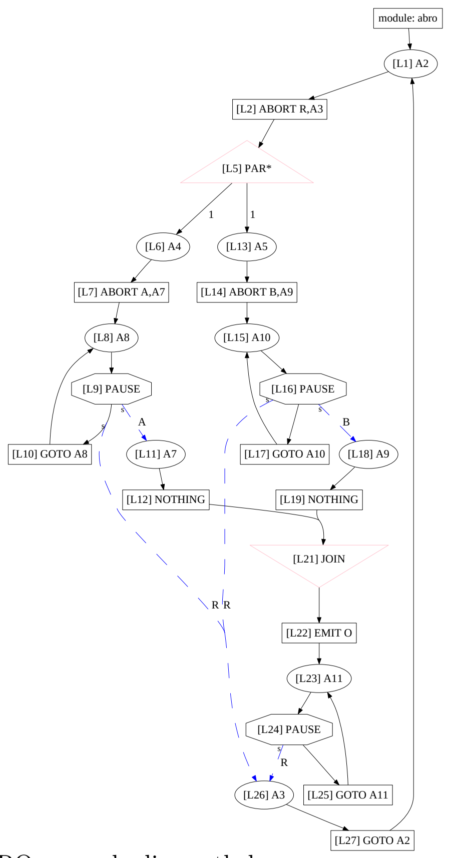
Figure 7.1: CKAG Collapsing (Pattern, Replacement)

ABORTI instruction is immediate and so this pattern has an additional edge between the ABORTI and its abort label.

The collapsing of the ABRO example is shown in Figure 7.2. The lines of code were reduced from 27 to 15 and the WCRT from 13 to 11. First three HALT patterns are found, of these two can be reduced to an AWAIT. During the erasing of the address labels A7 and A9 of the AWAIT pattern, the following NOTHING instructions are also erased. The NOTHING instructions separated each an abort address label from a PAR body label and are now superfluous, their erasing reduces the code size in addition.

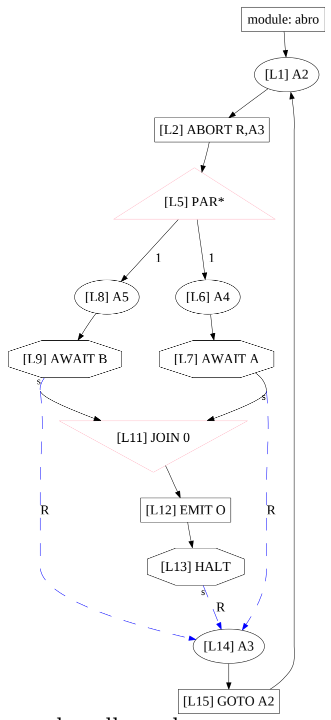
A possible further work would be the detection of more complex patterns. The KEP supports delayed AWAIT instructions in its syntax by setting a delay $n$ via the instruction LOAD _COUNT,#$n$ in front of the AWAIT. Currently such instructions are dismantled as early as on the Esterel level, which is necessary due to priority assignment reasons 3.3, but are not collapsed back, even it would be theoretically possible. The same situation with the await case, which is supported in KEP by the CAWAIT instructions. These patterns are fare more complex than the implemented ones, but should be implemented in the future.

```
INPUT A,B,R
OUTPUT O
EMIT _TICKLEN,#13

[L01] A2:
[L02]       ABORT R,A3
[L03]       PAR 1,A4,1
[L04]       PAR 1,A5,2
[L05]       PARE A6,1
[L06] A4:
[L07]       ABORT A,A7
[L08] A8:
[L09]       PAUSE
[L10]       GOTO A8
[L11] A7:
[L12]       NOTHING
[L13] A5:
[L14]       ABORT B,A9
[L15] A10:
[L16]       PAUSE
[L17]       GOTO A10
[L18] A9:
[L19]       NOTHING
[L20] A6:
[L21]       JOIN
[L22]       EMIT O
[L23] A11:
[L24]       PAUSE
[L25]       GOTO A11
[L26] A3:
[L27]       GOTO A2
```



(a) The ABRO example dismantled

```
INPUT A,B,R
OUTPUT O
EMIT _TICKLEN,#11

[L01] A2:
[L02]       ABORT R,A3
[L03]       PAR 1,A4,1
[L04]       PAR 1,A5,2
[L05]       PARE A6,1
[L06] A4:
[L07]       AWAIT A
[L08] A5:
[L09]       AWAIT B
[L10] A6:
[L11]       JOIN 0
[L12]       EMIT O
[L13]       HALT
[L14] A3:
[L15]       GOTO A2
```



(b) The ABRO example collapsed

Figure 7.2: The abro example before and after the collapsing: the code size is significantly reduced.

## 7.3 Priority Assignment Modification

This section explains how to minimize the use of PRIO instructions by modifying a given priority assignment. The priority assignment described in Chapter 6 computes priorities of minimal count for each node of a CKAG fulfilling all dependency and control flow constraints defined in Chapter 6.

The basic idea of this optimization is to heighten priorities without violating the constraints to avoid declining priorities in the control flow, because each of these priority differences induces the insertion of a PRIO instruction. Considering the example in Figure 7.3 (b), the EMIT S has a priority of two and the PAUSE of one assigned. By heightening the PAUSE's priority to two, no constraint would be damaged and the insertion of the instruction PRIO 1 is now unnecessary.

Note that this optimization approach does not increase the maximum used priority of a program, although the priority *n.prio* of a node might grow. The reason is that a priority is always heightened to an existing priority *n′.prio*, which already assigned, so the overall priority maximum is not affected.

A heuristic is implemented, called *dominant writer heuristic*, that heightens the priorities of all nodes of a thread to the maximum priority of this thread. This is done when a thread and its sub-threads contain of no reader nodes. Thus all constraints remain fulfilled and at the same time no PRIO instructions are needed to realize the assignment.

## 7.4 Minimizing Dependency Count

The signal dependencies are conservative, *i. e.*, some of them might be superfluous and can be removed, but this is in general hard to compute. Given a signal dependency $(w, r)_s$ with writer $w$ and reader $r$ of signal $s$, this dependency would be superfluous, if $w$ and $r$ are never executed within the same instant.

This optimization results in a greater amount of Esterel programs that can be correctly compiled to KEP, as potentially wrong priority cycles are avoided. We call these *pseudo priority cycles*, caused by one or more *pseudo dependencies*. A pseudo dependency is a dependency between instructions that cannot be instantaneously active and therefore does not have to be scheduled. The computation of signal dependencies is made conservatively, *i. e.*, the amount of computed dependencies could be greater than it would be, if computed exactly with a NP-complete algorithm. Such an algorithm could identify a pseudo dependency, based of a program simulation technique to test the program for all possible states and transitions. Nevertheless if a program is cycle-free, as each constructive program can be translated to [30], no pseudo cycles will occur.

Another aspect of this optimization is the use of PRIO instructions. They will be reduced, since each dependency could cause an increase in priority differences of nodes/instructions during the priority assignment, which need PRIO instructions to be realized.

A trivial knock-out criterion for a dependency belongs to its signal $s$: if it is never emitted, then the according dependency can be removed. For signals of kind input, this behavior cannot be determined, but for local and trap signals this is easily possible and currently implemented in the strl2kasm.

A more advanced reduction could result from a dead code analysis plus removal. As before described, if the unreachable code contains writer and reader, their dependencies are also removed. The removal of one writer might reduce an arbitrarily high number $n$ of dependencies, if the specific writer has $n$ readers. So the dead code analysis can have a significant impact on the amount of signal dependencies.

## 7.5 Thread-Id Value Assignment

This section describes a variant of the KEP thread-id value assignment as presented in Section 4.2 that takes additionally the CKAG signal dependencies, explained in Chapter 6, into account. This optimization technique simplifies the priority assignment by solving the induced constraints before the priority assignment has even started.

The method described in the previous section removes signal dependencies entirely, if possible. Such a dependency causes no dependency constraint, simply because it does not exist anymore. So both techniques have the same advantages for the priority assignment, but neither technique makes the other obsolete, both are needed and are complementary. A dependency constraint could eventually be solved by the id value assignment that might not be removable by the previous method. The other way round, if the id value optimization has no success, the removing of superfluous dependencies might after all lead to a constraint fulfillment. Apparently the previous method will be performed at first, because the id value assignment has no impact, whether a signal dependency is superfluous or not.

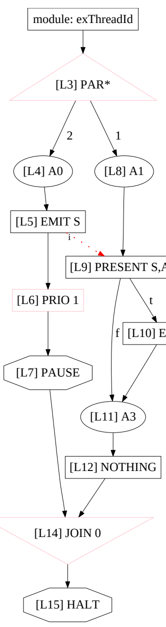
Figure 7.3 illustrates the KEP thread-id value optimization by an example. The example program contains a dependency belonging to signal $S$ between the emit S and the present S statements. This signal dependency results in the dependency constraint to execute the EMIT S before the PRESENT S. This is realized by the priority assignment that assigns writer EMIT S the priority two, reader PRESENT S gets a priority of one. The two sub-threads of the main thread are initialized with these priorities at creation, see their priorities in Figure 7.3 (b) at the *fork node*'s child edges. Within the first thread, particularly the writer thread, the priority has to be lowered according to the assignment. A PRIO 1 instruction is inserted at the PAUSE in line $L06$ behind the EMIT S, lowering the thread priority from two to one.

The first scheduling criterion among active threads is a thread's priority. If there are multiple threads with the same priority the KEP thread-id values of the threads are considered. A consequence of this behavior is that constraints could be solved by assigning them the same priorities, with a higher id value of the writer thread. On the other hand, as in this example, if the writer thread has a lower thread-id value than its reader thread, its priority has to be greater. The KEP thread-id values must

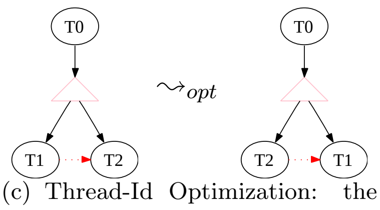
```
module exThreadId:

input A,B;
output S,T;

[
   emit S;
   pause
  ||
   present S then
     emit T
   end present
]

end module
```

(a) Esterel source

```
INPUT A,B
OUTPUT S,T
EMIT _TICKLEN,#10

[L01,T0,P2]         PAR 2,A0,1
[L02,T0,P2]         PAR 1,A1,2
[L03,T0,P2/2]       PARE A2,2
[L04,T1,P2]    A0:
[L05,T1,P2]         EMIT S
[L06,T1]            PRIO 1
[L07,T1,P1/1]       PAUSE
[L08,T2,P1]    A1:
[L09,T2,P1]         PRESENT S,A3
[L10,T2,P1]         EMIT T
[L11,T2,P1]    A3:
[L12,T2,P1]        NOTHING
[L13]          A2:
[L14,T0,P2]         JOIN 0
[L15,T0,P1/1]       HALT
```

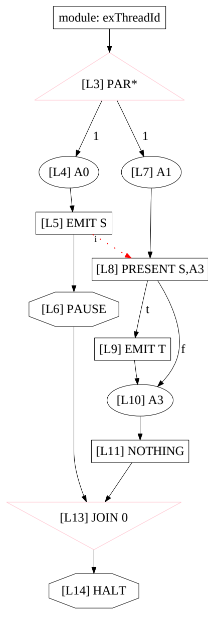


(b) unoptimized KEP and CKAG



(c) Thread-Id Optimization: the writer thread gets the higher thread-id value

```
INPUT A,B
OUTPUT S,T
EMIT _TICKLEN,#9

[L01,T0,P1]         PAR 1,A0,2
[L02,T0,P1]         PAR 1,A1,1
[L03,T0,P1/1]       PARE A2,1
[L04,T2,P1]    A0:
[L05,T2,P1]         EMIT S
[L06,T2,P1/1]       PAUSE
[L07,T1,P1]    A1:
[L08,T1,P1]         PRESENT S,A3
[L09,T1,P1]         EMIT T
[L10,T1,P1]    A3:
[L11,T1,P1]         NOTHING
[L12]          A2:
[L13,T0,P1]         JOIN 0
[L14,T0,P1/1]       HALT
```



(d) KEP and CKAG with optimized KEP thread-id values

Figure 7.3: The *thread-id optimization* applied on the Esterel program shown in (a). In (b) the writer thread has a lower thread-id value, so it must have a greater priority in the assignment, which results here in the use of a PRIO instruction. After the optimization in (c) the program could be scheduled with priority one for all threads, this saves the PRIO instruction, reduces the WCRT and the maximum priority, see (d).

be defined according to a DFS principle as explained in Section 4.2. *I.e.*, the sub-threads must have greater id values than their parents, but among the children are no limitations. The default is that the id values are assigned to their position as given from the Esterel source: the ids are assigned with an increasing number resulting in lower thread-id values for the anterior threads, namely the parent threads. In the example the writer thread is the first thread in parallel statement and therefore gets the lower id value of one, denoted with $T1$, than the reader thread with value $T2$. This is an arbitrary choice, since the Esterel program would not change semantically if the thread bodies of the concurrent statement are switched.

So we have the freedom to choose another order to assign KEP thread-id values to sub-threads that take signal dependencies into account, before the priority assignment starts. The optimization should assign lower thread-id values to reader threads. This respects the KEP scheduling principle: a higher thread-id takes preference under equal thread priorities. In the example we have a strict division between writer and reader threads and the optimization, as seen in Figure 7.3 (c) takes place by visiting first the reader thread. This results in a switch of the sub-thread's id values. The main thread remains unchanged with the thread-id value of zero ($T0$) and its sub-threads keep greater id values.

The optimization of the example leads to a priority assignment of priority one for all instructions, the former PRIO 1 is not needed anymore. The maximum priority is reduced from two to one and also the program size and WCRT are decreased by one. The lowering of the maximum priority is relevant for the bit size of the KEP instruction set architecture, as priorities occur, *e. g.*, in PAR and PRIO instructions. Therefore the thread-id optimization has an impact on several performance critical program properties.

Note that the example is a minimal example to show the core problem and its solution. But in fact the PRIO 1 instruction is not really needed here and does not change the program semantics. This problem could be addressed by an optimization technique employed on the priority assignment itself to avoid the PRIO 1 insertion, see Section 7.3. Nevertheless the priority lowering and therefore the PRIO insertions are needed in general: the PAUSE could occur within the scope of an ABORT signal and must have the lower priority as being a reader to that signal.

The reduction of PRIO instructions lowers the program size at least by its amount, but there might be an additional effect by the previous described collapsing. The use of less PRIO instructions could result in more patterns found by the collapsing algorithm, which means even less instructions. The Esterel statements halt, sustain and await are dismantled to allow the insertion of PRIO instructions in between, see Section 3.3, and if no PRIO instructions are needed, they are collapsed again. Therefore the id value optimization increase typically the number of patterns found, and each extra pattern reduces the program size by two instructions.

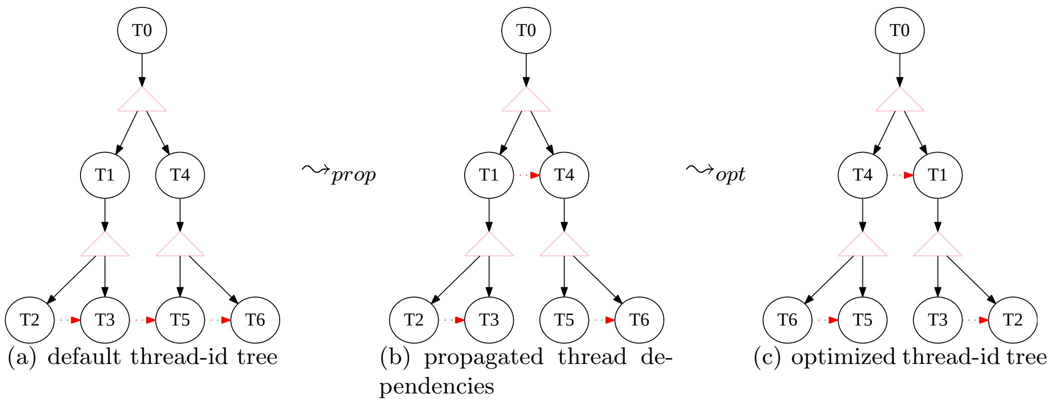
Next the KEP thread-id value optimization is described in more detail. First are the *thread dependencies* defined, which reduce the signal dependencies to the KEP thread-id level, followed by the *propagated thread dependencies*, since KEP thread-id values are assigned according to the sub-thread relation via DFS.

```
procedure main (tree $\mathbb{T}$)
  compute_thread_dependencies($\mathbb{T}$);
  compute_propagated_thread_dependencies($\mathbb{T}$);

  $id\_value\_counter = 0$;
  new_id($\mathbb{T}.main$);
end
```

```
procedure new_id (thread−id $t$)
  $t.id = id\_value\_counter++$;// new id value

   forall  $f \in t.forks$ do
    // ordered assignment
    new_ids($f.subthreads$, $true$);
  end
end
```

```
procedure new_ids($T$, boolean $ordered$)
  if  ($|T| > 0$) then
    if  ($ordered$) then
      if  ($T_{\text{middle}} = T$) then
        Choose $e \in T_{\text{middle}}$
        new_id($e$);
        new_ids($T_{\text{middle}} \setminus \{e\}$,$true$);// ordered
      else
        new_ids($T_{\text{sink}}$,$false$);// not ordered
        new_ids($T_{\text{middle}}$,$true$);// ordered
        new_ids($T_{\text{source}}$,$false$);// not ordered
      fi
    else
      forall  $t \in T$ do
        new_id($t$);
      end
    fi
  fi
end
```

Figure 7.4: Optimized Thread-Id Value Assignment: before the thread-id value optimization takes place, first the thread dependencies are computed, followed by its propagated thread dependencies. After that the DFS is started at the main thread by *id_optimize* with a start value of zero.



(a) default thread-id tree (b) propagated thread dependencies (c) optimized thread-id tree

Figure 7.5: First are the thread-dependencies of the default tree computed (a), then its propagated thread dependencies (b). In the last step (c) is a new value assignment accordingly calculated.

**Definition 7.1** (Thread Dependencies). *The KEP thread-id tuple set $\mathbb{D}_T \subseteq \mathbb{T} \times \mathbb{T}$ is defined as the thread dependencies of the signal dependencies $\mathbb{D} \subseteq N \times N$. Set $\mathbb{D}_T$ corresponds to $\mathbb{D}$, but without their information about CKAG nodes and KEP instructions, just the KEP thread-ids are left:*

$$\mathbb{D}_T := \mathbb{D}.threadid := \{(d_w.threadid, d_r.threadid) \in \mathbb{T} \times \mathbb{T} \mid (d_w, d_r) \in \mathbb{D}\}.$$

The thread dependency relation $\mathbb{D}_T$ can be seen as the signal dependency relation $\mathbb{D}$ modulo nodes and instructions.

Given a thread dependency $(t_w, t_r) \in \mathbb{D}_T$, we call $t_w$ *writer thread* and $t_r$ *reader thread*. Writer and reader thread are always concurrent per definition of the signal dependencies, were they are derived from.

The thread-id assignment has to be made according to the DFS principle as mentioned in Section 4.2 to ensure higher thread-id values for sub-threads. Due to this KEP thread-id tree invariant, the dependencies were 'propagated' to the highest possible level, according to the sub-thread relation, where the threads are still concurrent:

**Definition 7.2** (Propagated Thread Dependency). *Given a thread dependency $t = (t_w, t_r) \in \mathbb{D}_T$, the according* propagated thread dependency *is defined as the tuple $t_p = (p_w, p_r) \in \mathbb{T} \times \mathbb{T}$ of concurrent thread-ids $p_w$ and $p_r$, that are maximal with the sub-thread relation in this behavior:*

$$t_p = \ max\ \{(p_1, p_2) \in \mathbb{T} \times \mathbb{T} \mid t_w \leq p_1 \ \wedge \ t_r \leq p_2 \ \wedge \ p_1 \mid\mid p_2\}.$$

*Note that $t$ and $t_p$ might be the same, this is exactly the case when their thread-ids are the children of their least common fork.*

*The set of propagated thread dependencies $\mathbb{D}_P$ is defined by:*

$$\mathbb{D}_P := \{t_p \in \mathbb{T} \times \mathbb{T} : t \in \mathbb{D}_T\}.$$

*According to the definition of a propagated thread dependency $(p_w, p_r)$ are $p_w$ and $p_r$ children of the same fork, in particular their least common fork, so is $\mathbb{D}_P$ partitioned by* fork nodes*. Given* fork node *$f$, the subset/sub-relation $\mathbb{D}_P(f)$ is defined as:*

$$\mathbb{D}_P(f) := \{(p_w, p_r) \in \mathbb{D}_P : p_w, p_r \in f.subthreads\}.$$

*The partition is described by the distinct union of all these subsets:*

$$\mathbb{D}_P = \dot{\bigcup}_{f \in ForkNodes} \mathbb{D}_P(f).$$

Next the optimization algorithm is explained in detail, taking the propagated thread dependencies into account.

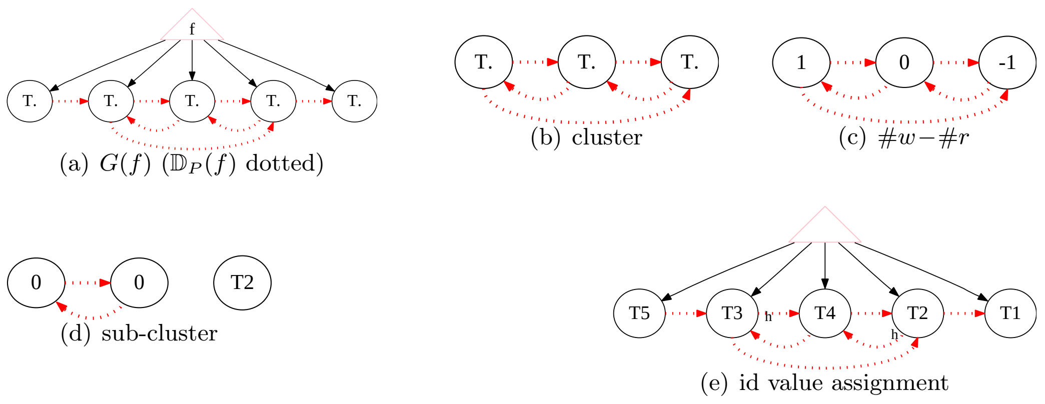


Figure 7.6: Thread-Id Value Assignment of a Cluster: The first and the last thread are $T_{source}$ and $T_{sink}$ and get (here) in the assognment values 5 and 1/highest and lowest, respectively (a). Hereafter, the rest $T_{middle}$ cannot be split further into *source* and *sink* parts (b). The $\#w - \#r$ heuristic is applied, the minimum is $-1$ and the thread gets value 2. The rest is again a cluster, both 0 in $\#w - \#r$ (d), and get values 3 and 4, the result can be seen in (d). The minimum of two hard propagated thread dependencies has been reached, denoted by $h$.

## 7.5.1 Thread-Id Value Assignment Algorithm

The algorithm described in this section takes a KEP thread-id tree $\mathbb{T}$ and computes according to its propagated dependencies $\mathbb{D}_P$ a new KEP thread-id tree $\mathbb{T}'$. The KEP thread-id tree structure remains unchanged, only the id values are modified.

The id values of a KEP thread-id tree can be seen as a function/assignment from KEP thread-ids to integer:

**Definition 7.3** (Thread-Id Assignment). *Given a KEP thread-id tree $\mathbb{T}$, the function $id_{\mathbb{T}} : \mathbb{T} \rightarrow \mathbb{N}_0, t \rightarrow t.id_{\mathbb{T}}$ induced by the thread-id values is called thread-id assignment.*

The $id_{\mathbb{T}'}$ assignment should minimize the number of propagated thread dependencies that cannot be scheduled by a constant priority assignment, *i. e.*, the respective writer thread-id has a lower thread-id value than its reader.

**Definition 7.4** (Hard Propagated Thread Dependency). *Given a propagated thread dependency $p = (p_w, p_r) \in \mathbb{D}_P$, $p$ is called hard in $\mathbb{T}$, if the writer thread has a lower id value than its reader in $\mathbb{T}$ respective by the assignment $id_{\mathbb{T}}$:*

$$p \textit{ hard in } \mathbb{T} \;:=\; p_w.id_{\mathbb{T}} < p_r.id_{\mathbb{T}}.$$

*The set of hard propagated thread dependencies of $\mathbb{D}_P$ is defined by:*

$$\mathbb{D}_P^{h,\mathbb{T}} := \{p \in \mathbb{D}_P : p \textit{ hard in } \mathbb{T}\}$$

So the algorithm should compute a thread-id tree $\mathbb{T}'$, that is minimal with respect to the number of hard propagated thread dependencies:

$$\forall \text{ thread-id assignment } id_{\mathbb{T}} : \; |\mathbb{D}_P^{h,\mathbb{T}'}| \;\leq\; |\mathbb{D}_P^{h,\mathbb{T}}|.$$

The best case scenario is the fulfillment of all propagated thread dependencies, *i. e.*, $|\mathbb{D}_P^{h,\mathbb{T}'}| = 0$. In this case the fulfillment of all signal dependencies by a constant schedule with priority one for all KEP threads is possible. The scheduling would be realized by thread-id values only and no PRIO instructions are needed. If there exists a cycle of propagated thread dependencies, their dependencies still have to be scheduled with a non-constant priority assignment. More generally, each *Strongly Connected Component* (SCC) of at least two KEP thread-ids must be scheduled non trivially, we call them *dependency cluster*.

**Definition 7.5** (Dependency Cluster)**.** *Given the graph $G = (\mathbb{T}, \mathbb{D}_P)$ with KEP thread-ids $\mathbb{T}$ as nodes and their propagated thread dependencies $\mathbb{D}_P$ as edges. A SCC $c \subseteq \mathbb{T}$ with $|c| > 1$ is called a dependency cluster of $\mathbb{T}$.*

*The partition of $\mathbb{D}_P$ is inherited to $G = \dot{\bigcup}_{f \in ForkNodes} G(f)$, whereby, given fork node $f$, the sub-graph according to $f$ is defined as $G(f) := (f.subthreads, \mathbb{D}_P(f))$.*

*All SCCs, especially the clusters, are part of exactly one sub-graph $G(f)$ of $G$.*

**Theorem 7.6** (Cluster Assignment)**.** *A dependency cluster $c$ cannot be scheduled with a constant priority assignment.*

*Proof.* Given a dependency cluster $c$ with a constant priority assignment. We show that at least one dependency constraint is not fulfilled. By definition of a SCC and $|c| > 1$, we find in $c$ a cycle $(t_1, \ldots, t_n, t_1)$ with $n > 1$ of propagated thread dependencies $(t_i, t_ti + 1)$. So there exists a thread dependency $(t_w, t_r)$ with $t_w.id < t_r.id$, because $t_i.id > t_{i+1}.id$ cannot be true for all $i = 1, \ldots, n$. Per definition of the propagated thread dependencies there exists a signal dependency $(d_w, d_r)$ with $d_w.threadid \leq t_w$ and $d_r.threadid \leq t_r$. It follows $d_w.threadid.id < d_r.threadid.id$, because of the KEP thread-id tree characteristics. □

The basic idea of the algorithm is still a DFS, as in the default value assignment, to ensure the KEP thread-id tree properties. The KEP thread-id value assignment starts with value zero at the main thread and increases this count for each new KEP thread-id the DFS visits. The only degree of freedom is the order of how sub-threads are visited at any given *fork node f*. Here the optimization takes place: the propagated thread dependencies are ordered from reader to writer or sink to source respectively, and are visited accordingly . The set of KEP thread-ids that have to be ordered is $\mathbb{D}_P(f)$ and we use the according sub-graph $G(f)$ to order. We identify sink and source sets, the sinks get the lowest id values and the source get the highest. These two types of sets are defined in the following.

**Definition 7.7** (Thread Dependency Sink and Source (Set))**.** *Given KEP thread-ids $T \subseteq \mathbb{T}$. The set $T_{sink} \subseteq T$ is called thread dependency sink of $T$, when it contains* *the KEP thread-ids, that are no writers of a KEP thread-id in $T$:*

$$T_{sink} := \{t \in T \mid \forall t' \in T : (t,t') \notin \mathbb{D}_P\}$$

*The set $T_{source} \subseteq T$ is called thread dependency source of $T$, if it consists of all writers of $T$, that are itself no reader of $T$:*

$$T_{source} := \{t \in T \mid \exists t' \in T : (t,t') \in \mathbb{D}_P \ \wedge \ \forall t' \in T : (t',t) \notin \mathbb{D}_P\}$$

*Per definition these two sets are disjoint:*

$$T_{sink} \cap T_{source} = \emptyset$$

*The set $T_{middle} := T \setminus (T_{sink} \dot\cup T_{source}) \subseteq T$ is called the dependency middle of $T$. The dependency middle accomplishes $T_{sink}$ and $T_{source}$ via distinct union to $T$ and form therefore a partition of $T$:*

$$T_{sink} \dot\cup T_{middle} \dot\cup T_{source} \ = \ T$$

Given sub-graph $G_f$ to sort, its sink and source thread dependency sets $G(f)_{sink}$ and $G(f)_{source}$ are computed, then the KEP thread-ids of $G(f)_{sink}$ get the lowest and $G(f)_{source}$ the highest id values. If the union of $G(f)_{sink}$ and $G(f)_{source}$ equals not $G(f)$, then this principle is recursively applied on the dependency middle $G(f)$ $(G(f)_{sink} \cup G(f)_{source})$ of $G(f)$ and so on. If the middle remains unchanged, a dependency cluster is found

**Theorem 7.8** (Middle Cluster). *Given KEP thread-ids $\emptyset \neq T \subseteq G(f)$, if dependency sink $T_{sink}$ and source $T_{source}$ sets are empty, then $T = T_{middle}$ contains at least one cluster:*

$$\forall\, T \subseteq G(f) : \ T_{sink} \dot\cup T_{source} \ = \emptyset \ \ \Rightarrow \ \ \exists cluster\ c : \ c \subseteq T$$

*Proof.* The propagated dependency graph induced by $T$ consist of at least one cycle, because each thread-id $t$ has an outgoing and incoming edge:

$$\begin{aligned} T = T_{middle} &= \{t \in T \mid \neg t \in T_{sink} \ \wedge \ \neg t \in T_{source}\} \\ &= \{t \in T \mid \exists t' \in T : (t,t') \in \mathbb{D}_P \ \wedge \ \exists t' \in T : (t',t) \in \mathbb{D}_P\}. \end{aligned}$$

Choose a SCC containing the cycle. The cycle has at least two elements (no reflexive edges), therefore this SCC forms a dependency cluster. □

If $T$ consists of a cluster then a KEP thread-id $t$ is chosen to get the next thread-id value. This ensures the termination of the algorithm. The thread $t$ gets the lowest thread-id value of thread-ids in $T$. As a heuristic to minimize the hard propagated thread dependencies, the amount of its writer and reader count is minimized and maximized respectively. Therefore choose $t$ as follows:

$$t \in \{t' \in T \mid \forall t'' : \ \#w_T(t') - \#w_T(r') \ \leq \ \#w_T(t'') - \#r_T(t'')\},$$

whereby $\#w_T(t) := |\mathbb{D}_P(t,.) \cap T \times T| \in \mathbb{N}$ and $\#r_T(t) := |\mathbb{D}_P(.,t) \cap T \times T| \in \mathbb{N}$ are defined as the count of propagated dependency writers respective readers of $t$ in $T$.

The complete algorithm as described is shown by Figure 7.4 in pseudo-code. Note that $T_{middle}$ might consist of multiple clusters. It is a presumption and has yet to be proven that the heuristic computes a thread-id value assignment with a minimum of hard propagated thread dependencies.

In Figure 7.5 an example is shown: first the thread dependencies are computed, three in number, so the original CKAG has at least three signal dependencies. After the thread dependency propagation (b), one thread dependency is changed, namely $(T3, T5)$ to $(T1, T4)$, and is now at the highest possible concurrent level at their least common fork. All propagated thread dependencies are hard, because the writer id values are each lower than the reader ones. Each sub-graph of the propagated thread dependency graph consists only of two nodes and one edge. This can easily be solved and results in a perfect solution with no hard dependencies, because no cluster exists, see (c). An example that consists of a cluster is presented in Figure 7.6. The heuristic reaches here the minimum of two hard propagated thread dependencies.

### 7.5.2 Weighted Propagated Thread Dependencies

The KEP thread-id value optimization described in the previous section can be refined by taking the thread dependencies and signal dependencies respectively into account. This optimization technique is currently not implemented, in contrast to the non-weighted thread-id value optimization, which is; but it is based on a similar—more general—principle and a possible implementation will benefit from this. This optimization will not effect a perfect thread-id value match, which will be, if possible, computed by the current algorithm. If a dependency cluster occurs, the best case is not possible, see Theorem 7.5.1: Currently only the hard propagated thread dependencies are be minimized. Each propagated thread dependencies might underlie several thread dependencies respective signal dependencies. The concept of *hard propagated thread dependencies* can analogously be transferred to thread dependencies and signal dependencies:

**Definition 7.9** (Hard Dependencies). *Given a thread dependency $t = (t_w, t_r) \in \mathbb{D}_T$ and a signal dependency $d = (d_w, d_r) \in \mathbb{D}$ of a KEP thread-id tree $\mathbb{T}$, $t$ and $d$ are called* hard *in $\mathbb{T}$ if the writer's thread-id value is less than its reader value:*

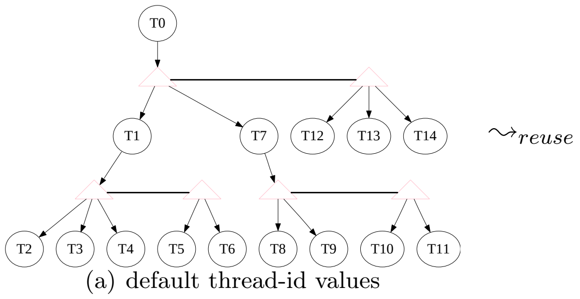
$$t \textit{ hard in } \mathbb{T} \;:=\; t_w.id_{\mathbb{T}} < t_r.id_{\mathbb{T}} \quad \textit{and respective}$$

$$d \textit{ hard in } \mathbb{T} \;:=\; d_w.threadid.id_{\mathbb{T}} < d_r.threadid.id_{\mathbb{T}}.$$

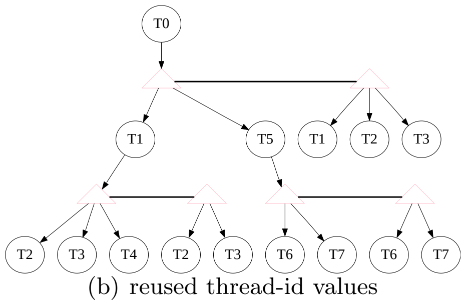
*The sets of hard thread and signal dependencies are predicated as $\mathbb{D}_T^{h,\mathbb{T}}$ and $\mathbb{D}^{h,\mathbb{T}}$ respectively.*

The basic idea of a further KEP thread-id value optimization is to take the number of thread or signal dependencies into account to choose a KEP thread-id in order to dissolve a dependency cluster with a minimum of hard thread or signal dependencies.

```
procedure new_id_reuse(thread−id $t$)
  $t.id = id\_value\_counter{+}{+}$;// assign id value

  int $max = 0$;
  int $start = id\_value\_counter$;

  forall $f \in t.forks$ do
    new_ids_reuse($f.subthreads$);
    $max$ = max($id\_value\_counter$,$max$);

    if ($f = t.forks$.end()−1) then
      $id\_value\_counter = max$;
    else
      $id\_value\_counter = start$;// reset
    fi
  end
end
```



(a) default thread-id values



(b) reused thread-id values

Figure 7.7: The algorithm of the KEP thread-id reusage is identical with the algorithm described in Figure 7.4, except that the new procedure *new_id_reuse* is used instead of *new_id*. The procedure *new_ids_reuse* is defined as *new_ids*, where calls of *new_id* are replaced by *new_id_reuse*. The procedures differ from each other by the resetting *id_value_counter* to *start* and *max* respectively. The maximum thread-id value number can be reduced by reusing thread-id values for KEP thread-ids, which are in sequence. These are each time the subthread trees of the three two-element *fork node* lists. In this case the maximum value halves by the value reuse from (a) 14 to (b) 7.

A new value assignment $id_{\mathbb{T}'}$ should now minimize $|\mathbb{D}_T^{h,\mathbb{T}}|$ and $|\mathbb{D}^{h,\mathbb{T}}|$ respectively. The additional number information can be represented as weights of propagated dependency edges $\mathbb{D}_P$.

**Definition 7.10** (Weighted Propagated Dependencies)**.** *Given a propagated thread dependency $p = (p_w, p_r) \in \mathbb{D}_P$, the dependency $(p_w, p_r)_n$ is called weighted propagated thread dependency with edge weight $n \in \mathbb{N}$.*

*We call $p_n$ weighted by thread dependencies if $n$ is the number of all thread dependencies with the propagated thread dependency of $p$:*

$$n = |\{(t_w, t_r) \in \mathbb{D}_T \mid t_w \leq p_w \ \wedge \ t_r \leq p_r\}|$$

*We call $p_n$ weighted by signal dependencies if $n$ is the number of all signal dependencies with the propagated thread dependency of $p$:*

$$n = |\{(d_w, d_r) \in \mathbb{D} \mid d_w.threadid \leq p_w \ \wedge \ d_r.threadid \leq p_r\}|$$

The expression $|\mathbb{D}_P^{h,\mathbb{T}}|$ should be minimized, this time with weighted dependencies $\mathbb{D}_P$, which leads to:

$$|\mathbb{D}_P^{h,\mathbb{T}}| \ = \sum_{p_n \in \mathbb{D}_P^{h,\mathbb{T}}} n \text{ has to be minimized.}$$

Note that the minimization problem of the previous section correlates with the weighted dependency problem with all weights set to one, the algorithm differ only in the cluster solving by taking the weights into account: Given a thread dependency cluster $c$ in $T_{middle}$ of weighted propagated thread dependencies, the further optimization algorithm chooses (to dissolve $c$) a thread-id $t$ with a minimum count of writer minus reader edges $(\#w'_T(t) - \#r'_T(t))$ with respect to the weights, *i. e.*,

$$\begin{aligned} \#w'_T(t) \ &:= \ \sum\{n \mid (t, .)_n \in \mathbb{D}_P \cap T{\times}T\} \quad \text{and} \\ \#r'_T(t) \ &:= \ \sum\{n \mid (., t)_n \in \mathbb{D}_P \cap T{\times}T\}. \end{aligned}$$

### 7.5.3 Thread-Id Reuse

Another optimization of KEP thread-ids, which is independent of the one described before and can be applied additionally, is the reuse of KEP thread-id values to lower the range of used KEP thread-id values. As mentioned in Section 4.2, the integer values of KEP thread-ids must be different, when they could be active at the same time. This might be the case when they are concurrent. Also KEP thread-ids in sub-thread relation must have different values, because the sub-threads must have greater id values according to KEP thread-id tree properties. The only way is to reuse id values, if KEP thread-ids are in sequence relation. Note that even if KEP thread-id values might be the same, their KEP thread-ids remain different.

This section describes an algorithm to reuse KEP thread-id values for KEP thread-ids related in sequence and simultaneously accomplish the KEP thread-id tree rules.

The id value assignment is done by a DFS algorithm on the thread tree structure, the same way as the default values are computed with an increasing value counter. The only difference is that we reset this counter for sequentially related KEP thread-ids. So all sub-thread KEP thread-ids should start for each sub-thread generating fork of a KEP thread-id with the same reused start id value. See Figure 7.7 to see the algorithm in pseudo-code, the reuse is there implemented by saving the counter value at start and resetting it for each new *fork node*. To ensure rule (ii) of the KEP thread-id tree properties, the maximum used id value by all sub-trees of *fork nodes* is saved to reset the counter for the next—then concurrent—KEP thread-id. See the example in Figures 7.7 (a) and (b), the two sub-trees of the (first) thread-id with id value one have a maximum id value of four, therefore gets the concurrent KEP thread-id an id value of five. Without this resetting the value would be four, because the counter stopped for the second fork at value three, which would violate the rule that concurrent KEP thread-ids must have different id values, as two concurrent thread-ids have an id value of four.

The maximum used KEP thread-id value in the example is decreased by reuse from fourteen to seven. How much effect the reuse optimization has, this depends in general on the amount of thread-ids that are in sequence relation to each other. The reuse has no effect when the sequence relation is empty, but might on the other hand be arbitrarily high, if an accordingly long list of *fork nodes* exists in a thread, since their sub-trees are all in sequence relation to each other. In the example three fork lists exist, each of length two greater one, so the reusage has an impact on all of them.

The thread-id value optimization according to dependencies and the reuse are independent of each other, because the dependency assignment whether they are weighted or not is performed to order the sub-thread of a *fork node*, which are never in sequence relation, and the other way round sequential thread-ids are per definition never part of any type of dependency relation.

# 8 Experimental Results

To validate the correctness of the strl2kasm compilation scheme, as well as of the KEP itself, we have collected a fairly substantial validation suite, currently containing some 700 Esterel programs. These include all common benchmarks, such as the *Estbench* [13], and other programs written to test specific situations and corner cases. An automated regression procedure compiles each program onto KEP assembler, downloads it into the KEP, provides an input trace for the program, and records the output at each step. The evaluation platform is shown in Figure 8.1. The user interacts via a host computer with a FPGA Board that runs the KEP as well as some testing infrastructure. First, an Esterel program is compiled into a KEP object file (.ko) which is uploaded to the FPGA board. Then the host provides input events to the KEP and reads the generated output events (.keso). The input events can either be provided by the user interactively, or they can be supplied via an .esi file. *EsterelStudio V5.0* [20] is used to compute these input traces automatically with state and transition coverage. The host can compare the output .keso results of the KEP to an execution trace (.eso) also generated by *EsterelStudio*. Note that the traces obtained this way still do not cover all possible paths, but this comparison to a reference implementation proved to be a very valuable aid in validating the correctness of both the KEP and the strl2kasm.

The renaming of interface signals due to the KEP case insensitivity leads to an undesired side effect: a correct KEP assembler program fails the semantically equality test of .keso and .eso files. Therefore, for a proper test, the interface signals should already be case insensitive in the Esterel source. Due to technical reasons and simplicity, it is also demanded that the file names of the Esterel test cases coincide with the main module names.

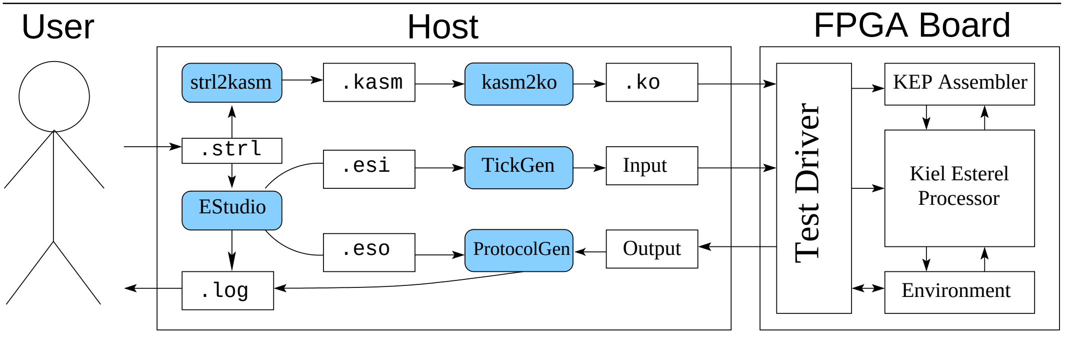


Figure 8.1: The structure of the KEP evaluation platform

| Esterel | | | KEP Assembler | | | | | | $t_{assign}$ | $t_{comp}$ |
|---|---|---|---|---|---|---|---|---|---|---|
| Module name | LoC | LoC$_{dis}$ | LoC$_{kep}$ | #Dep. | Depth | Max.Conc. | #PRIO | Max.Prio | [ms] | [ms] |
| abcd | 176 | 232 | 167 | 36 | 2 | 4 | 30 | 3 | 2.1 | 11.9 |
| abcdef | 260 | 344 | 251 | 90 | 2 | 6 | 48 | 3 | 3.3 | 16.8 |
| eight_buttons | 344 | 456 | 335 | 168 | 2 | 8 | 66 | 3 | 4.7 | 67.9 |
| channel_protocol | 53 | 78 | 61 | 8 | 3 | 4 | 10 | 2 | 0.7 | 5.7 |
| reactor_control | 31 | 50 | 32 | 5 | 2 | 3 | 0 | 0 | 0.5 | 4.1 |
| runner | 35 | 69 | 38 | 2 | 2 | 2 | 0 | 0 | 0.5 | 4.9 |
| ww_button | 100 | 184 | 134 | 6 | 3 | 4 | 6 | 2 | 1.7 | 10.6 |
| tcint | 341 | 505 | 472 | 65 | 5 | 17 | 45 | 3 | 6.3 | 98.9 |

Table 8.1: Experimental results of the priority assignment. The lines of code (LoC) of the Esterel source is compared with the target KEP assembler. Furthermore the amount of dependencies is presented and how many priority instructions are used to solve them. Note that the range of used priorities is low, as can be seen by the maximal priority values.

Table 8.1 summarizes the experimental results for a selection of programs taken from the *Estbench*. Obviously the generated code is very compact, and the KEP assembler line count is comparable to the Esterel source. This is primarily a reflection on the KEP ISA, which provides instructions that directly implement most of the Esterel statements. The connection between program size and number of signal dependencies is rather loose. For example, eight_buttons is smaller than tcint but contains more than twice the number of dependencies. The degree of concurrency again varies widely; not too surprisingly, it also influences the required number of PRIO statements (which induce potentially context switches). The reason is that signal dependencies exist only between concurrent threads, and these induce the dependency constraints. On the other hand, a program with an arbitrarily high concurrency level might contain not a single PRIO instruction. However, the overall number of generated PRIO statements seems acceptable compared to overall code size, and there were cases where the insertion of PRIO instructions was not necessary at all, despite several signal dependencies. This effect is enhanced when the thread-id assignment optimization mechanism is used, as Table 8.2 shows. Similarly, the maximum assigned priorities tended to be low in general, for none of the benchmarks they exceeded three. The priority assignment algorithm and the overall compilation are quite fast, generally within the millisecond range.

Next, the compilation results of Table 8.1 are compared with their optimized counterparts: this time compiled by setting the optimization flag -o when executing the cec-astkep module. Data that are independent from the optimization techniques are not shown again. These are the Esterel LoC, dependency, thread depth and the maximum concurrency columns. There is no big difference in the overall compilation time between normal and optimized compilation. This indicates an efficient implementation of the optimizations. Both the thread-id value assignment and the collapsing optimization decrease the KEP assembler LoC. We consider first the ef-

| Esterel | KEP Assembler (optimized) | | | | $t_{assign}$ | $t_{comp}$ | Collapsing | | | $t_{\text{collapsing}}$ |
|---|---|---|---|---|---|---|---|---|---|---|
| Module name | $\text{LoC}_{opt}$ | decr.[%] | #PRIO | Max.Prio | [ms] | [ms] | #HALT | #AWAIT | #SUSTAIN | [ms] |
| abcd | 152 | 9.0 | 30 | 3 | 2.2 | 12.6 | 0 | 6 | 0 | 0.4 |
| abcdef | 232 | 7.6 | 48 | 3 | 3.6 | 52.9 | 0 | 8 | 0 | 0.6 |
| eight_buttons | 312 | 6.9 | 66 | 3 | 5.6 | 25.9 | 0 | 10 | 0 | 0.8 |
| channel_protocol | 55 | 9.8 | 10 | 2 | 0.8 | 6.5 | 0 | 2 | 0 | 0.1 |
| reactor_control | 22 | 31.3 | 0 | 0 | 0.5 | 4.0 | 1 | 3 | 1 | 0.1 |
| runner | 26 | 31.6 | 0 | 0 | 0.6 | 5.1 | 2 | 3 | 2 | 0.1 |
| ww_button | 94 | 29.9 | 0 | 0 | 1.7 | 11.0 | 10 | 12 | 0 | 0.4 |
| tcint | 354 | 25.0 | 0 | 0 | 6.7 | 91.5 | 2 | 13 | 17 | 1.3 |

Table 8.2: Results of the Compiler Optimizations. The KEP assembler LoC have been decreased due to the thread-id value optimization and the collapsing.

fect of the thread-id value optimization, which is expressed by the number of PRIO instructions. For the *ww_button* and *tcint* examples, the number of PRIO instructions is decreased from 6 and 45 to zero, respectively. The PRIO count remains unchanged for the other examples, in particular, none of the 66 PRIO instructions of the *eight_buttons* example is resolved.

The impact of the collapsing optimization is shown by columns #HALT, #AWAIT and #SUSTAIN counting each time the according pattern is found. Note that the number of collapsed HALT instructions in Table 8.2 does not include the count of collapsed HALTs found intermediately during the collapsing of AWAIT patterns. As expected, the number of detected collapsing patterns correlates with the increase during the Esterel dismantling, *i. e.*, the programs with the highest dismantling increase have also the highest collapsing potential. For the *runner* example, the Esterel dismantling increases the code size by 97% and it has in relation to its code size the highest count of patterns found. Even for the *ww_button* and *tcint*, the collapsing affords good results. The overall collapsing speed is fast, when compared to the priority assignment.

In conclusion, the strl2kasm and its optimizations provide efficient code in a timely manner. A further testing could examine which effect the thread-id optimization and collapsing have on their own; it is assumed that the thread-id optimization has a significant positive effect on the collapsing since less patterns are destroyed by PRIO instructions. It would also be interesting to see whether the implementation of the weighted thread dependency optimization could decrease the PRIO count in those examples that expose no effect for the non-weighted optimization.

# 9 Implementation

This chapter illustrates the implementation of the strl2kasm compiler. The compiler is constructed of several modules. It uses the CEC [15] as a front-end, see Figure 9.1, namely the CEC parsing and module expanding are used. These are performed by cec-strlxml and cec-expandmodules, respectively. The CEC parser makes use of the *Antlr* [33] parser generator. The Esterel grammar and semantics are specified by files esterel.g and staticsemantics.g, respectively. The modules interact by *XML* code that represents the Esterel *Abstract Syntax Tree* (AST). After the Esterel source is parsed and printed to AST XML, this is the input of module cec-expandmodule. After the module expansion is performed, the output is again XML AST code. The XML printing is provided by the *Expat* [14] tool, which is used in the CEC by class IR::Node. In the strl2kasm implementation, the data structures for the KEP assembler and CKAG are inherited from IR::Node. This allows to print the CKAG and all other data structures for the purpose of further use by the strl2kasm in later compilation steps. All modules are shown in Figure 9.2 (b). Next the strl2kasm modules are explained, which implement the compilation steps described in the previous chapters.

## 9.1 Compiler Modules

The back-end consists of four modules: cec-kepdismantle, cec-astkep.cpp, cec-xmlkasm.cpp and cec-kepdot.cpp. Their sources are defined accordingly by the files cec-kepdismantle.cpp, cec-astkep.cpp, cec-xmlkasm.cpp and cec-kepdot.cpp, respectively. As for the CEC, all modules and related classes are implemented in the C++ [35] programming language.

The KEP dismantling, described in Chapter 3, is performed by module cec-kepdismantle. The input and output are Esterel AST files. This module replaces the CEC dismantling of module cec-dismantle. The KEP dismantling is implemented by the class KepDismantler based on class Rewriter of the CEC. It is defined in file KepDismantle.hpp.

The dismantling is made optional for some Esterel statements that are directly supported by the KEP instruction set. These statements are halt, await, sustain and wabort. The dismantling is activated by the option d, for dismantle, followed by the type of statement: -d $\langle stmt\_type \rangle$. When this option is not used, the according statements will not be dismantled. In general, all delayed statements must be dismantled to ensure a correct priority assignment. Therefore, the compilation script uses the following default dismantling options: -d halt -d await -d sustain. The instantaneous wabort statement is not dismantled by default, since it is directly supported

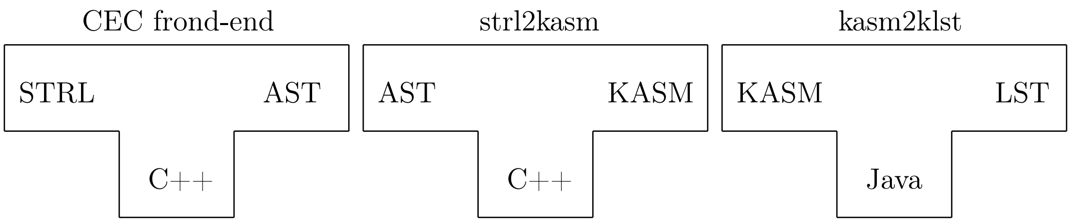
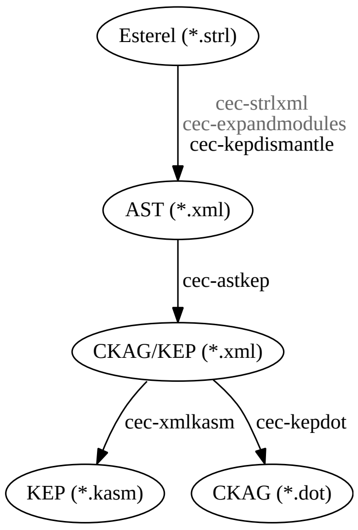
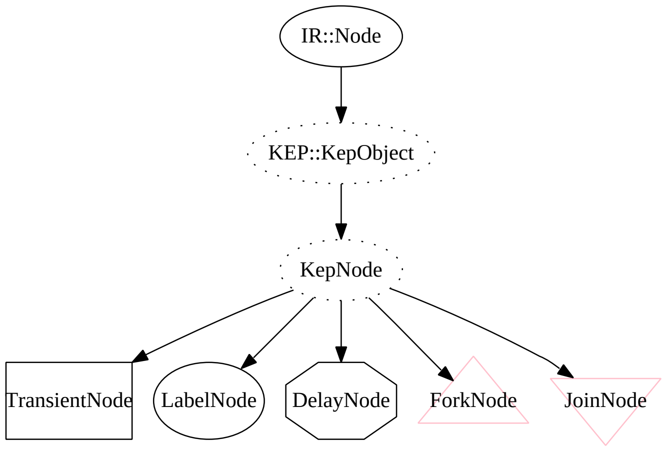


Figure 9.1: Compiler Overview



(a) Compiler Module Structure



(b) Node Class Hierarchy

Figure 9.2: Compiler Implementation

by the KEP instruction WABORT. The complex variants of await and abort are always dismantled, see Section 3.4.

The main compilation steps are performed by cec-astkep: the CKAG construction, priority assignment and the optimizations. The CKAG construction is performed as traversal over the Esterel AST. This is implemented by class EsterelKep in EsterelKep.*pp. Class EsterelKep is a visitor of the AST structure and therefore inherited from class AST::Visitor, a AST class. In cec-astkep.cpp an instance of EsterelKep is created, together with an instance of the CKAG building class KEP::CKAG_Builder. During traversal, the CKAG is created. Thereafter, the following compilation steps are performed by the according visitor classes. The optimizations are activated by option o for optimized. At the end, the computed CKAG and KEP assembler program are printed to XML.

An example compilation trace:

```
cec-strlxml < example.strl | \
cec-expandmodules | \
cec-kepdismantle -d sustain -d await -d halt | \
cec-astkep -o | \
cec-xmlkasm -i -p LINES -p PRIO > example.opt.kasm
```

First, the example Esterel program is given as input to cec-strlxml; the resulting XML output is piped as input for cec-expandmodules.

At the end is the assembler program printed by cec-xmlkasm to example.opt.kasm. The address labels are indented by option i, line numbers are added by -p LINES and additional information about the priority assignment are printed by -p PRIO. This printer, with all its options, is explained next inter alia in detail.

The strl2kasm supports for both the KEP assembler program and the CKAG a printer: cec-xmlkasm and cec-kepdot. The cec-xmlkasm printer outputs the program in the kasm format. It is implemented in file cec-xmlkasm.cpp by class KasmPrinter, also a visitor over the KEP structure. This module supports options that affect the layout of the KEP assembler program: the *verbose* option v prints additional information about the compilation process. The option *indent*, applied by i, indents the instructions to the address labels. Information about the instructions are printed when using the p option:

- -p LINES: The line position is printed as L$\langle pos \rangle$. There are leading zeros added to *pos*, if needed, depending on the overall program size. This ensures the same length for all printed line information.
- -p THREADS: If the instruction is represented by a node in the CKAG, as most are, its thread-id value *id* is printed as T$\langle id \rangle$.
- -p PRIO: The assigned priorities of the priority assignment are printed as P$\langle prio \rangle$(/$\langle prionext \rangle$).
- -p WCRT: The computed WCRT's from the WCRT analysis are printed as W$\langle wcrt \rangle$(/$\langle wcrtnext \rangle$).
- -p ALL: All printings are enabled.

This information is enclosed by '[' and ']' in front of the instructions, as seen in previous program examples. Multiple information are divided by commas. The use of p followed by an underline before an option (-p _$\langle option \rangle$) excludes an option, *e. g.*, if all information, except the line information, should be printed, use -p ALL -p _LINES. An option is not undone when added explicitly before, *e. g.*, by using -p LINES -p _LINES, the line information are still printed.

The cec-kepdot prints a CKAG to the *dot* format, which is used as graph description language by *graphviz* [24]. The visitor class KEPDOT in cec-kepdot.cpp implements the dot printing. In Figure 9.3 (a) an example *dot* file is shown that was generated by the printer. The according CKAG is visualized by dot/graphviz, see the result graph layout in Figure 9.3 (b). The same p printing options of cec-xmlkasm can applied in cec-kepdot. In that case, the specific nodes are enriched by an according string. The option g and a succeeding CKAG or THREADS specify the type of printed graph. As default is the CKAG printed, equivalent to -g CKAG; an additional -g THREADS prints both, the CKAG and its thread-id; -g THREADS prints only the thread-id tree.

## 9.2 Data Structures and their Visitor Classes

The basic data structures used for KEP instructions and CKAG nodes are implemented by files KEP.hpp/cpp. All classes defined within these files are derived from class KepObject, which is on their part inherited from the CEC class IR::Node to implement the expat [14] *XML* functionality.

For each type of node described in Section 4.1 an according class is defined: the classes TransientNode, LabelNode, DelayNode, ForkNode and JoinNode implement the *transient nodes*, *label nodes*, *delay nodes*, *fork nodes* and *join nodes* respectively. The node class hierarchy is presented in Figure 9.2 (c). They all inherit from class KepNode, which implements functionality that all nodes have in common: access to its children respective parents and the instruction, that it represents. The children of a node are seen as a set, although they are implemented as a vector: before the insertion of a new child, it is tested, whether it is the child already inserted. If not, it is inserted and the new parent of the child is set. This ensures the parent-child consistency that must hold true for the CKAG:

$$\forall p, c \in N : \; c \in p.children \;\Leftrightarrow\; p \in c.parents.$$

The class DelayNode has additional fields and methods for the preemption edges, as well as the classes ForkNode and JoinNode are specified according to their properties.

All instructions are sub-classes of class KepInstruction, whereby the class hierarchy is refined by classes SignalInstruction, AddressInstruction, DataInstruction, RegisterInstruction, PriorityInstruction and OpInstruction to implemented the specific behavior if needed. *E.g.*, the class Present is a sub-class of classes SignalInstruction and AddressInstruction, because it consists of both, a signal to test for and an else address to jump to. These two classes are inherited virtually, because both are again sub-classes of KepInstruction.

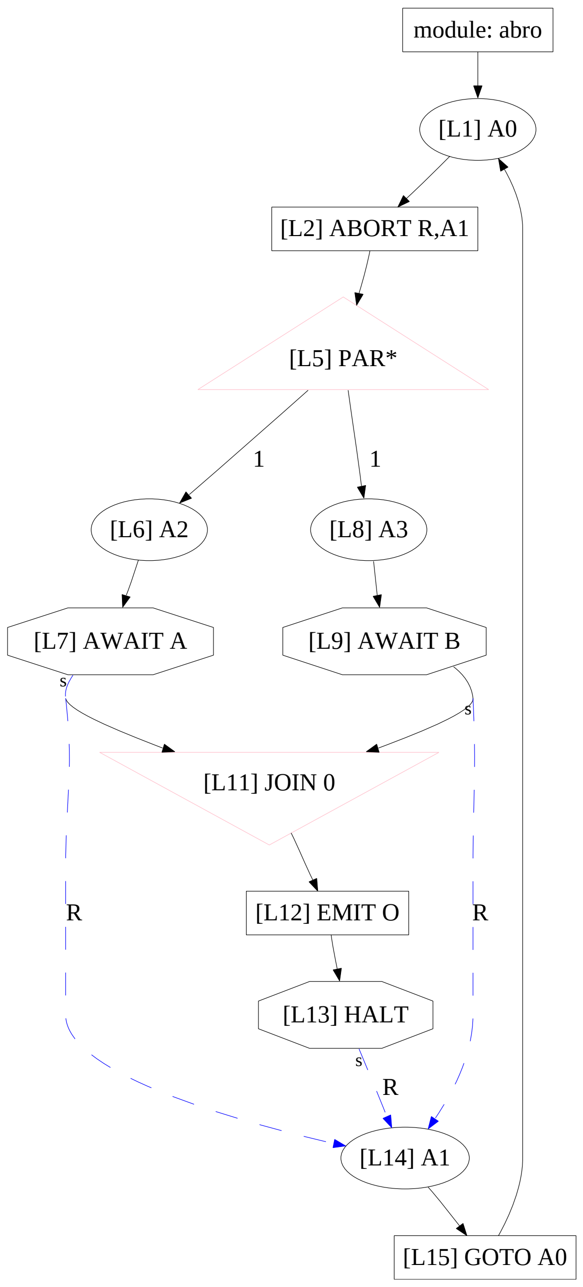
All data structures allow the use of the visitor pattern [23]. The main visitor from which all other visitors are derived from, is also defined by KEP.hpp/cpp. A default visit method is defined for each class, except the virtual ones that are never instantiated. These default methods have the following method body:

```
std::cerr << "\nTYPEID: " << typeid(*this).name() << "\n"; assert(0);
```

that prints the visitor class' name and the execution stopped by an assertion. The method that causes the termination is printed, so it can be easily determined, which method the current visitor is not already implemented, when implementing a new

```

digraph G {

 size="6,7";
nodesep=0.25;

// global graph settings
concentrate=true;

// global node default values
node [color=black fontsize=20 shape=box ];

// global edge default values
edge [color=black fontsize=20
 labelfontsize =15 ];

/// −−−−−< CKAG GRAPH >−−−−−
// NODES
N0 [label="module: abro" ];
N1 [label="[L1] A0" shape=ellipse ];
N2 [label="[L2] ABORT R,A1" ];
N3 [label="[L5] PAR*"
shape=triangle color=pink];
N4 [label="[L6] A2" shape=ellipse ];
N5 [label="[L7] AWAIT A" shape=octagon];
N6 [label="[L8] A3" shape=ellipse ];
N7 [label="[L9] AWAIT B" shape=octagon];
N8 [label="[L11] JOIN 0"
shape=invtriangle color=pink];
N9 [label="[L12] EMIT O" ];
N10 [label="[L13] HALT" shape=octagon];
N11 [label="[L14] A1" shape=ellipse ];
N12 [label="[L15] GOTO A0" ];

// EDGES
N0 −> N1 [];
N1 −> N2 [];
N2 −> N3 [];
N3 −> N4 [label=" 1" ];
N3 −> N6 [label=" 1" ];
N4 −> N5 [];
N5 −> N8 [];
N5 −> N11 [color=blue style="dashed"
 taillabel ="s" label="R" ];
N6 −> N7 [];
N7 −> N8 [];
N7 −> N11 [color=blue style="dashed"
 taillabel ="s" label="R" ];
N8 −> N9 [];
N9 −> N10 [];
N10 −> N11 [color=blue style="dashed"
 taillabel ="s" label="R" ];
N11 −> N12 [];
N12 −> N1 [];

// LAYOUT
N4 −> N6 [style=invis]
{rank=same N4 N6}

}
```

(a) Dot



(b) PDF

Figure 9.3: Dot File and its layouted PDF.

visitor class, *e. g.*, if in class PreemptionHandler that sets the preemption edges during the CKAG building is not implemented the method visit(KEP::DelayNode&), the following message of the default method is printed, when applied on a *delay node*:

```
TYPEID: N3KEP17PreemptionHandlerE
cec-astkep: KEP.hpp:1524:
virtual void KEP::Visitor::visit(KEP::DelayNode&): Assertion ‘0’ failed.
```

This example illustrates how the implementation of new KEP visitor classes is supported by their default methods.

The CKAG assembler graph and its creation are implemented in CKAG.hpp and CKAG.cpp. The building of sequential KEP programs is realized by class KAG_Builder, the concurrent building is completed by CKAG_Builder inherited from KAG_Builder. During the building process several helper classes are used, these are inter alia explained.

Miscellaneous helper classes for graph building and monitoring the compilation plus several classes that implement an optimization as described in Chapter 7 are implemented by KepHandler.hpp and KepHandler.cpp, all these classes are typically visitors of the KEP data structures inherited from KEP::Visitor.

- BasicHandler: This class provides the basic functionality that is used by other handlers. Therefore these are inherited from the BasicHandler. Especially the use of class KEP::Debug is provided to handle debug printings, as well as warning and error messages.

- PreemptionHandler: This handler represents the preemption environment during the CKAG building, therefore it consists of push and pop methods for each preemption type to indicate the start and end of a preemption scope, respectively. The preemption edges of delay or *join nodes* are set when this handler is welcomed.

- NodeCreationHandler: This handler is used during the building process to create nodes of given instructions. Therefore it uses a KEP thread-id stack to push and pop thread-ids when fork and *join nodes* are created, respectively. The top thread-id is each time used to assign the thread-id of the nodes during their creation.

- SymbolHandler: The administration of symbol namespaces is performed by this class.

  It is initialized with the KEP keywords to rename accordingly, if needed.

- ExpressionHandler: The ExpressionHandler performs the compilation of expressions. It uses again the ComputationHandler and ConditionalHandler to implement constant propagation and the building of conditional expressions.

- GraphHandler: Similar to the BasicHandler, this class is designed to be a helper for visitor classes that implement a computation on the CKAG.

- DeadCodeHandler: This class implements the previously described dead code analysis.
- ReachabilityHandler: The class ReachabilityHandler analyses instantaneous reachability of nodes in the CKAG. *E.g.*, this is needed to determine whether a fork-join might be instantaneous or delayed.
- DependencyHandler: This class computes the signal dependencies of a CKAG.
- PriorityHandler: The *priority assigning* is realized by this class. Therefore, the dependencies has already be computed by the DependencyHandler.
- ThreadHandler: The thread-id optimizations are implemented by the ThreadHandler. Additionally, the dot printing of thread-id trees is provided. This dot code is used when cec-kepdot is applied with -g THREADS.
- Def32Handler: This class replaces 32 bit integers by _UINT32REG and inserts according DEF32 instructions as described before.
- CollapseHandler: This class implements the *CKAG collapsing* optimization. It is applied after the priority assignment.

# 10 Conclusions and Further Work

I presented an Esterel compiler for the KEP, a synchronous reactive processor. Since the KEP instruction set architecture is very similar to Esterel, the compilation of most constructs is straight-forward. But the computation of a static priority schedule for concurrent threads is not trivial. The thread scheduling problem is related to the problem of generating statically scheduled code for sequential processors, for which Edwards has shown that finding efficient schedules is NP hard [17]. We have encountered the same complexity, but our performance metrics is a little different. The classical scheduling problem tries to minimize the number of context switches. On the KEP, context switches are free, because no state variables must be stored and resumed. However, to ensure that a program meets its dependency-implied scheduling constraints, threads must manage their priorities accordingly, and it is this priority switching which contributes to code size and costs an extra instruction at run time. Minimizing priority switches is related to classical constraint-based optimization problems as well as to compiler optimization problems such as loop invariant code motion. We solved the problems and the experimental results show an efficient implementation.

Since its inception, the reactive processing approach has demonstrated its promise and its practicality. However, much remains to be done. On the theoretical side, a precise characterization of the reactive execution semantics is still missing, and its relationship to other semantics needs to be investigated, in particular regarding causality issues. FPGA-based implementations of reactive processors have proven very competitive to classical processor designs. For a standard suite of Esterel benchmarks, the code size is typically an order of magnitude smaller than that of the MicroBlaze, a 32-bit COTS RISC processor core. The worst case reaction time is typically improved by 4x, and energy consumption is also typically reduced to a quarter.

It is an interesting result that even large programs like the *tcint*, which has over 400 lines of code and 65 signal dependencies, can be scheduled only by thread-id values, without additional priority changes. It seems, at least for the *estbench* examples, that this optimization has a high variance, *i. e.*, on some examples the optimization has a huge impact and on others none. However, such a perfect schedule is in general not possible.

The optimization of weighted propagated thread dependencies is a generalization of the propagated thread dependency optimization; non-weighted is equivalent to weight one. These edge weights correlate with the underlying number of signal dependencies. So it is expected that this enhancement leads to a still better optimization, but this optimization is currently not implemented and therefore could not be tested. A possible implementation does not change the algorithm in principle, only the measurement of the heuristic is changed by taking the weights into account. These could be easily computed beside the thread dependency computation, because their computation requires the iteration over all signal dependencies. For each thread dependency an according counter is created and increased for each appropriate signal dependency; when finished, these are the weights. During the computation of propagated thread dependencies, the weights have to be propagated, too. Therefore, the thread dependency weights are added to their according propagated thread dependency weights, where the weights are not added if a thread dependency is itself a propagated thread dependency.

Although the *min-writer-max-reader-heuristic* (see the Section 7.5) provides good results and we have not found any counter examples, it still has to be proven. If this heuristic is not optimal, then it is presumed that it is optimal at least for dependency clusters (again to be proven). Then, outgoing from an optimal algorithm for clusters, an optimal algorithm can be constructed for the general case. First the *SCCs* of the propagated thread dependency sub-graph are computed. This can be achieved by a DFS based algorithm, which uses the property that the *SCCs* of a directed graph and its reversed graph are identical [37]. These *SCCs* are forming a flow graph by *decomposition*: the *SCCs* are the nodes and the edges are defined over the existence of edges between the nodes of different *SCCs*. The above heuristic is now applied onto the *SCCs* in an ordering that starts at the flow sinks and end at the sources. Because the edges between different *SCCs* are no hard dependencies by construction of the thread-id value assignment and the *SCCs* are for itself optimal, therefore the overall sum would be optimal.

The Esterel pre construct is currently not implemented in the strl2kasm. The KEP supports the pre by the PRE expression that is applied on KEP signals in KEP instructions. The strl2kasm has implemented the possible use of he PRE in its data structures, but it is not used, because the CEC parsing module parses no pre expressions. Therefore no appropriate Esterel AST containing the pre is generated that would have been used to create PRE expressions. The integration of pre is in principle no problem, since it would not add any problems to the priority assignment. To use the pre would require to modify the AST structure in the CEC and therefore would result in a version branch of the CEC. This would make the integration of a future version of the CEC with the strl2kasm more problematic.

The support of *combined* valued signals is not yet fully supported. During the definition of such a valued signal, an associative operation $op$ is defined that is used to determine the overall signal value for multiple valued emissions. In such a case, the value is defined as the associative combination $op_{i=1,\ldots,n} v_i$ of $n \in \mathbb{N}$ emitted values $v_i$. Note that the emissions need to be separated from the signal value assigning. The main difficulty of implementing the combined is the initialization of the valued signal. When a combined signal is emitted the first time , the signal value is initialized accordingly; for the next emissions, the combined operation is performed with the current and the emitted value. Therefore a PRESENT test has to be made, followed by the operational value setting in the *then* and the initial emission and setting in the *else* case. Although such a control flow (present S then ... else emit S) would be non-constructive in Esterel, it would be no problem to implement this behavior in the KEP assembler program. But the current signal dependency computation leads to a program cycle, therefore an implementation of the *combine* has to erase dependencies between the combined emissions.

The KEP ISA supports directly the await case by CAWAIT instructions. These are currently not used, because after they are dismantled they are not collapsed back. The patterns needed are not yet identified, since they are much more complicated than the ones that are currently collapsed. However, the signal expressions in await case statements could be delayed and consists of Boolean operations; a delay again could be a complex computation. How these statements fit into node patterns is not obvious. Especially both the data expressions of signal delays and the Boolean signal expressions can be arbitrarily nested. Therefore their patterns have no fixed size. This suggests the extension of the collapse pattern concept to patterns that match structures in the CKAG on a higher and more abstract level. How such a *meta patterns* looks like and if whether these are feasible has to be analyzed.

The KEP registers that are used during the compilation of data expressions are freed thereafter and can be reused for the next data expression. But there is still potential for further improvement, namely a *global register allocation* [32]. Note that such an optimization must still respect the KEP symbol namespace, *i. e.*, a register symbol is only used in a set of threads that are non-concurrent to each other. Despite this condition, the global data allocation of KEP registers should not differ very much from standard, because a computation is not made across different threads. Nevertheless, the implementation of data computations is not the focus of the KEP, the reactive parts are the main design issue.

A possible further work be the integration of the strl2kasm, *KEP assembler to KEP listing* (strl2klst) and the KEP *EvalBench* to be used by a common GUI. This affects not directly the strl2kasm implementation, but its interface.

Another possible further work is the portation to other source and target languages. Languages that resemble with the synchronous and reactive approach would be possible candidates for a source language enhancement. Architectures, *e. g.*, a multi-threaded version of the EMPEROR, that use the principle of multi-threading via priorities could the object of further work.

# Bibliography

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

[2] Randy Allen and Ken Kennedy. *Optimizing Compilers for Modern Architectures: A Dependence-based Approach*. Morgan Kaufmann, 2001.

[3] Massimo Baleani, Frank Gennari, Yunjian Jiang, Yatish Patel, Robert K. Brayton, and Alberto Sangiovanni-Vincentelli. HW/SW partitioning and code generation of embedded control applications on a reconfigurable architecture platform. In *CODES'02: Proceedings of the Tenth International Symposium On Hardware/Software Codesign*, pages 151–156, New York, NY, USA, 2002. ACM Press. `http://doi.acm.org/10.1145/774789.774820`.

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

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

[6] Gérard Berry. *The Esterel v5 Language Primer*, 1999. `ftp://ftp-sop.inria.fr/meije/esterel/papers/primer.ps`.

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

[8] 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.

[9] 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.

[10] Gérard Berry and Georges Gonthier. The Esterel Synchronous Programming Language: Design, Semantics, Implementation. *Science of Computer Programming*, 19(2):87–152, 1992. `http://citeseer.ist.psu.edu/berry92esterel.html`.

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

[12] Marian Boldt, Claus Traulsen, and Reinhard von Hanxleden. Worst case reaction time analysis of concurrent reactive programs. In *Proceedings of the Workshop on Model-driven High-level Programming of Embedded Systems (SLA++P07)*, Braga, Portugal, March 2007.

[13] Estbench Esterel Benchmark Suite. `http://www1.cs.columbia.edu/~sedwards/software/estbench-1.0.tar.gz`.

[14] James Clark. The expat xml parser. `http://expat.sourceforge.net/`.

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

[16] Stephen A. Edwards. An Esterel compiler for a synchronous/reactive development system. Technical Report UCB/ERL M94/43, EECS Department, University of California, Berkeley, 1994. `http://www.eecs.berkeley.edu/Pubs/TechRpts/1994/2572.html`.

[17] 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.

[18] Stephen A. Edwards, Nicholas Halbwachs, Reinhard von Hanxleden, and Thomas Stauner. 04491 executive summary – synchronous programming - synchron'04. In Stephen A. Edwards, Nicolas Halbwachs, Reinhard v. Hanxleden, and Thomas Stauner, editors, *Synchronous Programming - SYNCHRON'04*, number 04491 in Dagstuhl Seminar Proceedings. Internationales Begegnungs- und Forschungszentrum (IBFI), Schloss Dagstuhl, Germany, 2005. `http://drops.dagstuhl.de/opus/volltexte/2005/195`.

[19] Esterel Technologies. Company homepage. `http://www.esterel-technologies.com`.

[20] Esterel Technologies. *Esterel Studio User Guide and Reference Manual*, 5.0 edition, May 2003.

[21] Sascha Gädtke. Hardware/Software Co-Design für einen Reaktiven Prozessor. Diploma thesis, Christian-Albrechts-Universität zu Kiel, Department of Computer Science, May 2007.

[22] Sascha Gädtke, Xin Li, Marian Boldt, and Reinhard von Hanxleden. HW/SW Co-Design for a Reactive Processor. In *Proceedings of the Student Poster Session at the ACM SIGPLAN/SIGBED Conference on Languages, Compilers, and*

*Tools for Embedded Systems (LCTES'06)*, Ottawa, Canada, June 2006. With accompanying poster.

[23] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns*. Addison-Wesley, 1995.

[24] Emden R. Gansner. Drawing graphs with GraphViz. Technical report, AT&T Bell Laboratories, Murray Hill, NJ, USA, November 2004. `http://www.research.att.com/sw/tools/graphviz/libguide.pdf`.

[25] 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.

[26] Nicolas Halbwachs, Paul Caspi, Pascal Raymond, and Daniel Pilaud. The synchronous data-flow programming language LUSTRE. *Proceedings of the IEEE*, 79(9):1305–1320, September 1991. `http://citeseer.nj.nec.com/halbwachs91synchronous.html`.

[27] 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`.

[28] Xin Li, Marian Boldt, and Reinhard von Hanxleden. Mapping Esterel onto a multi-threaded embedded processor. In *Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'06)*, San Jose, CA, October 21–25 2006.

[29] Xin Li, Jan Lukoschus, Marian Boldt, Michael Harder, and Reinhard von Hanxleden. An Esterel Processor with Full Preemption Support and its Worst Case Reaction Time Analysis. In *Proceedings of the International Conference on Compilers, Architecture, and Synthesis for Embedded Systems (CASES)*, pages 225–236, New York, NY, USA, September 2005. ACM Press.

[30] Jan Lukoschus and Reinhard von Hanxleden. Removing cycles in Esterel programs. *EURASIP Journal on Embedded Systems, Special Issue on Synchronous Paradigms in Embedded Systems*, pages Article ID 48979, 23 pages, 2007. `http://www.hindawi.com/getarticle.aspx?doi=10.1155/2007/48979`.

[31] Sharad Malik. Analysis of cyclic combinational circuits. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 13(7):950–956, July 1994.

[32] Steven S. Muchnick. *Advanced compiler design and implementation*. 1997.

[33] Terence J. Parr. *ANTLR Reference Manual*, 2006.

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

[35] Bjarne Stroustrup. *The C++ Programming Language, Second Edition*. Addison-Wesley, 1991.

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

[37] Robert E. Tarjan. Depth-first search and linear graph algorithms. *SIAM Journal of Computing*, 1(2):146–160, 1972.

[38] Reinhard von Hanxleden and Xin Li. The Kiel Esterel Processor Homepage. `http://www.informatik.uni-kiel.de/rtsys/kep/`.

[39] Reinhard von Hanxleden, Xin Li, Partha Roop, Zoran Salcic, and Li Hsien Yoong. Reactive processing for reactive systems. *ERCIM News*, 66:28–29, October 2006. `http://www.ercim.org/publication/Ercim_News/EN67.pdf`.

# A Examples

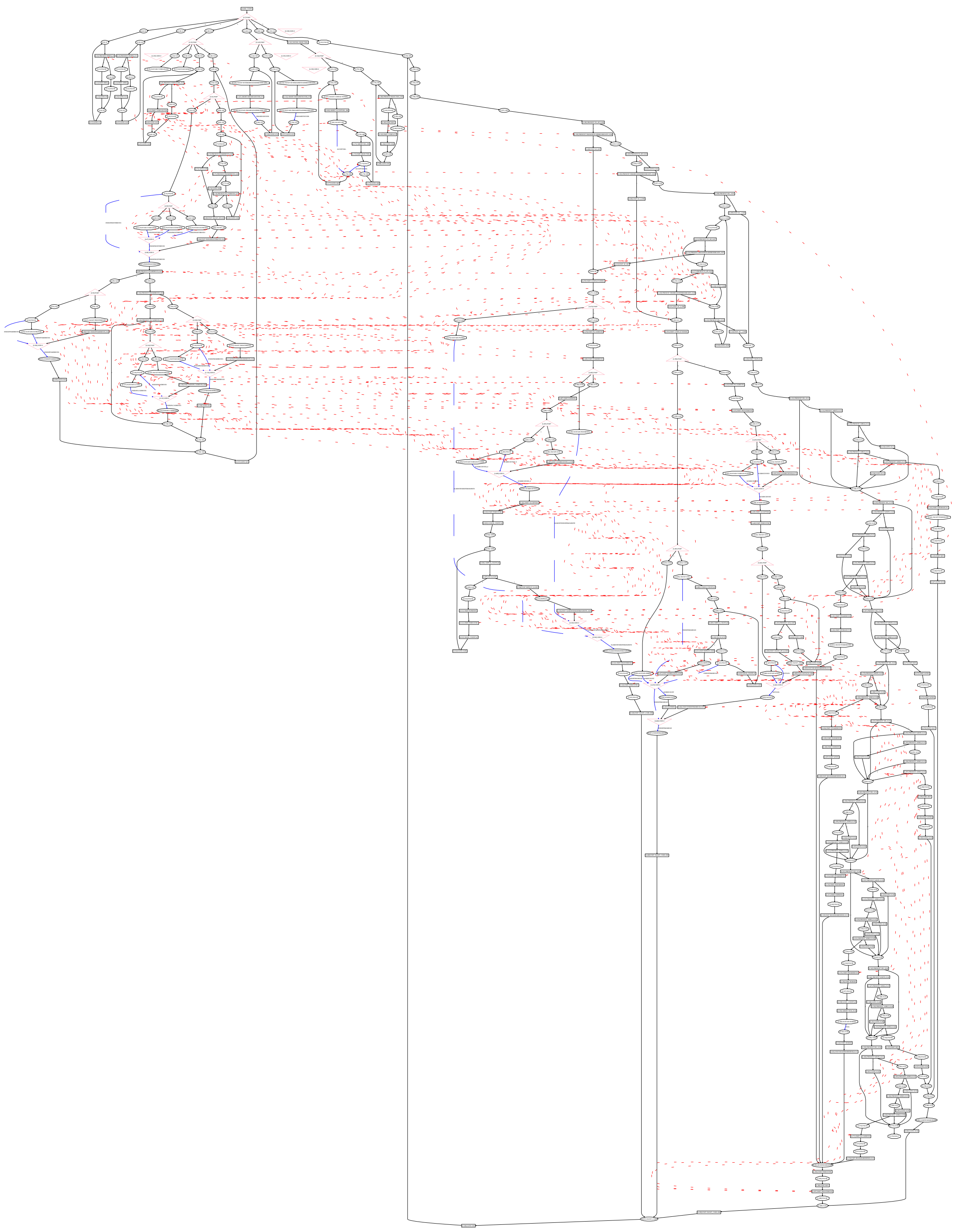


Figure A.1: The CKAG of *tcint*.

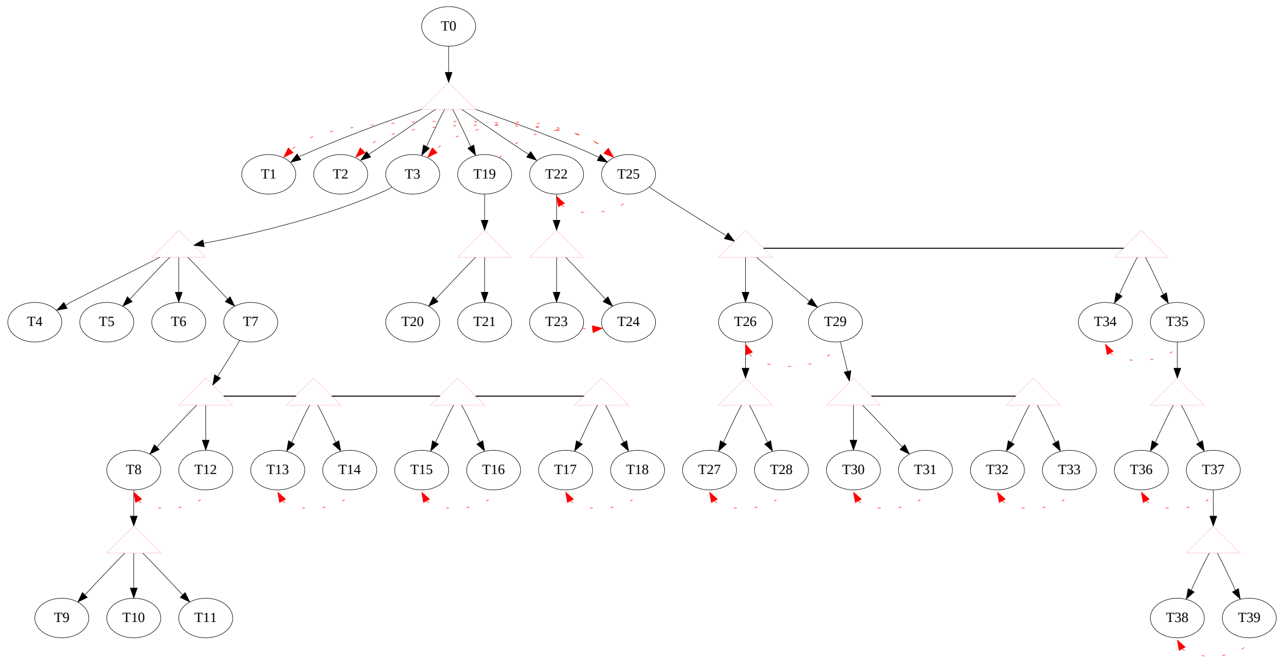


Figure A.2: Thread-Id Tree of Tcint

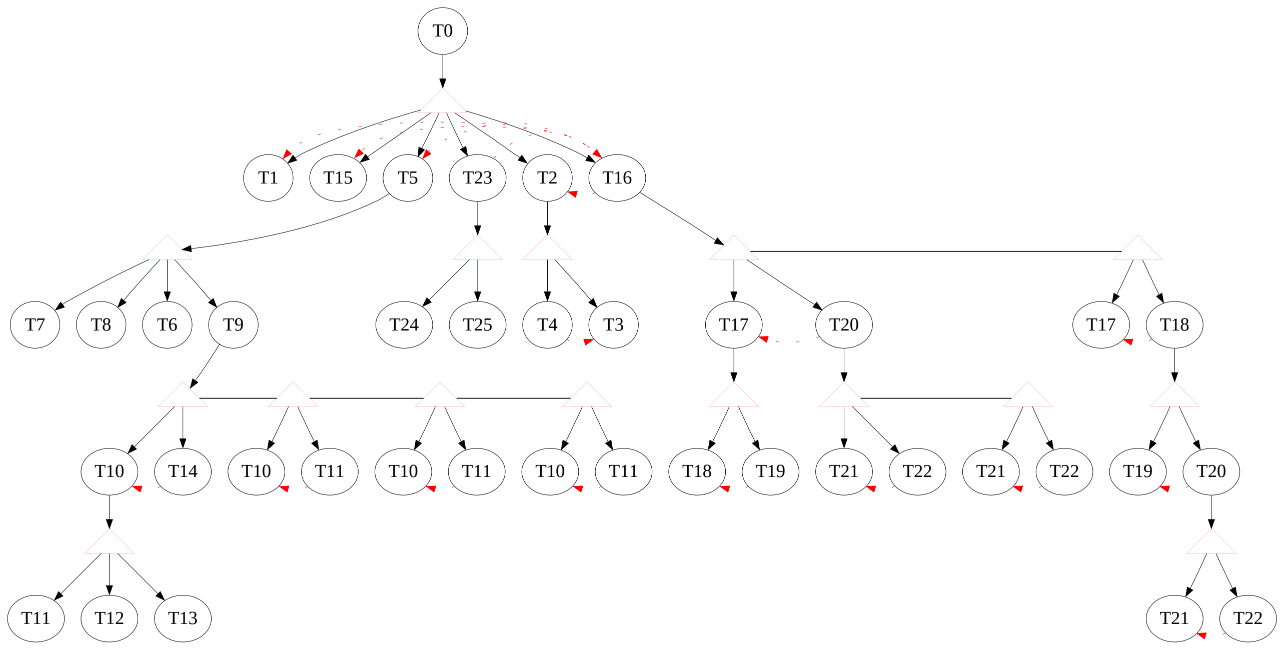


Figure A.3: Optimized Thread-Id Tree of Tcint

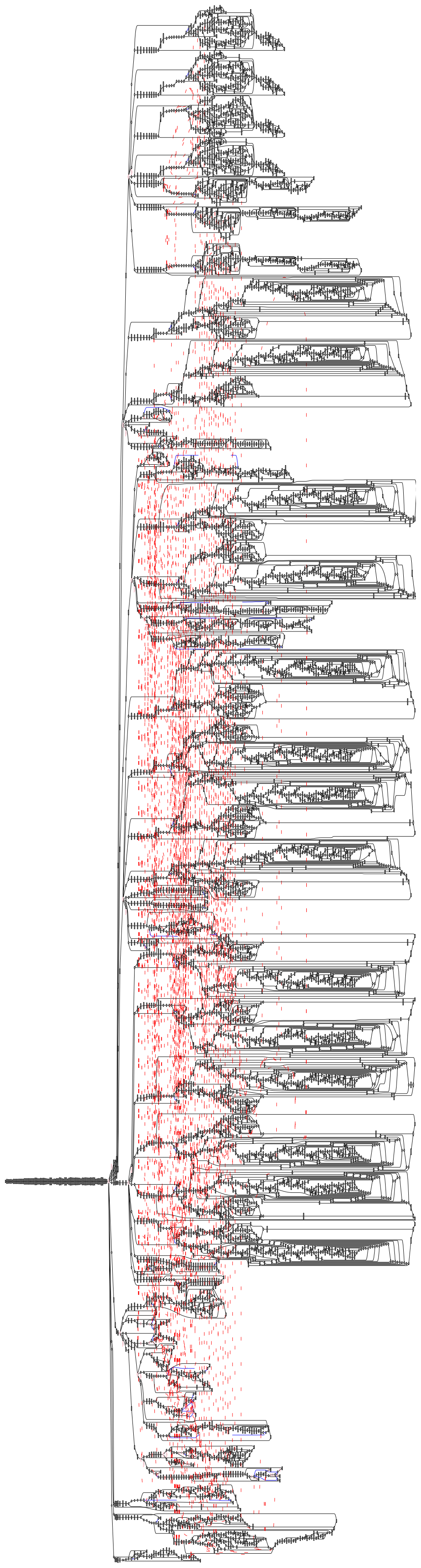


Figure A.4: CKAG of the Mca200

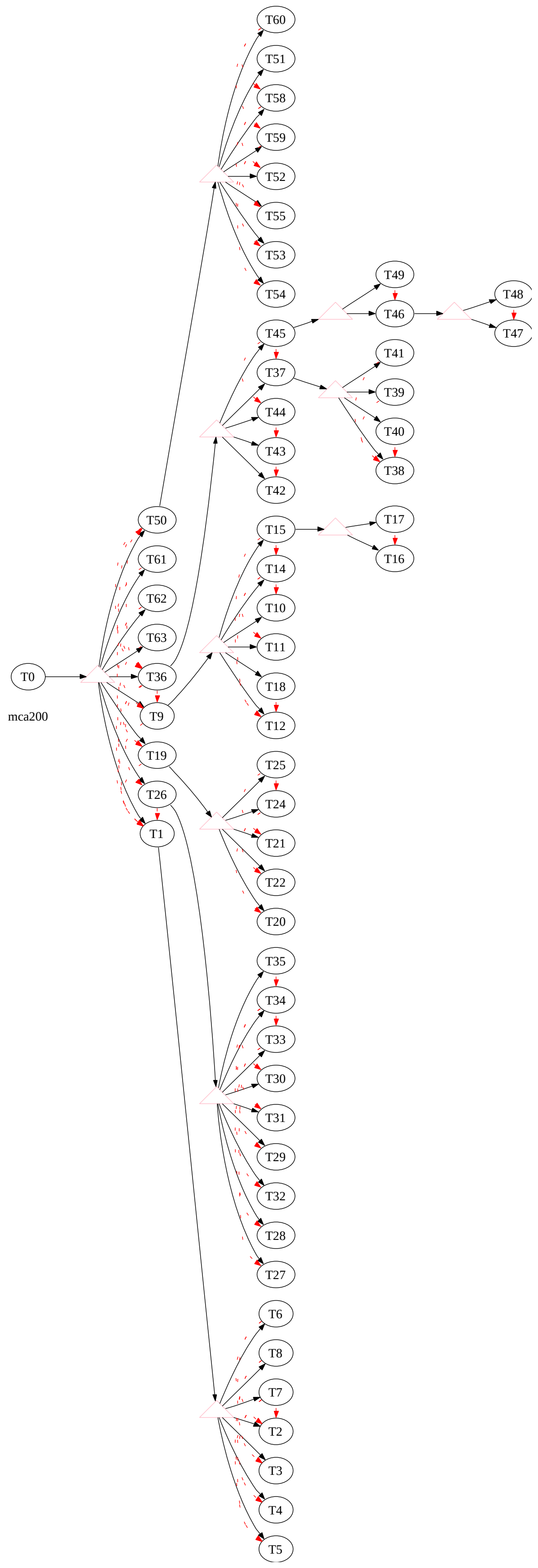


Figure A.5: Optimized Thread-Id Tree of Mca200