Christian-Albrechts-Universität zu Kiel

Diploma Thesis

Executing Safe State Machines with the Kiel Esterel Processor

Falk Starke

2009-02-03

Department of Computer Science
Real-Time and Embedded Systems Group

Prof. Dr. Reinhard von Hanxleden

Advised by:
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

Reactive programs, often used in embedded, safety critical environments, have to continuously react to stimuli from outside, processing those inputs and generating outputs. Their behavior should be deterministic and the outputs should not only be correct, but also on time. Esterel and Safe State Machines are two languages used for describing reactive programs. Yet in real life a model alone is not sufficient. It also has to be mapped to real life hardware, while making sure to retain the semantics, a difficult task. To this end, special reactive processors have been developed, for example, the Kiel Esterel Processor (KEP). Esterel can already be compiled to several reactive processors, including the KEP. Since Safe State Machines and Esterel are equivalent, they can be translated to Esterel and then compiled to a reactive processor. But might it not perhaps be more efficient to directly map Safe State Machines to an execution environment rather than taking the detour via Esterel?

It has been a rough ride. I would like to thank some people who rode with me:
Alexandra "Assja" Barchunova for occasionally throwing silly nonsense at me making my life less dull.
Özgün Bayramoglu for being more of a friend to me than most other people I call "friend".
Geoff Evans for providing suggestions from the view of a person who is into computer sciences, but not into real-time and embedded systems.
Prof. Dr. Reinhard von Hanxleden for allowing me to write my thesis as part of his workgroup, and for providing me with such an interesting topic.
My language advisor Aida Starke for not only spell checking my thesis but also providing suggestions about sentence and overall structure.
My math consultant Simon Schwardt for checking the math in this thesis, and for letting me hang out, relax and watch the Simpsons at his place every Monday.
My scientific advisor Claus Traulsen for being there when I needed his help and for not bugging me when I needed peace and quiet.

Contents

1. Introduction

> That is why we call it religion. If it was logical, we would call it science.
>
> *Prashant Nanivadekar*

Controlling airplane movement, medical life-sustaining equipment operation, or the River Thames' flood barrier: All of these applications' correctness not only depends on the correctness of calculation results, but also on their correct timing. Such systems are called real-time. Asking ourselves how exactly such systems are implemented, several ideas come to mind: the application logic might be entirely implemented in special circuits targeted exactly at the requirements of the Thames' flood barrier, embedded into the motors and sensors. Alternatively, a regular desktop PC bought at Woolworth's in London might be connected to the motors and sensors running some control program. Or there might be a mixture of both.

Two questions we don't ask ourselves often enough are: "Is this a reactive system?" and: "Do I really need a reactive system here?". Reactive systems are generally overlooked in our everyday lives, and those of us working with them often get confused about classifying or implementing specific systems because of the various, not always consistent, definitions and naming conventions, for example: "reactive system" and "real-time system."

Real-time systems are systems which are expected to adhere to some timing constraints on their computations, while reactive systems additionally always have to react in some way to input stimuli from the environment. In this work we will be dealing with synchronous reactive systems which have properties additional to the above-mentioned systems.

In synchronous reactive systems, time is treated as discrete events which occur continuously. A reactive system is connected to the surrounding environment by some inputs and some outputs (not necessarily distinct from the inputs). In every distinct event of time (called a "tick") the system reads its inputs and must react to these (speaking in the language of automata theory, this means that internally a transition originating from the current state of the system must be taken), possibly creating outputs, before the next tick. Therefore, in reactive processing, it is often said that "computations don't take any time". The process described, called a "reaction", must be deterministic. Any input and output signals are considered to be broadcasted and readable anywhere in the system, while the same holds true for local signals within their respective scope. This is called the "synchrony hypothesis". Each activation and taking of transitions within a reactive system is referred to as a "microstep", while the sum of all transitions taken in one tick, or the resulting change of state visible to an outside observer in one tick, is a “macrostep”. More concise definitions and discussions are provided by Pnueli and Harel [24, 32].

An important problem in synchronous reactive systems is the problem of causality, caused by the property that signals are instantly broadcasted over the entire signal scope. Imagine a program part that is aborted immediately when a signal S is present. How then should the system behave if the signal S is emitted from within that program part? Immediate abortion prevents execution of the aborted code, thus, it would have been impossible to emit the signal in the first place. Such paradoxical situations are called causality problems, and a system that suffers such problems is considered to be invalid.

Languages implementing the synchrony hypothesis are called “synchronous languages”. They originated from work on specifying languages that implement the restrictions of reactive processing. These languages offer control constructs compliant with the semantics of a reactive system while attempting to still be convenient for the developer. An example of such a language is the Esterel language [12, 9, 11].

Of course, just defining reactive systems in theory is not enough. Since we (seem to) know about their necessity, we also have to specify, implement, verify and execute such systems in practice (in fact, as we will see in section 1.3, in this field, practice tends to outweigh theory). For this purpose a lot of tools have evolved, most commonly programming languages. Some programming languages exist in this context only, for example, the Esterel synchronous programming language developed by Berry and Gonthier in 1992 [12], which will be further detailed in section 2.1. On the other hand, existing programming languages have been augmented in an attempt to provide reactivity, real-time Java [5] for example.

1.1. State machines

In the modern age of computer-aided software engineering, system developers make more use of graphical tools for system design and programming, which make it unnecessary for them to learn and understand the complicated syntax of yet another textual programming language. Often, a picture says more than a thousand lines of code, which is what makes graphical languages easier to learn than textual ones. There are several definitions for graphical programming languages, for example, the business process modeling notation (BPMN, see [46] and Figure 1.1) which is capable of describing everything from customs exchange to your weekly schedule to cooking recipes. Such languages were introduced to simplify the task of programming to allow people who did not study IT to create a program. This simplification reached its peak in the general public with the launch of Lego Mindstorms in 1998, giving children the opportunity to build and program their own robots. The programming language originally used for Mindstorms was a type of state machines. State machines provide a state of the art mechanism for developers to describe the behavior of reactive systems. Harel [23] contributed to the creation of the StateCharts as an extension of the finite state machine (FSM) (or automata) by adding syntax and semantics

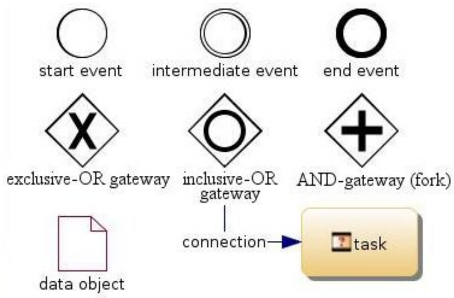


Figure 1.1.: The most common BPMN icons

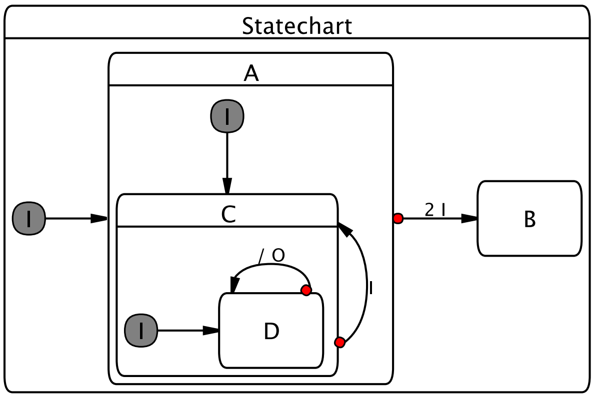


Figure 1.2.: A sample state machine

for machines executing in parallel and for machines entirely encapsulated in a state. André introduced Safe State Machines (SSM, [7]) which are strictly synchronous statecharts.

At this point, an important remark is necessary: For any given Safe State Machine (SSM), there is an equivalent Mealy machine (in terms of reactions to inputs by outputs) and some compilers which exploit this to generate code for their target architectures. However, the reason for using the additional syntax and semantics is straightforward: simplicity. Tasks that should be carried out in parallel require unintuitive Mealy machines. The growth of the number of states for parallel tasks is exponential in a Mealy machine, while only linear in a SSM (considering the worst case). Examples of the simplifications a SSM introduces will be discussed in more detail later. In real life applications, model checking and formal verification are made significantly easier through a mathematical calculus defining the semantics of a SSM.

Due to the significantly better usability of state machines, especially for the novice developer, there is also a good deal of research in layouting state machines [39, 13, 20, 34].

1.2. Reactive processing

Implementation of reactive systems can be done in several ways: using software only, pure hardware, or some hybrid system. All design techniques have their advantages and disadvantages. A pure software system, for example, can be easily changed, whereas pure hardware is usually the fastest way.

The most difficult way — hardware/software co-design — generates part software and part hardware used to run the software. This process benefits from recurring parts in hardware which can implement difficult semantics or speed up otherwise slow computations, but automatic separation into hardware and software is not trivial, and most tools leave this task to the designer. So far, hybrid systems consisted of a regular processor, either directly modified to support reactivity or used together with a reactive coprocessor (similar to the mathematical coprocessors of the pre-Intel-Pentium era). One of these is the New Zealand University's RePIC [14, 36], an extension of the REFLIX [37].

The real-time and embedded systems workgroup of the University of Kiel took a "reverse" approach: A processor directly supporting reactive control structures was implemented and additionally got regular CPU style arithmetic operations. It is called the Kiel Esterel Processor (KEP) [30]. The KEP aims to be fully capable of retaining Esterel's semantics. In a more recent work [45] the KEP was rewritten in Esterel and became known as the KEPe. The goals were to explore Esterel as a hardware specification language and better maintain the code. The KEPe implements the Esterel kernel language without valued signals (since the KEPe is missing an ALU).

The software specified in Esterel or as SSM somehow has to get on a processor to be executed. The community has seen several software implementations over the years such as simulating automata [12], netlists [18], and control flow graphs [17]. Most approaches target a software simulation, but the RePIC as well as the KEP have an instruction set architecture (ISA) of their own. Both were designed for compilation of Esterel to their respective ISA. While the RePIC requires several processors to be set up for concurrency — making it hard for compilation of more than 2 threads — the KEP can handle concurrency by multithreading. In 2008, the New Zealand group responsible for the RePIC introduced their improved concurrency support with the StarPro [48] processor.

The main difficulty caused by the implementation is the aforementioned concurrency. Since processors generally can only perform computation steps sequentially (unless working with multicore CPUs, which produce other difficulties), true parallelism is hard to implement. A further issue is that the semantics of reactive systems allow instantaneous feedback and parallelism causing data and preemption dependencies which form the data dependency graph (DDG) and the control flow graph (CFG), respectively. Thread-spanning dependencies impose further restrictions on the sequence in which the reaction's instructions have to be executed. Boldt wrote an Esterel compiler for a synchronous reactive processor (in this case meaning the KEP, see [29]) in 2007, which translated the Esterel language to the KEP ISA.

Even though the KEP is also not capable of true parallelism, with the help of the compiler it can be simulated well enough. Thus, in Boldt's compiler, the parallelism is sequentialized while adhering to dependencies by using the KEP thread priority feature.

1.3. Topic of this thesis — compiling SSMs to the KEP

This thesis deals with compiling SSM to the KEP ISA by the state machine to KEP compiler (smakc!). Theoretically, this is already possible in a sequenced approach. André [6] described how to translate state machines into Esterel, and a modified technique is used in Esterel Studio [44]. From there, Boldt's compiler could be used.

But if we look at state machines and processors closely, we find some key similarities (which in fact are not coincidental, but an effect of the evolution of automata and processors at the same time). The capability of a processor to skip program code (called "branching") is usually referred to as the infamous `goto` statement which has been subjected to a lot of discussion (for some opinions, see [28, 16]). This statement is provided with the address of some other part of the code, at which point the processor immediately continues execution. At processor level a necessity, it has been banned from programming language level by almost all languages by now due to its incomprehensible side effects. But if we look at the drawing of a state machine, we see states — usually depicted as circles, boxes or polygons, and transitions — depicted as arrows. Throughout human history, the arrow symbol and the arrow itself as a missile always had clear, intuitive semantics: from here to there. smakc! honors this heritage in the attempt to generate efficient machine ISA code from state machines.

smakc!ing state machines to the KEP involves several important subtasks.

The first of these tasks is a preprocessing step before compiling: As a state machine implements the perfect synchrony (meaning that in theory for a computation to take time, this has to be explicitly specified), it has to be checked for causality problems. Because the smakc! compiler is implemented as a series of transformations of a state machine, this step was left out for implementation at a later point.

Next, all parallelism has to be sequentialized (sometimes also called linearized) due to the limitation that processors are either not capable of parallel computation or, if so, do not implement the perfect synchrony. This step requires generation of the DDG which reflects which code part must be executed before another one to retain semantics. Further details on this topic will be discussed in chapter 5. Analyzing the DDG allows us to decide if the program is sequentializeable. Cycles in the DDG have proved to have the most influence on this decision since a cycle means that a code part has to be executed before itself. In some cases, cycles can be resolved or broken. This scheduling is one of the main parts of this work.

After causality checking and scheduling, the code for the target architecture can be generated. Some compilers rely on major restructuring work to sequentialize while preserving semantics as well as dependencies [49]. Others use an indirect compilation approach by simulating the state machine [19]. We will analyze different strategies which are not based on interpretation and will introduce computation strategies borrowed from the field of approximative algorithms. This allows a tradeoff between increased compiler speed and suboptimal resource usage, such as the KEP thread priorities which allow fewer programs to run simultaneously the higher the maximum priority is.

1.4. Compilation of Esterel and SSMs in general

As some reactive processors and multiple compilers already exist, others obviously had to deal with the problems of how to handle reactive semantics, scheduling or sequencing, and cycle detection and handling. The first approaches at compiling synchronous languages came from the inventor of Esterel himself. Berry generated C code from Esterel programs by producing automata [12], later netlists [18]. The problems with these methods quickly became obvious: The automata code was extremely large, and the netlist code, although not as large as automata, was much too slow. Stephen A. Edwards can be seen as the father of modern Esterel compilation, as he introduced the control flow graph method. His influence in this field can be seen in several works [17, 19, 18]. Edward's approaches to compiling Esterel are in general based on an interpretation/simulation principle; the original code is simulated to a certain extent, and the simulation results are mapped to sequential code [19]. Other authors borrowed from or implemented his ideas. To name just a few: the Saxo-RT [15] and the CEC [17] make use of Edward's ideas in compiling concurrent to sequential code.

One would think that the availability of multiple CPUs makes it easier to implement parallel semantics (in fact an approach taken by the RePIC project [14]) but the communication times and synchronization between multiple processors have to be considered when implementing the parallel semantics of reactive languages with instantaneous signal broadcast. This is one of the reasons why the RePIC has only two [14] processors. More recent versions have overcome this limitation although the creators of the RePIC or Emperor processor clearly identify the parallel semantics and signal broadcasting as the major source of problems [47, 48].

1.5. Scheduling and sequencing

Difficulties naturally arise when trying to execute a program explicitly defined as having parallel tasks on a single sequential processor. Operating systems simulate this parallelism by concurrent threads of execution and a short time for switching between them. Although the task of scheduling multiple threads or processes among a single CPU or multiple CPUs is already quite challenging, parallel semantics and data dependencies pose additional problems, as do hardware interrupts in conventional operating systems. While some systems try to make disruptions of control flow by preemptions assessable through several methods such as limiting time for interrupts

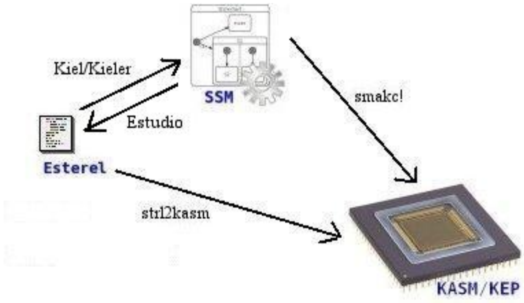


Figure 1.3.: Many paths lead to the KEP.

(for example QNX, see [41]), others treat interrupts as regular inputs which have to be considered in program design too.

This is why parallel tasks are usually “linearized” adhering to the data dependencies. This problem is a variant of the scheduling problem with precedence constraints. This will be discussed in more detail in section 6.2.

1.6. Data dependency and cycle detection

The problem of cycle detection for the DDG is also related to other problems. We will use a similar variant of the dependency detection algorithm from Boldt’s compiler [29] for use with state machines. It generates a DDG which we will search for cycles. A cycle is also a strongly connected component (SCC), which provides us with the possibility of applying SCC algorithms. We could use this for determining the tick length for the KEP worst case reaction time (WCRT) self-monitoring feature too, but we will also analyze a different approach using a modified Floyd-Warshall algorithm.

1.7. Contribution of this work

> That’s hot!™
>
> *Paris Hilton*

This work details the state machine to KEP compiler, or smakc! for short.

André described a way to translate SSMs to Esterel in [6]. The Kiel Integrated Environment for Layout (KIEL) [4] project and its follow-up, the KIEL for the Eclipse rich client platform (KIELER) project [3], both originally tools for design and layout of state machines, synthesize SSMs from Esterel programs (described in [33]). From Esterel, Boldt’s compiler [29] can transform directly to the KEP ISA. smakc! completes the picture by adding the possibility to transform state machines to the KEP, see Figure 1.3.

Multiple approaches exist to accomplish this task. The first method that comes to the mind is using the existing synthesis from SSMs to Esterel and then using

Boldt's compiler. However, to efficiently transform arbitrary state machines, we must encode jumps, represented by `goto` statements. As a state of the art high level language, Esterel does not even have such a statement. In 2007, Edwards and Tardieu presented a way to accomplish instantaneous jumps in Esterel (commonly referred to as "Esterel+Goto", see [43]), building upon previous work about non-instantaneous jumps [42] by extending Esterel traps (an exception-handling mechanism). This introduces new rules, and exchanges some rules in the Esterel formal calculus. On the implementation side, as the KEPe [45] had to be used, this posed a serious problem. The `trap` mechanism had been dropped from the KEP3 to the KEPe, since they can be transformed into equivalent `abort`-based programs. However, the transformation is not trivial. Thus, for transforming state machines to KEP assembler language (KASM), the Esterel language would have to be modified, rendering all Esterel tools currently in use obsolete. Boldt's compiler would have to be heavily modified to support the new mechanisms in traps and to translate them to aborts accordingly. Therefore, we opted to directly transform SSMs to KASM.

Several questions arise: Given semantically equivalent Esterel code and a SSM, which one is more efficient with regard to code size, resource usage, compilation speed when compiled to a processor? Is transforming Esterel to a SSM and then compiling the SSM feasible at all? This work will try to answer these questions.

smakc! is a fast, modular and extendable compiler algorithm of runtime $O(n \log n)$, or $O(n^3)$ in case the Floyd-Warshall algorithm is used. It is capable of handling black boxes, and is also parallelizable. Although the latter is only beneficial in terms of the big-O-Notation if the number of machines is at least logarithmic in the input size, a constant number of machines provides a significant speedup in practice.

To our best knowledge, so far there has been no attempt to compile and sequentialize perfectly synchronous systems such as SSMs or Esterel using the above-mentioned methods.

1.8. Outline

The main part of this thesis starts out with a step-by-step introduction to Esterel and SSMs, two synchronous languages. In section 2.3 we will explore the target architecture, the KEP, in detail. Next, the smakc! compiler will be introduced in a brief overview, followed by a discussion of the abstract implementation considerations and issues in chapter 4, chapter 5 and chapter 6. Since scheduling is a major issue, it will be discussed in more detail. In chapter 7, the details of smakc!s implementation will be laid out. The results are discussed in chapter 8. The thesis will conclude in section 9.1. The smakc! command line interface and its API will be detailed in the user's guide and developer's guide, respectively.

2. Synchronous reactive languages and reactive processing

We will now take a more detailed tour through Esterel and SSMs. These two "programming languages" were both created to implement the "perfect synchrony" in reactive systems. The perfect synchrony requires that the outputs of a process, generated for a given set of inputs, must occur instantaneously when the inputs are read. In physical reality, this is of course not possible, but the assumption allows considering logical correctness and timing constraints separately. To fulfill real-life timing constraints, the longest tick is determined. The hardware can then be clocked accordingly. Since outputs must be generated in a given amount of time[1] and the system has to be deterministic, the system must react in some way to any input, at any time. As an example, think of the airbag controller chip in your car: It should continuously check sensor inputs to determine if the car crashed. The calculation's outcome had better be on time if you don't want black and blue marks on your forehead, and it had better never stop being on time.

To this end, Esterel was developed. Esterel's control constructs enforce reactivity, for example, by restricting loops to a finite number of iterations per tick while still allowing preemption and parallelism, all the while retaining a deterministic system.

The same goes for SSMs and, in fact, the two languages are equivalent in expressiveness. We will take a closer look at both of them starting with Esterel, and we will compare the two using a well-known example called ABRO, a program with the following semantics:

ABRO: The system has boolean valued inputs A, B, R, and an output O. Output O shall be true as soon as both inputs A and B have been true. This behavior should be restarted if R is true.

The program ABRO was inspired by a memory controller which has to wait for an address (A) and the data (B) to become available. Then the write operation (O) can take place. At any time, the controller can be reset (R), effectively also preempting a write operation in the same instant.

2.1. Esterel

Esterel is a textual programming language in the imperative style. Its kernel language consists of some constructs which are already known from other programming

[1] Recall: A reactive system is a system whose correctness also depends on the timeliness of its calculations.

languages:

`nothing`	does exactly that
`;`	the sequence operator
`present S then p1 else p2 end`	an if-then-else statement
`emit S`	emits a so-called “signal”, it is considered to be “present” for the entire tick (signal status is “absent” by default)
`loop p end`	an infinite loop, but must contain a pause statement
`signal S in p end`	defines a local signal
`suspend p when S end`	process suspension as long as the signal S is present
`trap S in p end`	an exception handler block, somewhat similar to Java try-catch
`exit S`	raises the exception S

Here, similarities end. Esterel supports a set of operators with special capabilities which are not easily reproduced in other languages:

`pause`	explicitly let some time pass (wait until next tick), this is the only way to do this
`||`	parallel operator, code to the left and to the right of this operator executes synchronously parallel. The parallel statement terminates when all branches have terminated.

An `exit S` statement immediately exits the scope of the corresponding trap. The above statements make up a kernel language of Esterel (for a full specification of the Esterel language, see [10]). Our program ABRO uses derived statements, which can be seen as macros made up of kernel statements:

`loop p each R`	The program is executed and the program terminates at the end. It is restarted though whenever signal R is present.
`await S`	waits until the signal S becomes present, then continues

`await S` can be written in kernel statements as:

```
trap T in
  loop
    pause;
    present S then
```

```
      exit T
    end
  end
end
```

We want to present one more non-kernel statement, the `abort` statement, because we will be talking about aborting programs or state machines as a form of preemption very often:

`abort p when S`	Aborts program p when S becomes present and immediately starts executing the following code. Since signals are present or absent for the entire tick, if S is present, program p will not even be executed in parts.

In Esterel the program ABRO looks like this:

```
module ABRO:
  input A,B,R;
  output O;
  loop
    [await A || await B];
    emit O
  each R
end module
```

Apart from module name definition and declaration of inputs and outputs, the program consists mainly of a loop which is restarted instantly whenever signal R occurs. In the loop, a parallel simultaneously waits for inputs A and B. Once both parallel branches have terminated (which is the case when both A and B were present at least once, not necessarily in the same tick), the entire parallel statement terminates, and the following statement makes O present.

2.2. Safe State Machines

SSMs are a graphical programming language which seem to be different from Esterel code at first glance. They are an extension of the FSM, containing states and transitions just like a FSM, but additionally hierarchical states, preemption and parallel semantics. Transitions leaving a state are considered abortions, meaning that control currently in the state, possibly executing code associated with that state, is aborted and resumes at the start of the transition's target state.

A state can encapsulate other states which in turn form state machines again (and is then called "macrostate", "compound", "composite" or "complex state"). Those state machines are separated from each other graphically by a dashed line to indicate they are supposed to execute in parallel. In each of these parallel substatemachines, exactly one state must be flagged as initial to indicate the start of control flow.

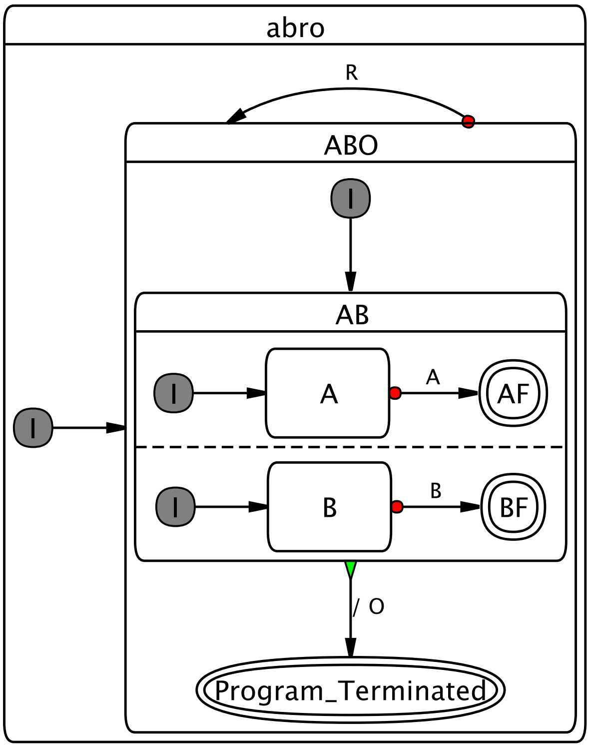


Figure 2.1.: The ABRO program as SSM.

Abort transitions leaving a macrostate preempt control flow in all contained substatemachines at once. But macrostates can also have a different type of outbound transition: the normal termination. As states can be flagged initial, they can also be flagged as final. If control flow reaches a final state, it cannot leave that state anymore. If however control flow of all parallel substatemachines resides in a final state, the containing macrostate is exited by a normal termination (if there is one). Normal terminations do not have conditions.

The ABRO program as SSM (Figure 2.1) illustrates the discussed features.

States drawn in other states (such as the AB state) display hierarchical containment. The dashed line indicates that the states A and B should be executed in parallel. The same goes for AF and BF. The small red dots at the origin of the transition are drawn for strong abort transitions (see below). A green triangle is drawn for normal terminations. Transition labels are of the form "<count|#><trigger>/ <effect>". The trigger is a signal combination that must evaluate to true for a transition to be enabled for the tick. The effect is simply a list of signals that are emitted when the transition is taken. Any of the label parts can be left out. The # sign will be explained later, trigger counts are not important for this discussion. The double border states are final states.

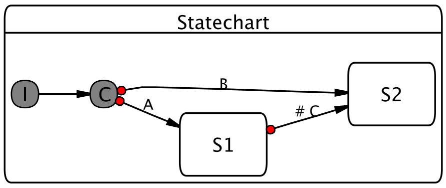


Figure 2.2.: Initial and conditional pseudostates and an immediate transition.

2.2.1. Pseudostates and initial transitions

Figure 2.1 depicts small black dots next to some of the states, with arrows pointing to the states. Figure 2.3, rendered by KIEL, displays those items more clearly as a special type of state: initial pseudostates. They belong to a family of special states called pseudostates. We will deal with two types of pseudostates: the initial and the conditional pseudostate. The initial pseudostate denotes the entry point for control flow in a state machine, and serves as a conditional pseudostate at the same time. Conditional pseudostates have been introduced for the "write things once" principle, basically replacing signal tests of the form `(A and B) or (A and C)` with `(A and (B or C))` (see Figure 2.2). Control flow is not allowed to rest at a pseudostate, it must be immediately transferred to another state. Regular transitions transfer control only after one tick has passed. The behavior of the transitions leaving pseudostates can also be applied to regular transitions by flagging them immediate with a "#" symbol, see the transition from S1 to S2 in Figure 2.2. We will refer to non-pseudo states as real states.

2.2.2. Strong vs. weak abortion and causality analysis

So far we did not specify when exactly a preemption takes place. In Esterel as in SSMs, we distinguish two types of abort transitions: the strong and the weak abortions. A strong abortion preempts a state at the start of a tick, forbidding any execution of code within the state. A weak abortion on the other hand preempts the state at the end of a tick, allowing the state to execute one more time. This might not seem significant at first but consider Figure 2.3.

The state $S1$ is strongly aborted at the occurrence of signal A, preventing any execution of code within. Now consider the instant control flow enters state *strong*. Control immediately branches to the initial state, and since it is a pseudostate, the only transition (to $S1$) is taken, and A is emitted. But signals are always broadcasted, and due to synchronous reactivity semantics, the signal status is present for the entire tick. This triggers the strong abortion though, which preempts state $S1$ at the start of the tick, preventing any execution of inner states including taking the transition from the initial state to $S3$.

To make this example clear: The emission of A prevents the emission of A, which enables the emission of A again, a causality problem. Such programs are not valid and must be rejected by compilers.

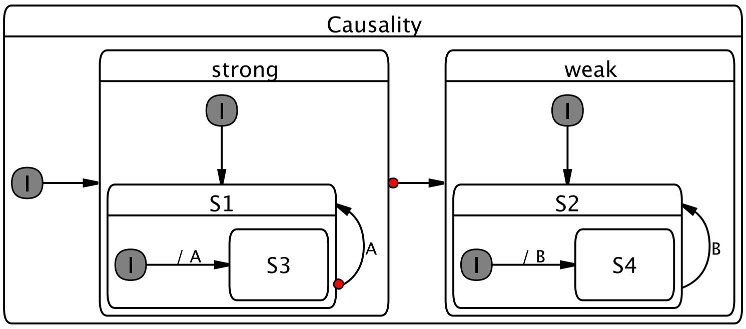


Figure 2.3.: Left: a causality problem.

The state to the right, using a weak abortion instead, is perfectly valid, since weak abortion allows a state to execute for another tick before preempting at the end of the tick.

2.2.3. Actions

Testing and emitting signals does not necessarily only have to take place when taking transitions. So called "actions" allow doing this also when entering, exiting or resting in a state. Any real state can have actions assigned as onEntry, onExit and onInside action. An action can have a trigger (a complex signal expression) and an effect, which is the emission of signals. When entering a state, its onEntry action's triggers are evaluated. If the expressions evaluate to true, the actions' signals are emitted. Whenever control flow rests in a state, the onInside actions' triggers are evaluated and signals are emitted if they evaluate to true. onExit actions are not quite analogous. Their effect signals are also emitted when their triggers evaluate to true, and obviously, they are tested when exiting a state. But exiting a state not only applies to exiting it by a transition. The state can also be exited by preemption of a containing macrostate at some arbitrary point in the containment hierarchy. The translation of onExit actions to KEP code is possible, but difficult, and not implemented in smakc!. We provide some ideas on how to do this in section 9.2. In fact, onExit actions were even difficult to express in Esterel. Therefore the current version of Esterel was augmented with additional language features to express the semantics of onExit actions.

2.3. On the KEP and the KEPe

The KEP was developed by the reactive and embedded systems group of the Department of Computer Science at the University of Kiel. Its purpose is to enable direct implementation of the perfect synchrony found in Esterel or SSMs by means of supplying special hardware features. Implementing perfect synchrony along with parallelism and preemption while still retaining a deterministic system is a very diffi-

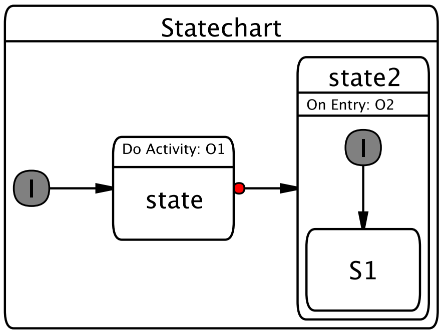


Figure 2.4.: onEntry and onInside actions rendered in KIEL.

cult task with regular hardware, since external hardware as well as the system clock raise interrupts, thereby making it impossible to exactly predict the system's behavior. The KEP is a special processor which circumvents these problems (although some work still has to be done by the compiler). For full information on the KEP, see [30].
The KEPe is a more recent development, an implementation of the KEP ISA kernel statements (valued signals as well as an ALU have been left out) written in Esterel. One of the questions was if the KEP could execute itself. Unfortunately, the question could not be answered due to Esterel version differences. For full information on the KEPe, see [29] and http://www.informatik.uni-kiel.de/rtsys/kep.
We will now take a short look at the KEPe commands and their significance regarding SSMs. Since the KEPe implements a subselection of the KEP instructions, the following considerations are valid for both processors.

The modern KEP family processors all make use of "watchers" which are divided into abort watchers and thread watchers.

Abort watchers get a start address, an end address and a signal. If the signal becomes present and control flow currently resides in the code block designated by the start and end address, control is immediately transferred to the end address.

The multithreading feature first introduced in the KEP3 is more difficult to explain. `PAR` statements define start addresses for parallel blocks of code, additionally defining initial thread priority of the threads. A `PARE` statement denotes the end of the last parallel block. For each parallel block, a thread watcher is initialized. The thread watchers have a similar task as the abort watchers: they watch control flow within their respective thread. If control flow leaves the thread's code segment, the thread watcher terminates the thread. If all parallel threads terminate, control flow is resumed in the parent thread at the address designated in the `PARE` statement. The processor always runs the thread with the highest priority, therefore, to get interleaved execution, the thread priorities have to be modified during runtime. This introduces the main scheduling difficulty: A thread can only change its own priority.

The KEPe ISA consists of the following instructions:

`emit S`	emits the signal S (making it present)
`sustain S`	control flow stops at this instruction and continuously emits the signal S
`load reg, #val`	loads a value into the specified register (can also be a signal)
`present S elseaddr`	checks if S is present. If so, continues with the next instruction, otherwise branches to elseaddr
`[w]abort[i] S, endaddr`	branches to end of code block designated by endaddr when S becomes present, [w] is a modifier for weak and [i] a modifier for immediate abortion
`suspend[i] S, endaddr`	suspends (freezes its execution) the code block designated by endaddr as long as signal S is present
`par addr, prio`	opens a new thread block with code segment up until addr, the new thread starts out with priority prio
`pare addr`	closes the last par block at addr
`goto addr`	immediately branches to addr
`signal name`	opens a new signal scope. This is simply done by resetting the signal if it previously existed
`await[i] S`	control flow stops at this instruction until the signal S becomes present
`join prio`	joins terminated parallel threads and assigns the parent thread the argument priority
`nothing`	does exactly that
`halt`	control flow stops at this node

All modern KEP versions are scalable. Upon generating a new KEP microprocessor, the number of abort and thread watchers, the size of the instruction memory and the maximum amount of i/o signals can be specified, for example.

3. smakc!ing state machines to the KEP — the big picture

> The White Rabbit put on his spectacles. “Where shall I begin, please your Majesty?” he asked.
> “Begin at the beginning,” the King said gravely, “and go on till you come to the end: then stop.”
>
> *from “Alice’s Adventures in Wonderland”*

In general, the compiler is provided with a set of input state machines and an ordered list of transformations to apply to the state machines in that order. The compiler therefore is quite simple to understand, as can be seen in Figure 3.1.

The compiler algorithm is composed of several sequential steps. In some of these steps, more than one option exists to perform it. Some steps can be left out entirely in some cases. The subalgorithms are:

1. A conditionals unrolling which transforms complex signal expressions to simple signal tests. This algorithm is detailed in section 4.2. It leaves SSMs that do not contain conditional expressions unchanged and can therefore be left out if the input state machines are known not to have such expressions.

2. A dependency detection which detects data and control flow dependencies and adds them to the SSM as special transition type. This transformation can be left out if the input state machines are known not to have any dependencies or if no scheduling is wanted.

3. A cycle detection algorithm. The algorithm operates on the data dependencies found in a state machine and is implemented by the Floyd-Warshall algorithm. It stops the compiler with an error if a cycle in the data dependencies was found in any input SSM, or continues with the cycle-free SSMs, depending on the value of the *force* compiler flag. This transformation can be left out, since the scheduling algorithm automatically detects cycles. However, it could be useful to run it anyway, as the cycle detection also finds out which states lie on a dependency cycle.

4. A transformation that upgrades all states to states with thread priorities. This is a true extension of the SSM model, since thread priorities are only needed for sequencing and are not required in synchronous languages. This transformation is mandatory if the scheduler or code generator are supposed to be used.

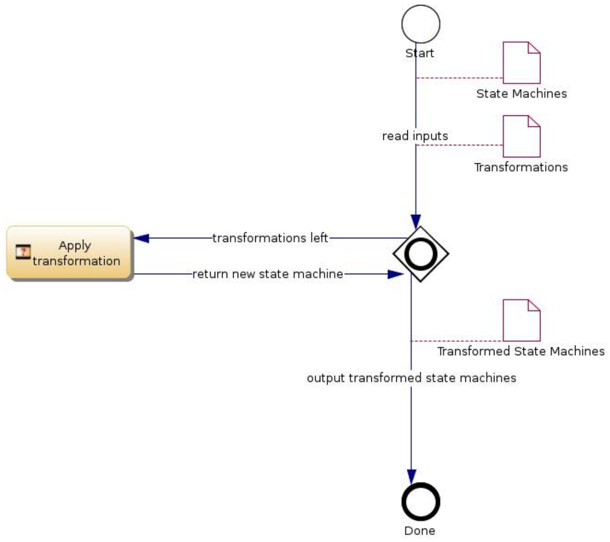


Figure 3.1.: The smakc! procedure in general.

5. A module linker (not yet implemented). This transformation links all encountered reference macrostates to the corresponding input state machines and performs signal renaming on them.

6. A scheduler. This algorithm requires states with thread priorities as input. It schedules the states according to the encountered dependencies and assigns thread priorities accordingly. The algorithm is implemented using ideas from strip packing.

7. A code writer. This transformation also requires states with thread priorities and generates the target platform's code. This transformation uses the Apache Velocity templating engine [35].

A full run would therefore consist of the steps depicted in Figure 3.2.

We will illustrate the full run by smakc!ing the SSM in Figure 3.3.

As noted in the overview of the transformations, the conditional resolving transformation is applied first. Since our SSM does not contain any signal expressions it is unchanged by that transformation. Next, the dependency detection is started. It will detect a dependency from $T1$ to $U1$ by signal A, and another dependency from $U1$ to $T2$ by signal B. These are shown in red in Figure 3.4. Since these do not form a cyclic dependency, the cycle detection transformation has nothing to complain about. The states have to be upgraded to states with thread priorities before they can be scheduled. The scheduling result is shown as the red numbers next to the state names in Figure 3.4.

Finally, after scheduling, code can be generated by the code writer transformation. It does not change the SSM, it only writes the target architeture code to an output stream. The generated code for our sample SSM is shown from 19 to 22.

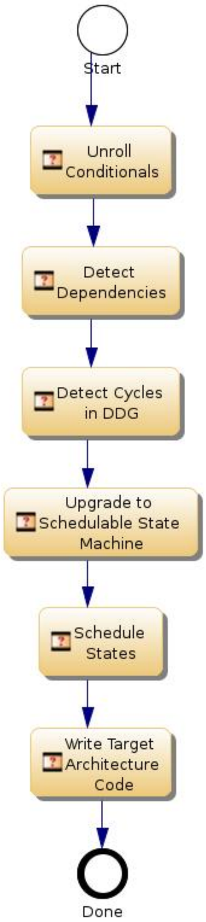


Figure 3.2.: Being smakc!ed from the SSM's point of view.

```
INPUT A
INPUT B
EMIT _TICKLEN,#0

BEGINSTARTUPTWODEPENDENCIES:
ENDSTARTUPTWODEPENDENCIES:
BEGINCOMPLEXSTATETWODEPENDENCIES:

  BEGINSTARTUPS:
  ENDSTARTUPS:
  BEGINCOMPLEXSTATES:
    PAR 3, BEGINSTARTUPT1, 1
    PAR 2, BEGINSTARTUPU1, 2
    PARE SUBSTATESENDS, 0

      BEGINSTARTUPT1:
      ENDSTARTUPT1:
      BEGINAWAITSTATET1:
```

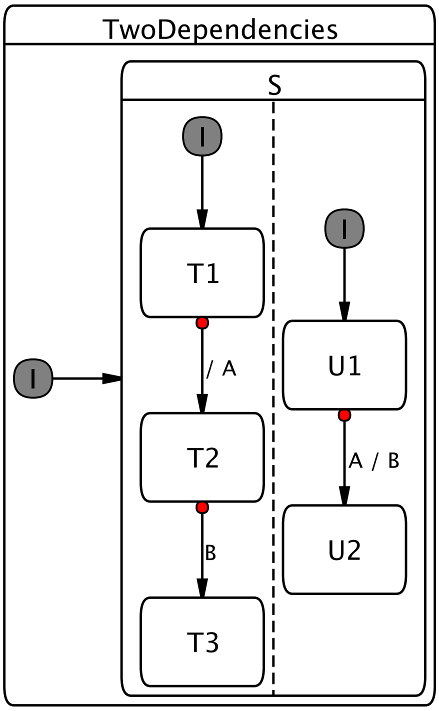


Figure 3.3.: Our example SSM "TwoDependencies".

```
    AWAIT TICK
    GOTO BEGINSTARTUPT2
  ENDAWAITSTATET1:
  BEGINSHUTDOWNT1:
  ENDSHUTDOWNT1:

  BEGINSTARTUPT2:
  ENDSTARTUPT2:
  BEGINAWAITSTATET2:
    AWAIT B
    GOTO BEGINSTARTUPT3
  ENDAWAITSTATET2:
  BEGINSHUTDOWNT2:
  ENDSHUTDOWNT2:

  BEGINSTARTUPT3:
  ENDSTARTUPT3:
  BEGINSIMPLESTATET3:
    HALT
  ENDSIMPLESTATET3:
  BEGINSHUTDOWNT3:
  ENDSUSPENDT3:
  ENDSHUTDOWNT3:

  BEGINSTARTUPU1:
  ENDSTARTUPU1:
```

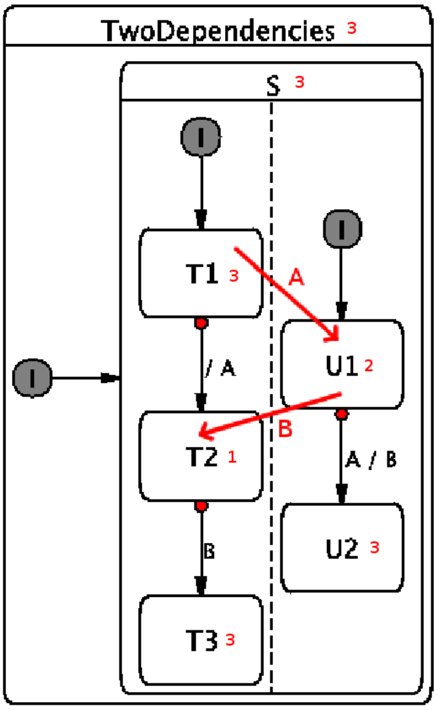


Figure 3.4.: Our SSM with dependencies shown in red, already upgraded with thread priorities and scheduled.

```
        BEGINAWAITSTATEU1:
          AWAIT A
          GOTO BEGINSTARTUPU2
        ENDAWAITSTATEU1:
        BEGINSHUTDOWNU1:
        ENDSHUTDOWNU1:

        BEGINSTARTUPU2:
        ENDSTARTUPU2:
        BEGINSIMPLESTATEU2:
          HALT
        ENDSIMPLESTATEU2:
        BEGINSHUTDOWNU2:
        ENDSUSPENDU2:
        ENDSHUTDOWNU2:

      SUBSTATESENDS:
      JOIN 3
      HALT
    ENDCOMPLEXSTATES:
    BEGINSHUTDOWNS:
    ENDSUSPENDS:
    ENDSHUTDOWNS:

SUBSTATESENDTWODEPENDENCIES:
```

```
HALT
ENDCOMPLEXSTATETWODEPENDENCIES:
BEGINSHUTDOWNTWODEPENDENCIES:
ENDSUSPENDTWODEPENDENCIES:
ENDSHUTDOWNTWODEPENDENCIES:
```

The exact details of the transformations are discussed in the following chapters.

Except for the Floyd-Warshall cycle detection algorithm, all transformations are implemented by depth first or breadth first search and thus are $\mathcal{O}(n \log n)$ algorithms. The Floyd-Warshall algorithm has a runtime of $\mathcal{O}(n^3)$ and dominates runtime as well as memory usage, since it requires setting up a matrix of size n^2.

An advantage of the strip packing algorithm is that the algorithm can automatically detect cycles in the DDG since the strip packing steps are bounded from above by a value in $\mathcal{O}(n)$, although by this method we do not know exactly which states cause the cyclic dependency.

Once the DDG has been set up, most of the remaining compilation tasks can be split into subtasks which allow parallel execution.

4. Basic transformation issues

4.1. Semantics, sequencing and dependencies

The most difficult problem in executing a SSM is the instantaneous broadcast of signals. The testing of a signal at a transition must either succeed or fail, dependent on the presence status of the signal in question, but not on the time that presence status is established. This concept, easily expressed in theory, is very difficult to implement, since processor instructions are generally executed sequentially, meaning that of a signal testing and a signal emitting statement, one has to go first, and preferably it should be the emission, since it is a statement that establishes a signal status. This creates implicit dependencies between code parts that were not explicitly created in the code when moving from a SSM to a processor. Therefore, the statements have to be scheduled to ensure the correct order of their execution. This big topic will be discussed in chapter 6.

4.2. Complex conditional expressions

Just Dropped In To See What Condition My Condition Was In

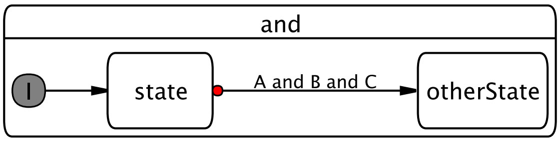
song from the movie "The Big Lebowsky"

Transforming an abstract representation of a system to an assembler language naturally also requires some minor restructuring tasks even though the target processor architecture already supports most statements directly. One of these tasks is breaking down boolean formulas (e.g. $((A\ and\ B)\ or\ C)$) to sequenced testing of single variables. In the case of SSMs this can be done with the common simple trick of introducing additional variables which are always recalculated to be equal to the result of a complex formula, and then testing that variable.

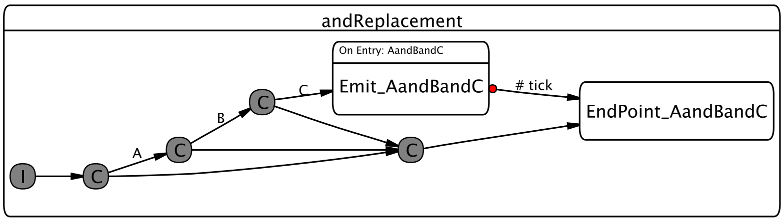
Instead of exchanging every complex condition test with the corresponding state machine in that place, we have opted to introduce a new state machine of its own in a parallel branch that calculates the signal expressions contained in its direct parent complex state. This way we save states in case the same signal expression appears more than once. On the other hand, we have to make sure the additional parallel state machine does not prevent the parent state from terminating. Imagine all original parallel branches are in a final state. Then the signal calculation state machine also has to enter a final state to allow the parent state to take a normal termination.

The signal calculation state machine consists of a combination of the state machines that resolve simple expressions. We will illustrate this for the boolean operators *and*, *or* and *not* using only literals.

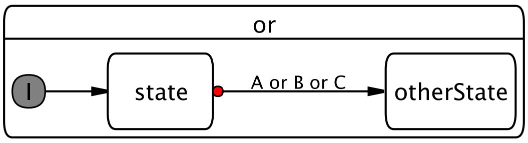
An *and* expression



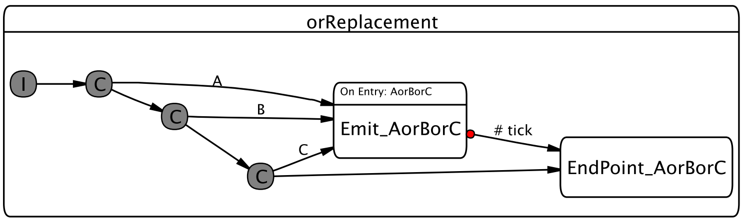
is transformed to



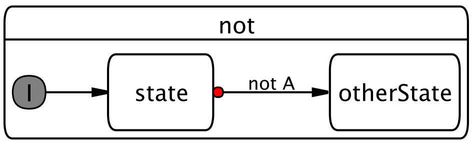
whereby the first of the signal calculation states is connected to the initial state of the signal computation thread, or to the previous endpoint(s). An *or* expression



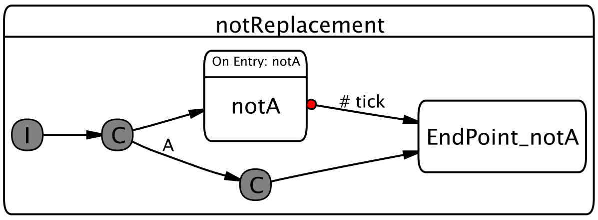
is translated to



And a *not* expression

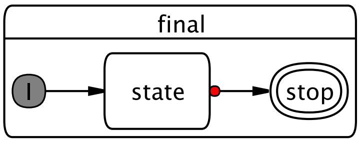


produces the following state machine:

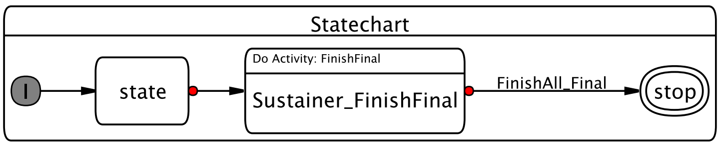


All of the states in the signal computation thread — except the end points which have to emit signals — are represented as conditional pseudostates to save resources. In the above examples, *AandBandC*, *AorBorC* and *notA* are new signals. We already mentioned that some more transformation has to be done to enable regular termination of a state with parallel threads. Think of a macrostate that has a normal termination and each thread contains a final state. Then all threads can terminate by reaching their final state, and the normal termination transition is taken. If we add another thread we have to ensure that the new state machine retains the same behavior. This requires some internal signaling. The original threads must communicate to the signal computation thread that they are finished to let it know that it can terminate too. On the other hand, due to the definition of signals being absent by default, they have to communicate this to the signal computation thread as long as there is still an original thread running.

To implement this, we used the following idea: Each thread gets a new local signal (local to the containing macrostate) representing the terminated or not terminated status. Each final state

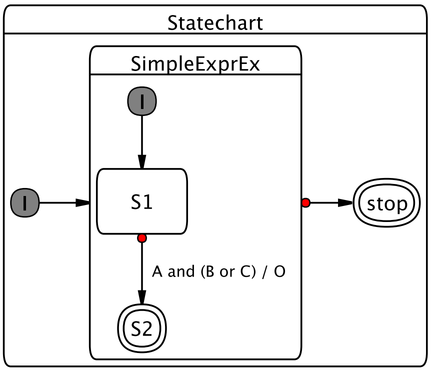


is exchanged for a fake final state, in which a signal is repeatedly emitted that the thread is done:

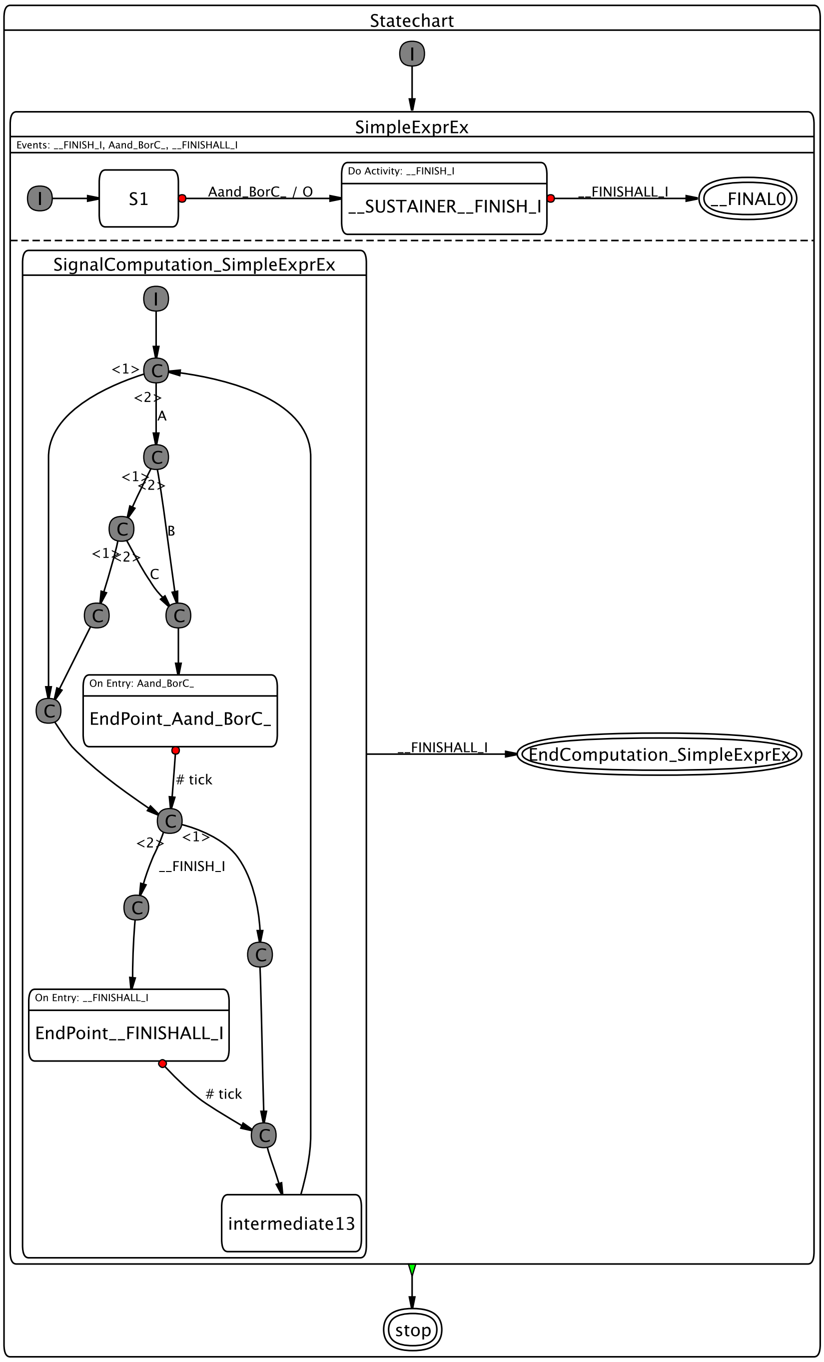


The conjunction of these final signals is tested just like a regular *and* conditional expression in the signal computation thread. That expression's end point emits the *FinishAll* signal, by which it terminates itself, and signals to the other waiting threads that they can stop emitting the *finish* signal and continue on to their respective final states.

The entire process is illustrated in the following example:



The conditionals unrolling results in a modified state machine:



Statechart
SimpleExprEx
Events: __FINISH_I, Aand_BorC_, __FINISHALL_I
S1
Aand_BorC_ / O
Do Activity: __FINISH_I
__SUSTAINER__FINISH_I
__FINISHALL_I
__FINAL0
SignalComputation_SimpleExprEx
A
B
C
On Entry: Aand_BorC_
EndPoint_Aand_BorC_
tick
__FINISH_I
On Entry: __FINISHALL_I
EndPoint__FINISHALL_I
tick
intermediate13
__FINISHALL_I
EndComputation_SimpleExprEx
stop

5. Analyzing signal dependencies

> Just try and stay out of my way. Just try! I'll get you, my pretty and your little dog too!
>
> *The Wicked Witch of the West*

The signal dependency detection used in smakc! is quite straightforward: Signal sources and signal sinks are identified. Then, for each pair of source and sink, we check if they could possibly form a dependency. The same method was implemented in Boldt's *strl2kasm* compiler [29], although his compiler determines dependencies at assembler code level, whereas smakc! determines them at SSM level. In their work about removing cyclic dependencies [31], Lukoschus and von Hanxleden also determine signal dependencies and remove cycles in programs that are known not to have any causality problems. Their approach can not be applied here, as the scope of the problem is completely different. In compilers, we need to know which states are origins and which states are targets of dependencies. Lukoschus and von Hanxleden are only interested in knowing which signals cause cyclic dependencies, and then remove the dependency by exchanging the signals throughout the entire program. They do not need to know what program parts cause the dependencies. Determining these is the main problem in dependency detection. We will now take a look at how this is done in smakc!.

First, some definitions:

Source: A state is a potential source of a signal S if S is emitted in any of the state's actions (onEntry, onInside, onExit), or if it is emitted on any transition leaving the state.

Sink: A state is a potential dependency sink of signal S if S appears in any expression that is tested on an outbound transition, in an action's trigger, or in a suspension.

Dependency: A source/sink state pair with the same signal form a dependency if the two states are concurrent to each other.

The reason for not considering dependencies of non-concurrent states is that those states are already in a fixed ordering which cannot be changed through scheduling of threads.

Determining possible sources and sinks is a fairly easy task, whereas determining if two states S_1 and S_2 are concurrent to each other requires the following calculations:

Concurrency detection algorithm 1

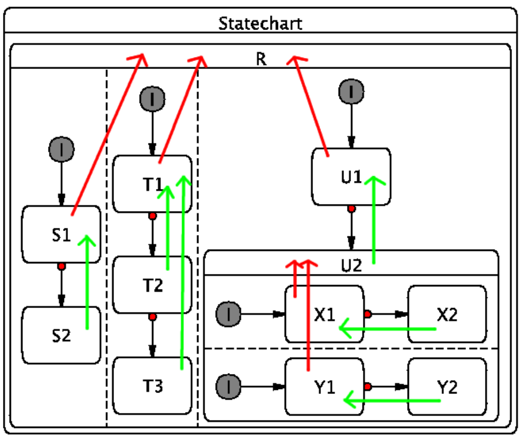
1. For each state, determine the containing parent state.
2. Use information from the previous step to find the topmost state containing S_1 and S_2, called S_{gcd}.
3. For each parallel substatemachine S_p in S_{gcd}, check if S_p contains both S_1 and S_2, in which case they are not concurrent, otherwise they are concurrent.

The dependencies returned by this algorithm are just rough guesses of what could be a dependency, since some constellations of simultaneous active states might not be possible at all in a state machine. As yet, no one is gathering or assessing information on which states can be active at the same time.

The performance tests on the token ring example (section 8.2) showed that the algorithm is fairly inefficient, especially for highly parallel SSMs, since it requires searching through all parallel branches of the *gcd* state. Imagine a SSM that consists of the root state, and some arbitrary amount of parallel branches containing an initial state connected to a final state. In the worst case, the last step of *algorithm 1* requires searching (almost) the entire SSM again. By "remembering" the states traversed during the search for the *gcd* state, this second search can be avoided and the runtime can be significantly improved. The modified algorithm consists of the following steps:

Concurrency detection algorithm 2

Step 1 With a simple DFS, set up a containment graph in which every non-initial node points to the initial node of its thread (green edges), and the initial node points to the containing macrostate (red edges), see the following image for an example:



Step 2 For any two nodes a and b, let a_{active} and b_{active} be boolean variables. Initially both set to false, the algorithm will be allowed to terminate only if they are both set to true (the termination of the algorithm is "activated" by the conjunction of the two variables). Let a_p and b_p be pointers to states, initially pointing to a and b respectively. Keeping node a_p fixed, traverse the containment graph starting at node b_p, remembering the previously traversed node. If a red edge is traversed, set b_{active} to true. If a_p is encountered on the way and (a_{active} and b_{active}) holds true, the current state is the *gcd* state. If the topmost state is reached, reset b_p to b and b_{active} to false, and advance a_p one step up the containment graph, setting a_{active} to true if a red edge is traversed and remembering the previously traversed node, and repeat the behavior for b_p. At the end of this step, the *gcd* state for a and b has been found.

Step 3 All that remains to be done is to compare the states remembered on the way just before encountering the *gcd* state for equality. If they are equal, the states are sequential, otherwise concurrent.

6. Scheduling

Linearizing parallel code is one of the main tasks of the smakc!. Often, we want multiple programs executing simultaneously on one machine. Reactive programs, in most cases used for modeling embedded systems, are inherently parallel. In most cases, there are also dependencies — in this case better known as precedence constraints — in the order of the tasks to be executed. A simple example is a web browser which first has to download an image from a remote computer to be able to display it for you. However, a single core CPU can only execute one program at a time. Various strategies have evolved to handle this problem:

The KEP implements semidynamic scheduling by thread priorities, resulting in the necessity for the compiler to statically schedule programs before they can be executed on the processor.

Linearizing tasks according to some given precedence constraints is a very old problem for which several solution methods exist. But first, we want to state the problem mathematically:

Let $T = \{T_1, \ldots, T_n\}$ be n tasks, and let $G = (T, D)$ be a directed acyclic graph having the tasks as node set and representing precedence constraints. So for any edge $(a, b) \in D$, task T_a should be executed before T_b. The goal is to find a map $\phi : T \to \{0, \ldots, n\}$ such that for $(a, b) \in D$, $T_a\phi > T_b\phi$.

In most cases, we want to minimize $|T\phi|$. The reason for this is quite technical: Due to hardware limitations, there is a maximum priority value. By minimizing the maximum priority used by a program, more programs can be run simultaneously. Additionally, by minimizing the highest priority used, the number of priority switches is also minimized to some extent, saving further processor cycles. Priority switches also require an instruction on the KEP, so each priority switch also adds to the total instruction count. Therefore, $|T\phi|$ also has impact on runtime and size of the generated code.

The KEP self-prioritizing of threads and the implementation of thread priorities imposes further constraints on the ordering of the tasks because, as mentioned in section 2.3, increasing the thread priority is essentially not possible during the same tick, which means that the thread priority can only stay equal or decrease.

In the following two sections, we will review two methods for such constraint-solving. The first is the simplex algorithm, an exact algorithm and one of the oldest among constraint solving algorithms. The second is a more recent algorithm from the class of approximative algorithms, trading accuracy of the result for increased

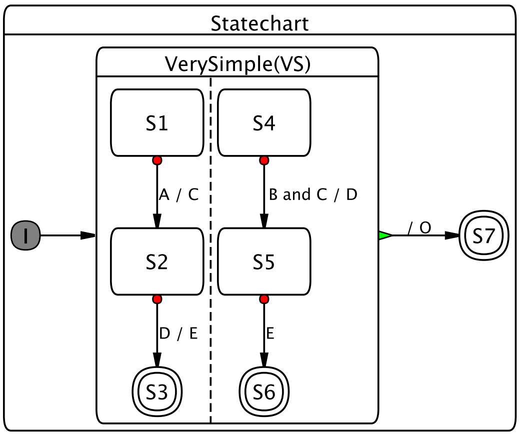


Figure 6.1.: A simple state machine with data dependencies.

speed.

As an example, we will use the state machine from Figure 6.1.

6.1. Using the simplex algorithm

In this section we will show how to apply the simplex algorithm to solve the constrained ordering of the data dependencies.

6.1.1. Linear constraint problems

The simplex algorithm solves linear constraint problems, in mathematical terms defined as the set of vectors $x = (x_1, \ldots, x_n)^T$ which, for a matrix A and a vector v, form the affine space of solutions to the equation $Ax = b$ (standard form), or the inequation $Ax \leq b$ (canonical form), whereby finding the optimal value of the objective function to a cost vector c defined as $\min c^T x$. All matrix and vector dimensions must be fitting, of course.
A linear constraint problem (linear problem (LP)) is usually noted using the following syntax:

$$\min c^T x \textit{ subject to}$$

$$Ax \leq b$$

All variables are required to be integers greater than or equal to zero.
Other forms require strict inequality or equality in the constraints. However, any LP in one form can be translated into another, equivalent LP of any other form.

To phrase the linearization of a state machine as a LP, we have to define the variables, the constraints that make up the constraint matrix and boundary vector, and

the cost vector.
As variables, we use the states of the state machine (referring to them by their name, or in case of conditional pseudostates or other unnamed state types by some previously assigned unique indexes). The cost vector will be the vector set to 1 in each component.
For each dependency from state S_a to S_b, we add the constraint:

$$S_a - S_b > 0$$

and for each path of control flow (that is, a regular transition, a normal termination, or from a complex state to the initial states of its substatemachines) S to S_{suc}, we add the constraint:

$$S - S_{suc} \geq 0$$

Variables will be greater than or equal to zero[1].
Thus, for our example state machine, we get the following LP:

$$\min VS + S_1 + S_2 + S_3 + S_4 + S_5 + S_6 + S_7 \ s.t.$$

$$(1)\ VS - S_1 \geq 0$$

$$(2)\ VS - S_4 \geq 0$$

$$(3)\ VS - S_7 \geq 0$$

$$(4)\ S_1 - S_2 \geq 0$$

$$(5)\ S_2 - S_3 \geq 0$$

$$(6)\ S_4 - S_5 \geq 0$$

$$(7)\ S_5 - S_6 \geq 0$$

$$(8)\ S_1 - S_4 > 0$$

$$(9)\ S_4 - S_2 > 0$$

$$(10)\ S_2 - S_5 > 0$$

The inequalities from (1) to (7) are the inequalities resulting from control flow, while (8) to (10) result from the dependencies. To convert the LP to canonical form we need to introduce new variables to convert the true inequalities into lesser-or-equal inequalities. These additional variables will not be considered in the results.

[1] Note that although this might seem like an integer LP, in fact it is not, since we are only interested in the ordering of the states.

Such variables are called *slack variables*, and as all regular variables must be greater than or equal to zero.

Introducing the slack variables t_1, t_2 and t_3, the true inequalities (8), (9), (10) are converted to:

$$(8')\ S_1 - S_4 - t_1 \geq 0$$

$$(9')\ S_4 - S_2 - t_2 \geq 0$$

$$(10')\ S_2 - S_5 - t_3 \geq 0$$

Note that for any x,

$$x \geq 0 \Leftrightarrow -x \leq 0$$

thus our LP is in canonical form. The solution of the LP is an assignment of values to the variables fulfilling all constraints and minimizing the cost function.

6.1.2. Solving LPs: the simplex algorithm

Research for the simplex algorithm started during World War II and was finished in 1947 by the US mathematician George Dantzig. Since then, it has undergone a lot of modification, extension and testing, for example by Karmakar in 1984 [27] whose research provided the foundation for the polynomial time inner point algorithms. The simplex algorithm searches the vertices of the multidimensional polytope defined by the system of constraints in an ordered fashion, since the optimal solution is one of the vertices. The number of vertices is constant for any given system of equations, so the algorithm runs in polynomial time. However, if we want to solve a series of different LPs, the runtime also depends on the number of vertices, which is exponential in the number of equations.

The linear constraints and cost function of a LP make it easily understandable in geometric terms. Looking only at states S_1 and S_2 of the above example, equation (4) states that

$$S_1 - S_2 \geq 0 \Leftrightarrow S_1 \geq S_2$$

The cost function, reduced to those variables, reads $\min S_1 + S_2$. The situation can be interpreted graphically as can be seen in Figure 6.2.

The inequalities form a convex polyhedron in their vector space (instead called polytope if it is bounded). If we now fix all of the variables of the cost function except one, we get a line in the space formed by the variables. Minimizing the cost function can now be seen as "pushing" that line around in the space until it reaches

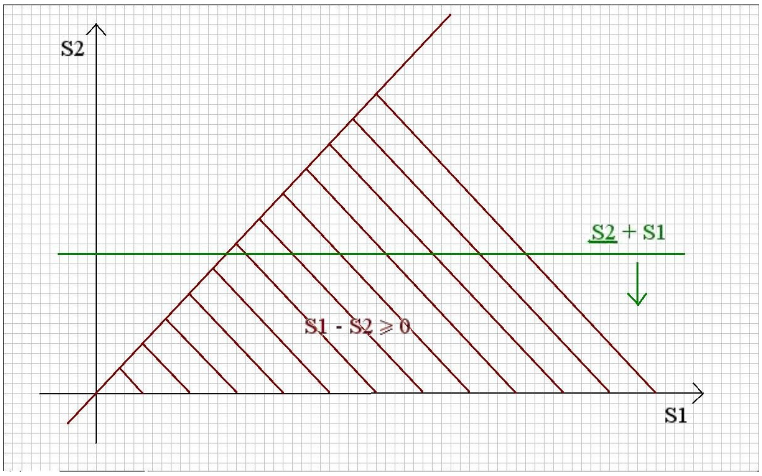


Figure 6.2.: Geometric interpretation of the simplex method, $S1$ fixed, $S2$ variable.

its minimum value.
If you try this experiment by drawing a triangle on a piece of paper and using a pen as line to push around on it, it is intuitively clear that the maximum or minimum values of the cost function can always be found on a vertex of the polyhedron, if there are any at all.
The simplex algorithm exploits this property by starting with one vertex and advancing to an adjacent vertex with a better cost function value. If at some time there is no such vertex, the optimal solution has been found.

6.1.3. Disadvantages of the simplex algorithm

The simplex algorithm has been implemented several times and in several different variations [26]. Such frameworks usually work on standard or canonical form, with the drawback that additional variables have to be introduced. When using frameworks, it is difficult to get usable results, since optimizations to the original variables are often done by pushing the cost to the slack variables which are not subject to the cost function. Thereby, the cost function is optimized, but we lose differences between the original variables expressed in our unmodified equations.

Furthermore, although the simplex algorithm runs in polynomial time in the average case, it can be exponential in the worst case. The ellipsoid method or inner point method circumvent this problem but have much longer runtime in practice than the simplex method.

The introduction of black boxes (states with one or more priority switches) also requires more precalculation in the setup of the LP.

For these reasons, we chose to refrain from implementing the simplex method for scheduling in smakc!.

6.2. Using strip packing

Ideas from strip packing can also be used to schedule the state machine parts. Concepts from strip packing almost naturally apply to linearizing, and handling black boxes becomes very straightforward.

As most strip packing problems are NP-complete, algorithms are usually approximative algorithms, trading accuracy for speed. Strip packing and scheduling are closely related up to the point of being almost equivalent, and often, they borrow from each other.

Scheduling started as far back as 1961 when T.C. Hu analyzed parallel sequencing and assembly line productions [25]. He gave an intuitive algorithm for non-cyclic, input-independent graphs and calculated completion time given some amount m of workers (m stood for men, but in later works evolved to machines), and the amount m of workers needed to complete all tasks in a fixed time. The basic idea of his technique was reused much later in an LP approach to scheduling with communication costs and precedence constraints.

If we take a look at the 80s, we find linearization methods researched by Ferrante and Mace [21, 22]. There is also some more contemporary work on linearization by Zeng et al. [49], although not specifically geared towards reactive systems.

More sophisticated algorithms taking data dependencies into account have appeared only recently, built on top of the now fairly well explored field of strip packing and bin packing (which are related in many ways). The strip packing problem is defined as the problem of packing a set of n-dimensional rectangles into an n-dimensional strip whose “height” (one of the dimensions) is unbounded, with minimal packing height. The sequentialization of tasks with dependencies can be expressed as the precedence constrained strip packing problem, for which an $\mathcal{O}(\log n)$-approximation algorithm was found in [8]. It makes use of a 2D packing algorithm, which can be either Steinberg [40] or Schiermeyer [38].

In our scheduling algorithm, we will borrow several ideas from [8].

6.2.1. Strip packing problems

In strip packing, we are given a container of fixed width and infinite height, and a list of items that all fit into the strip individually (as consequence, they also all fit into the strip together). The problem is to provide an algorithm that packs the items into the strip minimizing the height of the packing.

For simplicity's sake, often the strip base size and item base sizes are normalized such that the size of the strip is 1. For 2-dimensional strip packing, the problem is defined as:

Given a strip of base width 1, and n items I_1 to I_n with arbitrary height h_i and width $w_i \in (0, 1]$, find a packing of the items into the strip (denoted by pairs x_i, y_i specifying the coordinates of the lower left corner of the item in the packing) such

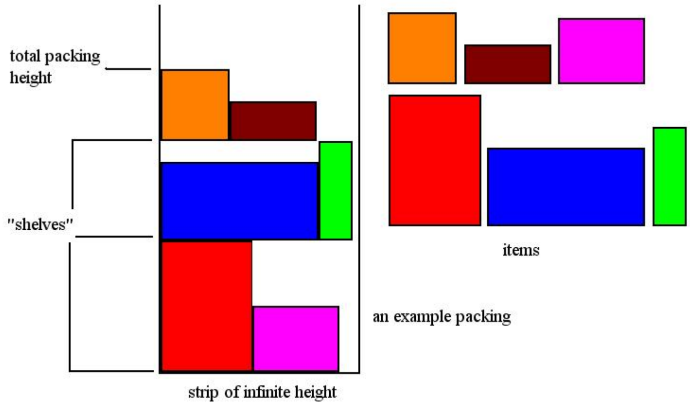


Figure 6.3.: Strip packing with a shelf algorithm.

that the items do not overlap, minimizing the total packing height.

Several methods of solving strip packing problems exist, most algorithms are surprisingly simple, almost trivial. The proofs for their performance however are extremely difficult and often make use of LPs. Most algorithms in this field belong to the class of "shelf algorithms", that is, they pack items onto an imaginary shelf and open up a new shelf on top of the highest item of the current shelf when it is full (see Figure 6.3).

Several variations of the problem exist, for example, with more dimensions, special item types (all squares), allowing rotations for the items or precedence constraints on the placement of the items.

Applying strip packing to state machines is very straightforward: We will use the states as items, and the dependencies as precedence constraints for the placement of the items. Using the states as items has the significant advantage that we can assign a height of 1 to simple states, whereas state machines which have already been scheduled can be reused in a new scheduling by assigning them the total height of their respective packing. This exploits the property that a finished packing can itself be used as item again in a new packing, using the packing height as item height. Then, a valid packing reflects the ordering of the states in such a way that the constraints are fulfilled.

6.2.2. Solving precedence-constrained strip packing for a SSM

So let S be the set of states of a state machine and $G = (S, E)$ the directed acyclic graph representing the precedence constraints. For $s \in S$, let h_s be 1 for regular states, and the height of a strip packing for pre-scheduled state machines. The inbound-neighborhood of a state is defined as:

$$\mathcal{IN}(s) = \{s' \mid (s', s) \in E\}$$

The function F will serve as a lower bound for the height of the top edge for an item under a valid placement:

$$\text{If } \mathcal{IN}(s) = \emptyset \text{ define } F(s) = 0$$

$$\text{If } \mathcal{IN}(s) \neq \emptyset \text{ define } F(s) = \max_{s' \in \mathcal{IN}(s)} (F(s') + h_s)$$

For any subset S' of the item set, define $H(S') = \max_{s \in S'} F(s)$. For an arbitrary subset S of the item set, we additionally define the following partition:

$$S_{mid} = \{s : (F(s) \geq H(S)/2) \wedge (F(s) - h_s < H(S)/2)\}$$

$$S_{bot} = \{s : F(s) < H(S)/2\}$$

$$S_{top} = \{s : F(s) - h_s \geq H(S)/2\}$$

The following lemmas form the foundation of the packing algorithm.

Lemma 1: For a fixed arbitrary y, let S' be the set of items s such that $F(s) \geq y$ and $F(s) - h_s < y$. Then there are no dependencies between the items in S'.

Proof. Assume for a contradiction that there are $s, s' \in S'$ such that there is a path in G from s to s'. Let s_{pre} be the predecessor of s' on this path. Since $(s_{pre}, s') \in E$, $F(s') = \max_{s'' \in \mathcal{IN}(s')}(F(s'') + h_{s'}) \geq F(s_{pre}) + h_{s'}$. Since the values of the function F do not decrease along a path due to the definition by a maximum,

$$(*)\ F(s') \geq F(s) + h_{s'}$$

but since $s \in S'$, $F(s) > y$, and since $s' \in S'$, $F(s') - h_{s'} \leq y$, putting these inequalities together, we get

$$F(s) + h_{s'} > F(s'),$$

a contradiction to $(*)$.

Lemma 2: Let $S \neq \emptyset$ be a subset of the item set and S_{top}, S_{mid} and S_{bot} be the aforementioned partition. Then the set S_{mid} can not be empty.

Proof. We will prove this by contradiction too. Assume $S_{mid} = \emptyset$.

1. $\exists s \in S_{top}$:
   Since $F(s) = 0$ if $\mathcal{IN}(s) = \emptyset$ and $h_s > 0$, $F(s) - h_s < 0$ for items with inbound degree 0, and therefore, such items cannot be in S_{top}. Thus, any item in S_{top} must have an inbound degree of at least 1 in G.
   Now consider the subgraph G_{top} of G induced by S_{top}. Pick any item $s \in S_{top}$ with inbound degree of 0 in G_{top}. All items s' with $(s', s) \in E$ must be in S_{bot} (as S_{mid} is empty). Since $F(s) = \max_{s' \in \mathcal{IN}(s)} F(s') + h_s$, and $F(s') < H(S)/2$ for all $s' \in S_{bot}$,
   $$F(s) < H(S)/2 + h_s \Rightarrow F(s) - h_s < H(S)/2,$$
   thus, $s \notin S_{top}$, a contradiction.
2. $S_{top} = \emptyset$:
   By definition, $H(S) = \max_{s \in S} F(s)$. Since all items are in S_{bot}, there is at least one item $s \in S_{bot}$ such that $F(s) = H(S)$. But by definition of S_{bot}, $F(s) < H(S)/2$, a contradiction.

We can now use the following algorithm for scheduling the states of a state machine, using a subroutine $\mathcal{A}(y, S)$ that schedules the states in S at position y and returns the maximum height of an item in S under the placement: $\mathcal{A}(y, S) = \max_{s \in S}(y + h_s)$.

The following algorithm will be called $\mathcal{DC}$ in analogy with [8].

Algorithm $\mathcal{DC}(y, S)$

1. If $S = \emptyset$, return y
2. Recalculate $F(s)$ for each $s \in S$ using the subgraph of G induced by S
3. Partition S into S_{bot}, S_{mid} and S_{top} as defined above.
4. Assign $y_{bot} = \mathcal{DC}(y, S_{bot})$
5. Assign $y_{mid} = \mathcal{A}(y_{bot}, S_{mid})$
6. Return $\mathcal{DC}(y_{mid}, S_{top})$

Lemma 1 ensures that there are no dependencies between the items of S_{mid}, so they can all be placed at the same height in the strip. By lemma 2, we know that in each call to $\mathcal{DC}$ there is at least one item in S_{mid}, meaning that each call schedules at least one state. Thus, the algorithm is correct.

6.3. Applying the result to the sequencing of SSMs

Once we have a sequencing of the states, obtained by some method (for example the simplex or strip packing methods introduced above), the sequence can be applied to the target architecture. As with the KEP, this is done by thread priorities (see section 2.3). The processor always executes the thread with the highest priority, which means that the ordering of the states corresponds to decreasing thread priorities. So, in the example in Figure 6.1, the ordering would be VS, $S1$, $S4$, $S2$, $S5$, $S3$, $S6$ and $S7$ (this is due to the control flow dependency). KEP thread priorities could therefore be assigned as follows:

VS	4
$S1$	4
$S2$	2
$S3$	2
$S4$	3
$S5$	1
$S6$	1
$S7$	1

As mentioned above, smakc! uses an approximate algorithm for scheduling. The algorithm implemented in smakc! considers immediate transitions to be dependencies too, as the target of such a transition can be reached with only one tick, therefore the correct priority must be set already in the previous state. This method can produce suboptimal maximum priorities and can make some valid state machines unschedulable. An example for this would be a cyclic control flow among states, with all transitions immediate except for one. If that one transition contains a signal dependency, smakc! would consider these transitions as a dependency cycle.

In KASM, thread priorities are set with the `prio n` instruction. Such instructions are inserted into the code at transitions whenever the target state should be executed with a different priority than the source state. For a full code sample, see section 7.3.

6.4. Comparison to Boldt's compiler

While part of the real-time and embedded systems workgroup at the University of Kiel, Boldt wrote a compiler to translate the Esterel language to KEP assembler [29]. As outlined in chapter 2, SSMs and Esterel programs are equivalent. Boldt had to deal with scheduling too. We will now point out the differences and similarities between the two methods.

Boldt creates a vaguely treelike structure called the concurrent KEP assembler graph (CKAG) in which each node is labeled with a KASM instruction. The tree edges represent control flow successors (also taking into account several types of preemption successors and dependencies). Upon the CKAG, dependencies are calculated and schedulability is decided. Boldt stores two values for nodes: `prio` (the priority the node should be executed with) and `prionext` (the priority, with which the node should be resumed after a tick). In a recursive approach descending through the graph in DFS order, (next-)priorities are set as maximum of the priorities of successor nodes.

The obvious difference is that scheduling does not take place on source level, but on target (KASM) level, as KASM instructions are annotated. This has the advantage that granularity of scheduling is brought down to the smallest level, which will eliminate unnecessary priority changes in some cases. On the downside, scheduling becomes target platform dependent.

At first glance, the recursive, maximum-based approach seems different from our strip packing method, but reconsider the definition of the function F in subsection 6.2.2 for the special case where all items are of uniform height. That function basically implements the same concept. The addition of variable item sizes makes the algorithm more versatile. Boldt's compiler could be modified to also support KASM modules by assigning weights to the edges of the CKAG.

Solution	How	Where
Multiple threads and context switching	The operating system manages a list of processes and it assigns amounts of CPU time to them, allowing them to execute for some small amount of time before assigning the CPU to the next process (called "context switching")	software
Multiple CPU cores or multiple CPUs	Tasks are executed in true parallelism on different CPUs (or different cores). Here, new problems arise, since delays in communication between the CPUs have to be considered, as well as the maximum throughput of the wires leading to a multicore CPU. Also, multiple CPUs might want to access the same memory block at the same time	mostly hardware
One multithreading-capable CPU core	The operating system is mostly relieved of having to manage context switching, but the programmer or compiler is additionally burdened with having to fix in advance a scheduling for the user programs	hardware and software

Table 6.1.: Variations of scheduling.

7. smakc! implementation

This is how we go about it — to make our heads explode all night!

Monster Magnet: Heads Explode

smakc! was implemented in the Java programming language and uses several other projects, in particular Eclipse EMF [1] for KIELER integration, and the Apache Velocity engine for code generation.

7.1. Compiler package

As noted earlier, the compiler simply takes a set of state machines and an ordered list of transformations as arguments, and applies the transformations to the state machines in the given order. The transformations must implement the `smakc.compiler.interfaces.compiler.Transformation<A,B>` interface. The interface consists of a sole method, the `Collection<A>transform(Collection<B>)` method, meaning that a set of objects of type `B` should be transformed to a set of objects of type `A`. The compiler applies the first transformation to the input state machines, caches the result, and continues with the next transformation. The relationship between these classes is depicted in Figure 7.1. The class is implemented using sets of SSMs since originally, a linker was also planned which would link reference macrostates to the corresponding state machine. The transformations naturally have to be applied in the correct order if a later transformation requires an extension of the state machine which is provided by an earlier transformation. Transformations must be located in the package `smakc.compiler.transformations`.

The SSMs are represented in an internal datastructure also represented by interfaces. During the development of smakc!, a SSM data model was also developed in parallel for the KIELER project [3]. After significant simplification, the KIELER SSM model is now almost equal to the smakc! internal model. As smakc! also accepts the KIELER model as input, we will take a look at the KIELER model in Figure 7.2 and point out the differences:

The emissions class is missing entirely, since the KEPe does not support valued signals. Suspensions as well as the suspension immediate flag are located in the state class for simplicity. Signal renamings belong to the special BlakcBox (not a spelling mistake) class which is used to represent entire encapsulated SSMs.

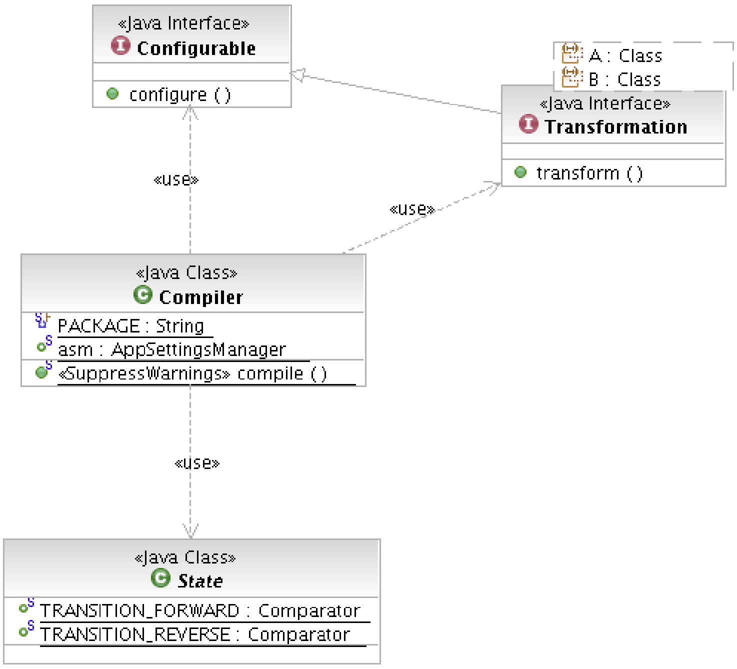


Figure 7.1.: The `Compiler` uses `Transformation`s which are `Configurable` to process `State`s, which refer to the root state of a state machine.

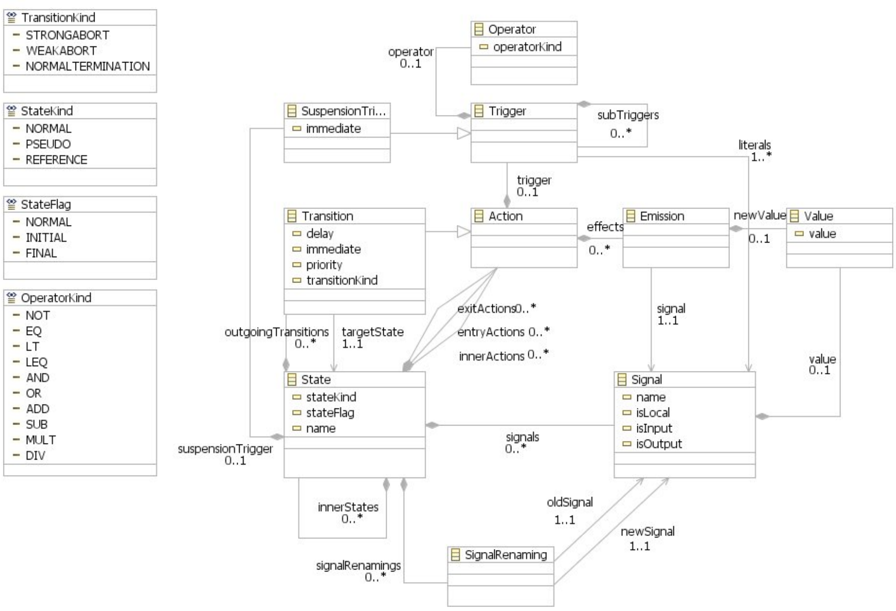


Figure 7.2.: The SSM model as used by the KIELER project

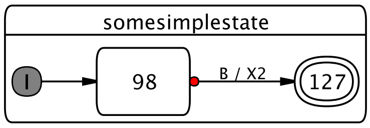


Figure 7.3.: A simple state with an outbound transition.

7.2. Statemachineproviders package

This package contains the input modules for different SSM formats. The format loaders must implement the interface StateMachineLoader which has only one method: State load(String source) which is supposed to load the file designated by the source string and return a state (the topmost state of the state machine). smakc! looks for the implementation in the smakc.statemachineproviders.<file extension> package for a class called LoaderImpl and will use it, if found.

The loader package itself is responsible for the representation and implementation of the ssm interfaces, for a simple reason: SSMs could be represented very differently in their source format, and what should be loaded immediately and what can be "lazily" loaded differs as formats differ. A lot of optimization could be lost when forcing all formats to the smakc! internal format.

7.3. Generating code with the Apache Velocity engine

The Apache Velocity [35] engine was chosen for code generation. It is a templating engine, meaning that it uses static content templates with placeholders which will be filled with datastructure values by the engine at parse time. Velocity additionally provides a simple scripting language which allows setting variables, looping over arrays and conditional code output. It was originally designed for dynamic web content. The engine caches templates that have already been requested and reuses them when requested again to avoid unnecessary function calls and memory usage. We chose a templating engine because of its versatility, for migrating to a new ISA should be only a matter of creating a new set of templates.

To make states recognizable in the target code, state parts start and end with specific keywords, represented as jump labels in code. A state initializer block begins with BEGINSTARTUP and ends with ENDSTARTUP. The state itself analogously starts with BEGINSIMPLESTATE and ends with ENDSIMPLESTATE (in case of a simple state). To illustrate this, the code for the state shown in Figure 7.3 is shown below:

```
BEGINSTARTUP98:
        ABORT B, ENDABORT_98B127_P1
ENDSTARTUP98:

BEGINSIMPLESTATE98:
        PAUSE
        GOTO BEGINSIMPLESTATE98
ENDSIMPLESTATE98:
```

```

BEGINSHUTDOWN98:
ENDSUSPEND98:
ENDABORT_98B127_P1:
        EMIT X2
        PRIO 1
        GOTO BEGINSTARTUP127
ENDSHUTDOWN98:
```

As the generated code shows, the transition has been translated to an abort block which spans the entire state. The state itself just idles. When signal B becomes present, the abort block fires and the processor starts executing code at the abort end label. The signal *X2* is emitted and priority is switched. Then, the `goto` statement jumps to the beginning of the startup block of state 127 (the final state seen in Figure 7.3). The outer startup and shutdown blocks are common to (almost) all state types. The inner simple state block is only parsed in for simple states. smakc! also implements some optimizations to the code which are made possible by the KEP ISA: A simple state without inner code (actions) is represented as a `halt` instruction, and an empty state with only one outbound transition as optimized to an `await` instruction. Complex states such as the one in Figure 7.4, get the following inner code block:

```
BEGINSTARTUPMODULE_ABORT40B:
ENDSTARTUPMODULE_ABORT40B:

BEGINCOMPLEXSTATEMODULE_ABORT40B:
        PAR 1, BEGINSTARTUPINITIAL_0, 1
        PAR 1, BEGINSTARTUPINITIAL_1, 2
        PAR 1, BEGINSTARTUPINITIAL_2, 3
        PAR 1, BEGINSTARTUPINITIAL_4, 4
        PAR 1, BEGINSTARTUPINITIAL_5, 5
        PARE SUBSTATESENDMODULE_ABORT40B, 0

        (inner states here)

        SUBSTATESENDMODULE_ABORT40B:
        JOIN 1
        HALT
ENDCOMPLEXSTATEMODULE_ABORT40B:

BEGINSHUTDOWNMODULE_ABORT40B:
ENDSUSPENDMODULE_ABORT40B:
ENDSHUTDOWNMODULE_ABORT40B:
```

As an optimization, conditional pseudostates are not implemented as states with abort blocks around them, but rather as series of `present` tests (just like several `if` tests). The conditional pseudostate in Figure 7.5 generates the following code:

```
BEGINSTARTUP37:
ENDSTARTUP37:

BEGINCONDITIONALPSEUDOSTATE37:
        PRESENT W, ENDSIGNALTEST_W_37
                EMIT X1
```

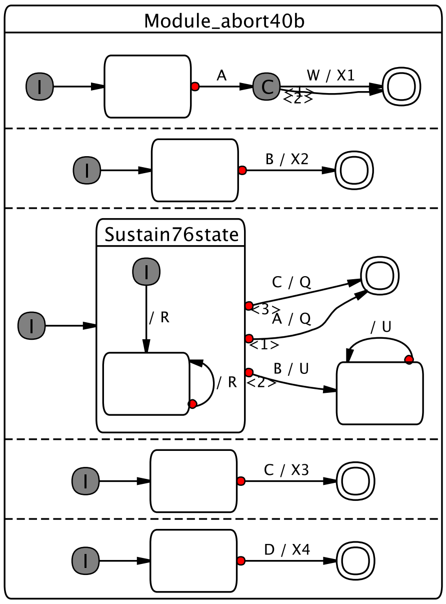


Figure 7.4.: A complex state.

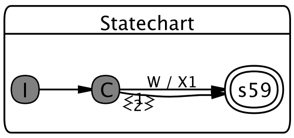


Figure 7.5.: Conditional pseudostate with priority 1 on the W transition.

```
            PRIO 2
            GOTO BEGINSTARTUP59
        ENDSIGNALTEST_W_37:
        PRESENT TICK, ENDSIGNALTEST_TICK_37
            PRIO 2
            GOTO BEGINSTARTUP59
        ENDSIGNALTEST_TICK_37:
ENDCONDITIONALPSEUDOSTATE37:

BEGINSHUTDOWN37:
HALT
ENDSHUTDOWN37:
```

For more about Velocity script coding, please refer to [35]. We will just show the conditional pseudostate as a simple example:

```
#set ($empty = ${list.add(${State})})

BEGINSTARTUP${State.getId().toUpperCase()}:
#foreach ($action in ${State.getStateEntryActions()})
    #parse("${basepath}Action.kasm")
```

```
#end
ENDSTARTUP${State.getId().toUpperCase()}:
BEGINCONDITIONALPSEUDOSTATE${State.getId().toUpperCase()}:
#foreach ( $transition in ${State.getScheduledOutboundTransitions()})
  #if (${ transition .hasCondition()})
    #set ($condname = ${transition.getCondition(). getExtendedName().toUpperCase()})
  #else
    #set ($condname = "TICK")
  #end
  PRESENT ${condname}, ENDSIGNALTEST_${condname}_${State.getId().toUpperCase()}
  #foreach ( $signal in ${ transition . getEffectSignals ()})
    #parse("${basepath}Emit.kasm")
  #end
  #if (${State. getThreadPriority ()} != ${ transition . getDestination (). getThreadPriority ()})
    PRIO ${transition . getDestination (). getThreadPriority ()}
  #end
  GOTO BEGINSTARTUP${transition.getDestination().getId().toUpperCase()}
  ENDSIGNALTEST_${condname}_${State.getId().toUpperCase()}:
#end
ENDCONDITIONALPSEUDOSTATE${State.getId().toUpperCase()}:
BEGINSHUTDOWN${State.getId().toUpperCase()}:

HALT

ENDSHUTDOWN${State.getId().toUpperCase()}:

#foreach ( $transition in ${State.getScheduledOutboundTransitions()})
  #set ($State = ${ transition . getDestination ()})
  #if (!${ list . contains($State)})
    #if (${State. isConditionalPseudoState ()})
      #parse("${basepath}PseudoState.kasm")
    #else
      #parse("${basepath}InitBlokc.kasm")
    #end
  #end
#end
```

The code shows some features of the Velocity scripting language. For example, a `foreach` loop is provided. In this example, it is used to traverse the outbound transitions and parse them into the code, additionally adding `emit` and `prio` statements where necessary. The `parse` directive parses a subtemplate into the current template at the point where it is called.

The scripting capabilities and the template mechanism should make it fairly easy to migrate to new architectures.

8. Experimental results

Sensory perception only — O son of Kuntī; winter, summer, happiness and pain; giving, appearing, disappearing; nonpermanent, all of them; just try to tolerate, O descendant of the Bharata dynasty.

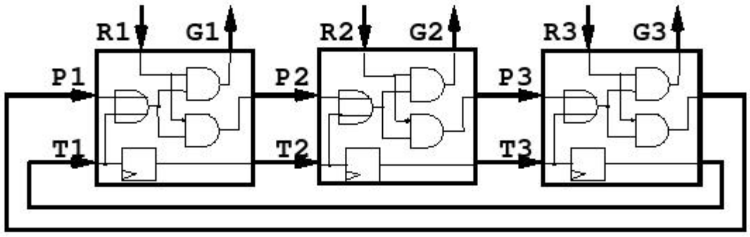
Krishna, in the Bhagavad Gītā

8.1. Automated verification of compilation results

The output code produced by smakc! must be verified. For this, we used the same samples as benchmark originally created by Boldt [29], Li [30] and Tiedje [45]. In total, there are 757 Esterel programs testing various Esterel language features. Figure 8.1 lists the maximum counts per operator in all samples. Figure 8.2 lists the average count per operator. They were automatically translated to SSMs in the kit format by the KIEL tool [33]. Using the Krepevalbench [2], the results produced by smakc! could be verified against original Esterel traces produced by Esterel Studio, see Figure 8.3. Since the KEPe was used for verification, code using valued signals could not be used. Some other examples could also not be ported to kit due to version conflicts. In total, 428 of the 757 examples could be used in the verification process.

8.2. Compilation speed and code size

For performance testing, a prominent example called the "token ring arbiter" was used. As the name already implies, stations are arranged on a ring, and a token (representing the usage of a shared resource) is passed through the stations. A token ring with three stations is shown below:



A station is connected in the network by its token (T) and pass (P) input/output signals, and locally by its request (R) and grant (G) signals. At startup, one station

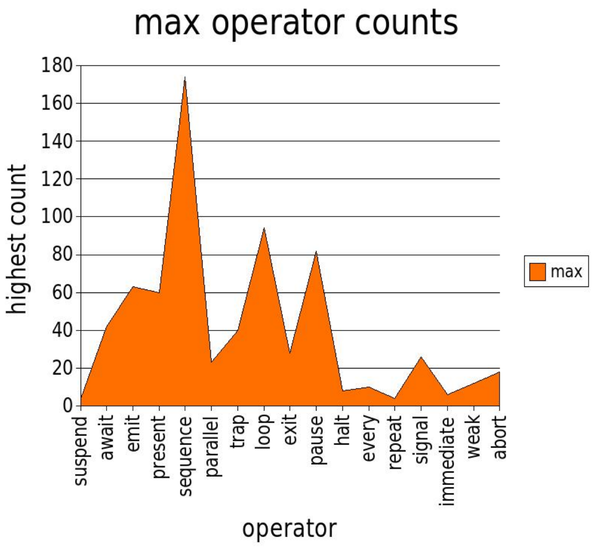


Figure 8.1.: Maximum operator count encountered in the benchmark samples, per operator.

gets the token. The token ring arbiter example is one of the most often used benchmarks in sychronous language processing, the name of its inventor however seems to have been forgotten. Berry first introduced it as example for Esterel [10]. However he attributes it to someone else.

The token ring arbiter is an example of cyclic data dependency although the network does not suffer from causality problems since the token is given to one station at startup. It also shows the power of SSMs, or Esterel: The stations all run in synchronous concurrency, causing extreme state counts in equivalent automata. The high level of concurrency makes it a good example for performance testing. Due to the cyclic dependency in the token ring arbiter, scheduling could not be applied, but smakc! was tested on rings of sizes 5, 10, 20, 30, 40, 50, 60, 70, 80, 90, 100, 150, 200, 250, 300, 350, 400, 450 and 500 stations using four configurations: In the first configuration, conditional expressions were resolved, then dependency and cycle detection were applied (1). Since the cycle detection employs the only $\mathcal{O}(n^3)$ algorithm in an otherwise $\mathcal{O}(n \log n)$ process, the second configuration was resolving conditionals and dependency detection only, for comparison (2). The third configuration for the token ring arbiter was resolving conditionals, upgrading the states to states with thread priorities and finally generating KASM code (3). Note that the scheduling was left out, since the DDG contains cycles. Instead, in configuration (4), a variant of the token ring arbiter in which the feedback line to the first station is missing was used. This example can be compiled by both smakc! and Boldt's *strl2kasm* compiler. Thus, the last configuration allowed measuring compile times of both compilers. For measuring CPU time, the Unix *time* utility was used.

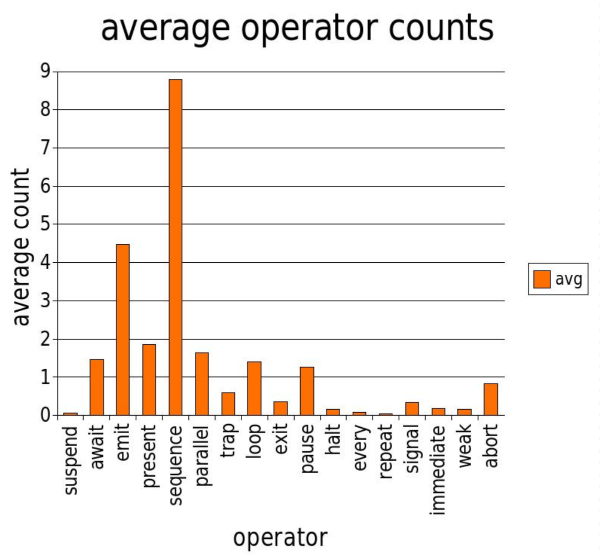


Figure 8.2.: Average operator count encountered in the benchmark samples, per operator.

Figure 8.4 shows the results on configuration (1). The runtime quickly exceeds the average user's patience, the 150 station ring already takes almost 4 minutes, and the 300 station ring takes not quite half an hour.

At this point, it should be noted that the SSM for a station of the token ring already consists of several states and a transition with a complex conditional expression further adding to the state count after conditionals resolving. Therefore, a 500 station ring consists of roughly 6000 states by the time the cycle detection is launched.

An advantage of the Floyd-Warshall algorithm is that although it sets up a n^2 size matrix the calculation of the matrix requires neither more space nor more function calls, so once the algorithm has been started, it will also terminate without an "out of memory" error.

As expected, leaving out the $\mathcal{O}(n^3)$ cycle detection in configuration (2) drastically cuts the runtime, as can be seen in Figure 8.5 (note that the numbers on the runtime axis are smaller by two orders of magnitude).

Generating code in configuration (3) was mainly influenced by the Apache Velocity [35] engine (see chapter 7). Code generation fails for rings of size greater than 200 (that is about 2400 states), since the Velocity engine steps through the structure in DFS, and recursive calls stack up memory space, exceeding the 2GB limitation of the Java VM used for testing. Code generation time also matches the overall $\mathcal{O}(nlogn)$ runtime as can be seen in Figure 8.6, except for smaller values, which is due to the initialization of the templating engine.

The least surprising result is the size of the generated code. Thanks to the templating mechanism (see chapter 7), the size of the code for a single state can be bounded from above by the code for the state with the highest outbound transition

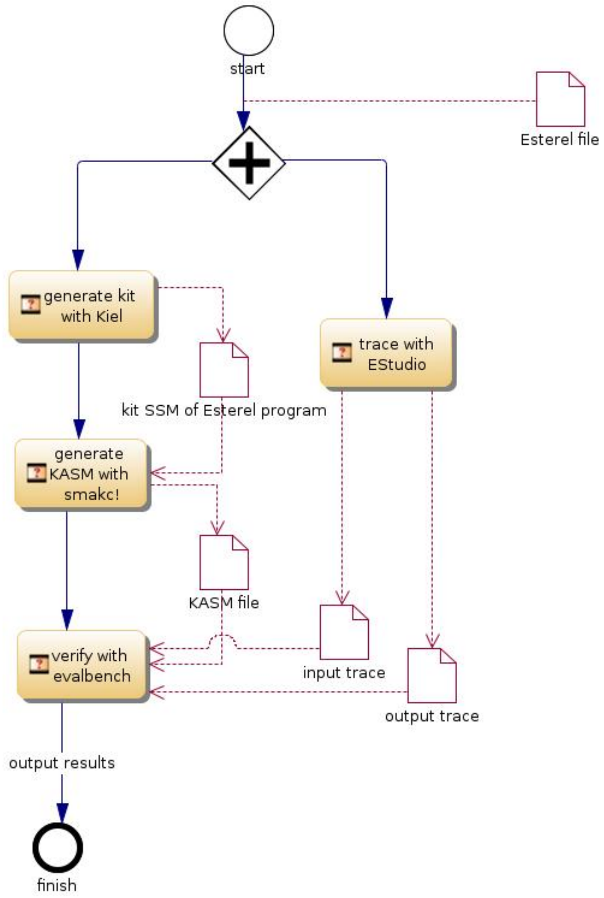


Figure 8.3.: The validation process.

count. Therefore, the code size is linear in the number of states, see Figure 8.7.

Another result that is not really surprising is the compile time of the two compilers pitched against each other. For this performance test, the token ring was split to a token line, allowing both compilers to produce scheduled code. As *strl2kasm* is implemented in C++ and compiled to native binary code, whereas smakc! was written in Java and executed as bytecode by the Java VM, naturally smakc! is slower. Additionally, smakc!s transformations are all implemented sequentially to keep the program modular, while *strl2kasm* generates most information needed in a single pass on the source code. Figure 8.8 shows the results. Obviously, both compilers have equal runtime in terms of the $\mathcal{O}$ notation, but smakc! has larger constant factors (of about 10).

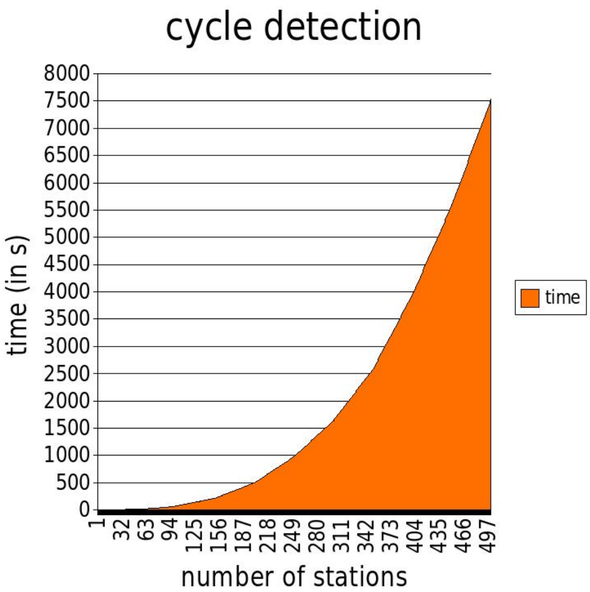


Figure 8.4.: Compile time with conditionals resolving, dependency and cycle detection.

8.3. Two-way comparison to Boldt's compiler

In this section we will compare smakc! to the Esterel compiler of Marian Boldt (*strl2kasm*). To make the challenge fair we will first start from SSMs, directly compiling them with smakc!. Then we will use Esterel Studio [44] to convert the SSM to Esterel and compile the code with Boldt's compiler. In the second test, we will start from Esterel code, directly compiling it with Boldt's compiler, and then porting the Esterel code to an SSM using KIEL [4]. After that, we can apply the smakc! transformation. The two-way comparison process is depicted in Figure 8.9.

Since instruction memory is scarce on embedded computers, we will especially pay attention to code size, usage of watcher ids and thread ids (since each watcher spared is less chip space used) and runtime of the generated code.

The Esterel examples we used for testing are listed in Appendix C, the SSMs in Appendix D. As Esterel Studio[1] already optimizes code when exporting to Esterel, the optimize feature of KIEL was applied to all state machines. Additionally, as smakc! produces a lot of labels which only serve the purpose of recognizing the original states in the generated code and a lot of whitespace results from the templating process, another module was run on smakc!s output code which strips the code of whitespace and labels that are never targeted.

Both compilers do optimizations. smakc! represents empty simple states as `halt` instructions, and states waiting for just one condition as `await` statements. Condi-

[1]Not to be confused with hair spray.

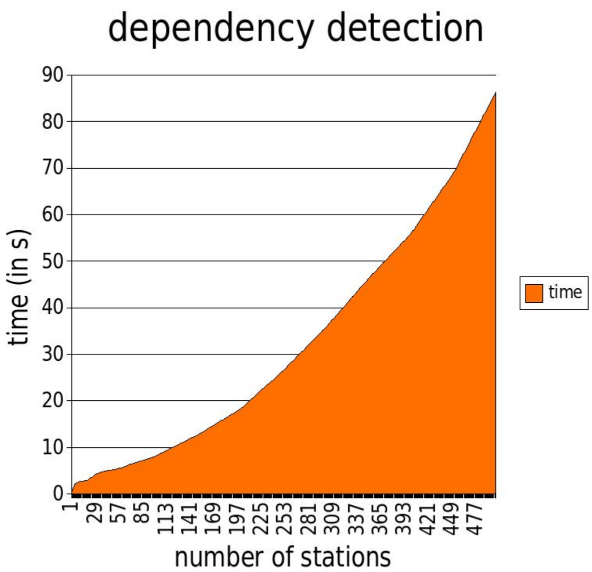


Figure 8.5.: Compile time with conditionals resolving and dependency detection.

tional pseudostates are implemented as series of `present` tests. The algorithm also tries to save on abort watcher and thread ids. The latter also goes for Boldt's compiler. He does not save on watcher ids though since not all KEP versions support assigning watcher ids. As Boldt does not compile from SSMs but rather from Esterel, he takes a different approach at optimizing code size. The *strl2kasm* compiler performs some "undismantling" on the finished code to transform statements such as an empty loop back to a `halt` instruction.

8.3.1. Code size

For code size, we measured the file size (Figure 8.10), line count (Figure 8.11) and line count of the generated KEP listing file (Figure 8.12). The listing file line count corresponds directly to the size of the program in the instruction memory of the KEP, which is why it was not necessary to explicitly compare binaries. The KASM file sizes differ a lot, since smakc! produces much longer labels, making smakc! much less competitive in this respect.

Since the KEP ISA was kept close to Esterel syntax and semantics, it is no surprise the generated code looks very similar. In the case of the ABRO program, apart from the labels, the code is almost the same (Figure 8.13). In all tests, the VEND_B example seems to be off the normal values. The vending machine was translated to several `EXIT` statements, which are the KASM equivalents of the Esterel `trap/exit` statement, bloating the code. The "parhierarchy" example was especially created as an example that translates to Esterel very badly and produces a lot of code, causing the higher line count in Figure 8.11.

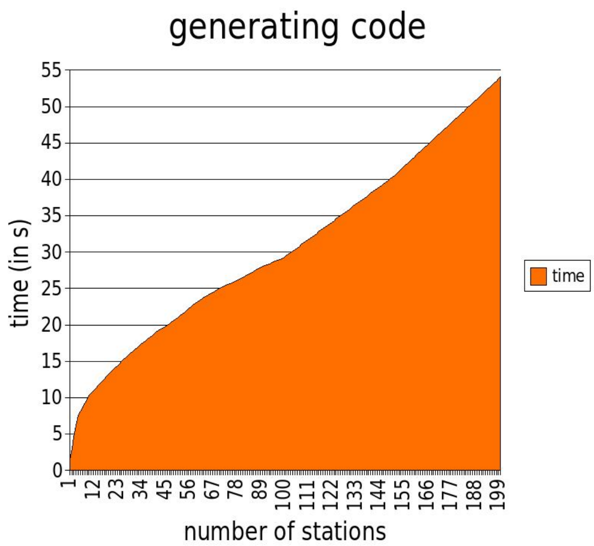


Figure 8.6.: Compile time for generating target platform code.

8.3.2. Watchers

Watchers for aborts (or for trap/exits) also take up more space on the KEP microprocessor. Boldt's compiler *strl2kasm* does not save on watcher ids. Most KEP versions do not support assigning watcher ids anyway. Just to demonstrate the feature by simple example, the SSM in Figure 8.14 can reassign all watcher ids used immediately, as the code runs entirely sequentially. smakc! does this, as the generated code shows. Watcher ids are the last numbers in a line starting with `abort` or `wabort`:

```
%%% ---BEGIN KEP CODE---
EMIT _TICKLEN,#0
ABORT TICK, ENDABORT_S1S2_P1, 1
ABORT TICK, ENDABORT_S1S3_P2, 2
HALT
ENDABORT_S1S3_P2:
GOTO BEGINSTARTUPS3
ENDABORT_S1S2_P1:
GOTO BEGINSTARTUPS2
BEGINSTARTUPS2:
ABORT TICK, ENDABORT_S2S4_P1, 1
ABORT TICK, ENDABORT_S2S5_P2, 2
HALT
ENDABORT_S2S5_P2:
GOTO BEGINSTARTUPS5
ENDABORT_S2S4_P1:
GOTO BEGINSTARTUPS4
BEGINSTARTUPS4:
HALT
BEGINSTARTUPS5:
HALT
BEGINSTARTUPS3:
```

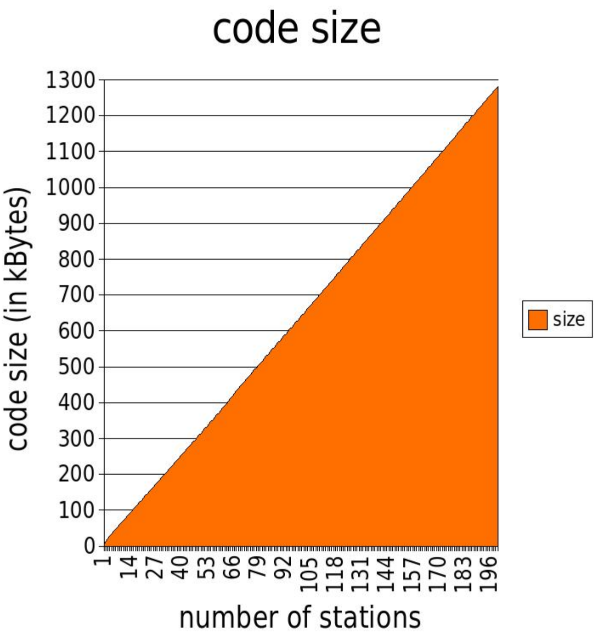


Figure 8.7.: Size of the code generated for the token ring.

```
ABORT TICK, ENDABORT_S3S6_P1, 1
ABORT TICK, ENDABORT_S3S7_P2, 2
HALT
ENDABORT_S3S7_P2:
GOTO BEGINSTARTUPS7
ENDABORT_S3S6_P1:
GOTO BEGINSTARTUPS6
BEGINSTARTUPS6:
HALT
BEGINSTARTUPS7:
HALT
HALT
%%% ---END KEP CODE---
```

The numbers found in Figure 8.15 are not really comparable, since *strl2kasm* does not try to save abort watchers by reusing them. Some watchers can be optimized anyway, as single aborts can be "undismantled" back to `await` statements.

8.3.3. Thread priorities

Both compilers try to reuse threads, as threads also require watchers. The thread watchers monitor the thread's program counter to check if it is within the thread's instruction scope. If it leaves that scope, the thread is considered terminated. By reusing thread ids, these units can be saved. Here, both compilers fare equally well (Figure 8.16).

Code generated for a specific example, the thread saver test, is also almost equal

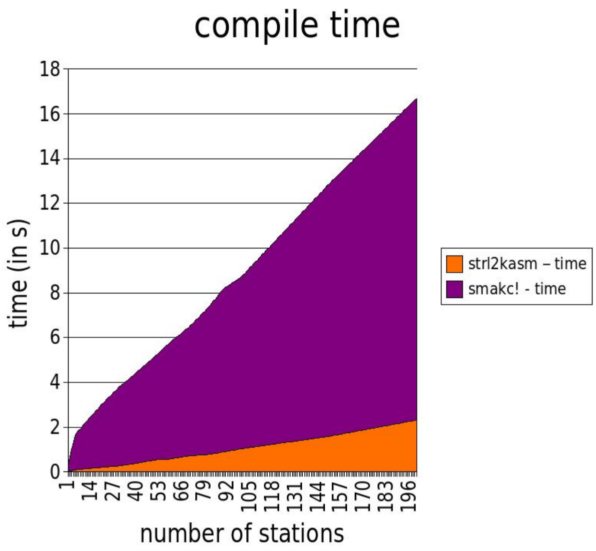


Figure 8.8.: Compile time of *smakc!* and *strl2kasm* on the token line.

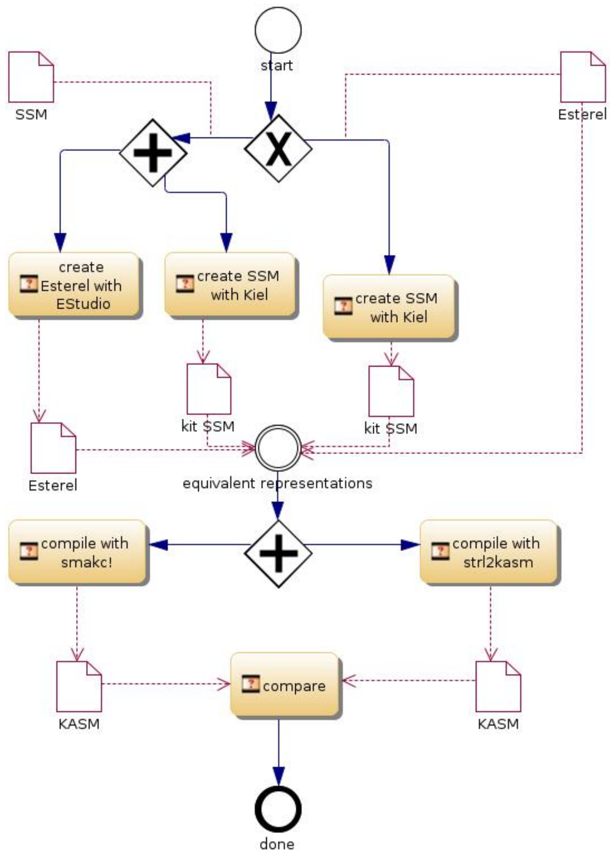


Figure 8.9.: Process of comparing *smakc!* with *strl2kasm* by Boldt.

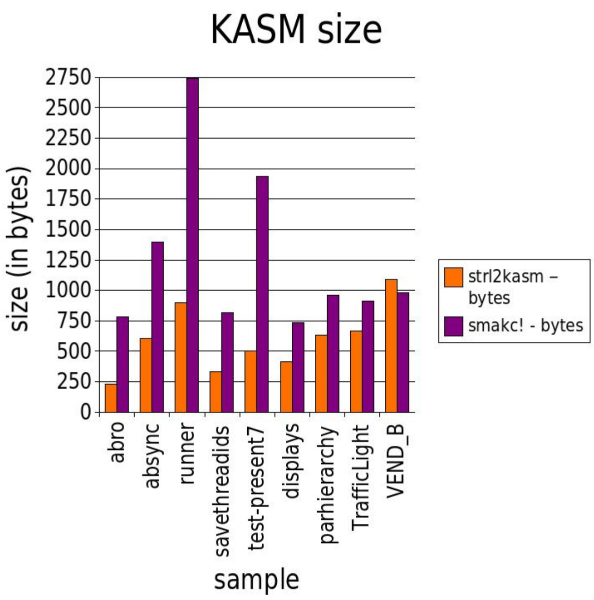


Figure 8.10.: File size of the generated KASM files. smakc! competitive only on state machines ("displays" and beyond)

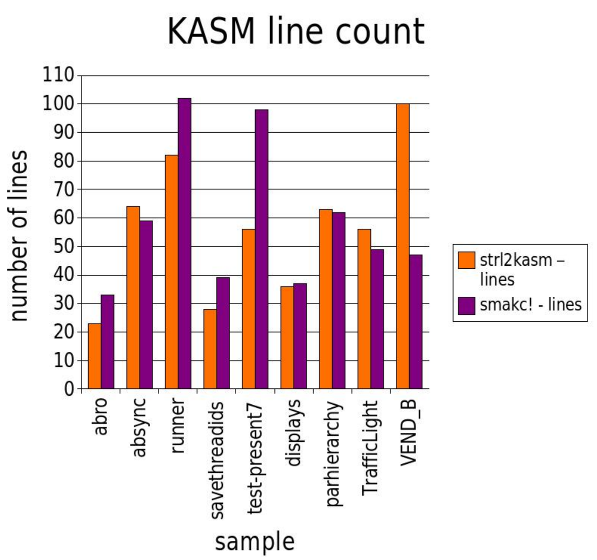


Figure 8.11.: Line counts of the generated KASM files. "parhierarchy" was especially created to translate badly to Esterel, "VEND_B" is a surprising anomaly.

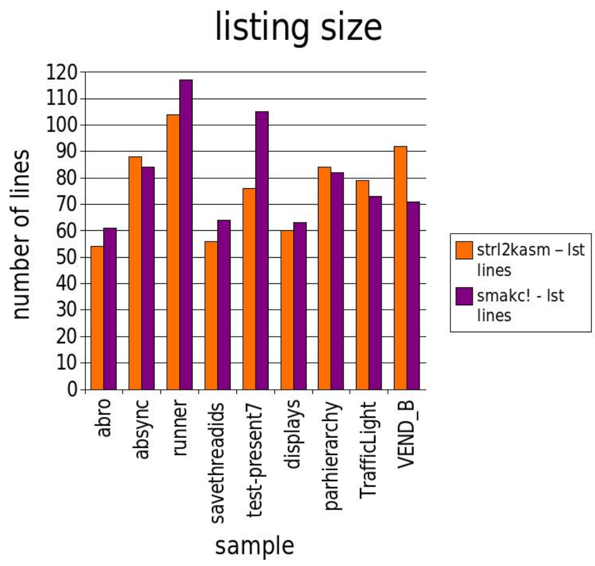


Figure 8.12.: Sizes of the listing generated from *kasm2klist*.

between the two compilers (Figure 8.17).

8.3.4. Reaction times

The reaction time is the time a program's tick takes, counted by the number of instructions executed. The longest tick is especially important since the hardware has to be clocked according to the length of the longest tick and real-life physical timing constraints. The timing constraints and the length of the longest tick are fixed for a given problem and a given KASM program implementing a solution to that problem. By those two and a simple formula, the necessary clock rate of the hardware can be determined. However, by making the tick length as short as possible, the clock rate can also be slower, resulting in less power consumption. The examples used in the two-way compare were also checked for minimum (Figure 8.18), average (Figure 8.19) and maximum (Figure 8.20) tick length. The results show that in some cases, smakc! performs better than *strl2kasm* on regular Esterel code (see for example "absync" and "test_present7" in Figure 8.20), and *strl2kasm* performs better than smakc! on SSMs ("savethreadids" in Figure 8.20). However, in general, the KASM code produced by *strl2kasm* has the better reaction times.

One should note that reaction times were measured by the K(r)epevalbench [2] using random traces generated by Esterel Studio. The traces do not perform an exhaustive search on all possible program and signal states, and therefore the minimum and maximum tick lengths could differ a little from the measured values presented here.

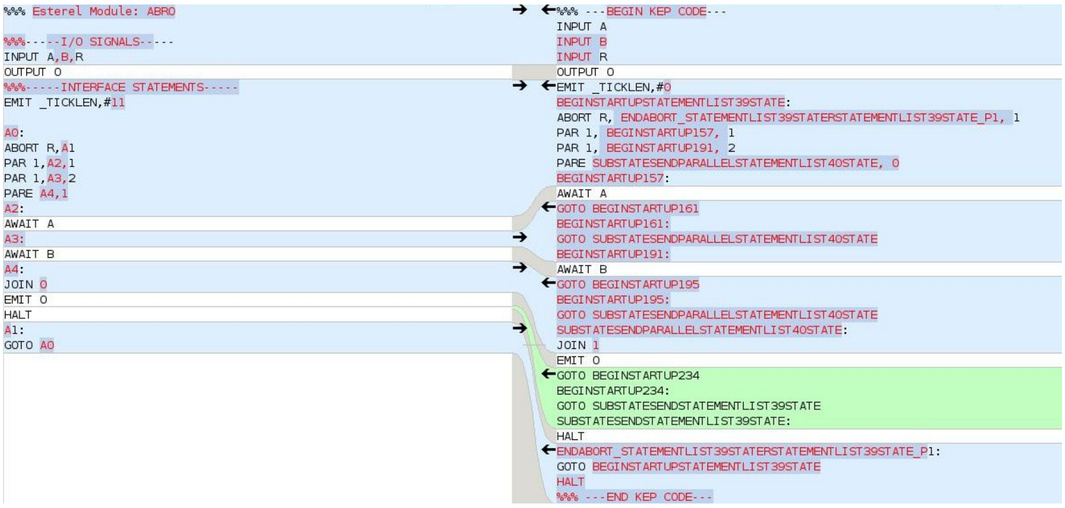


Figure 8.13.: A diff by *meld* on the KASM code for ABRO, on the left generated by *strl2kasm*, on the right by *smakc!*. Both programs clearly have the same structure.

8.3.5. Examination of an example

We will now examine a special example in detail. The example (Figure 8.21) is basically the complete graph on three nodes.

The SSM was originally created with Esterel Studio. It is exported to the Esterel code shown on page 63. As the code shows, the states are represented in a very unintuitive way.

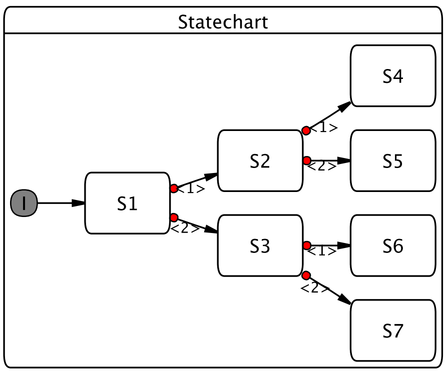


Figure 8.14.: A SSM to test the usage of watchers.

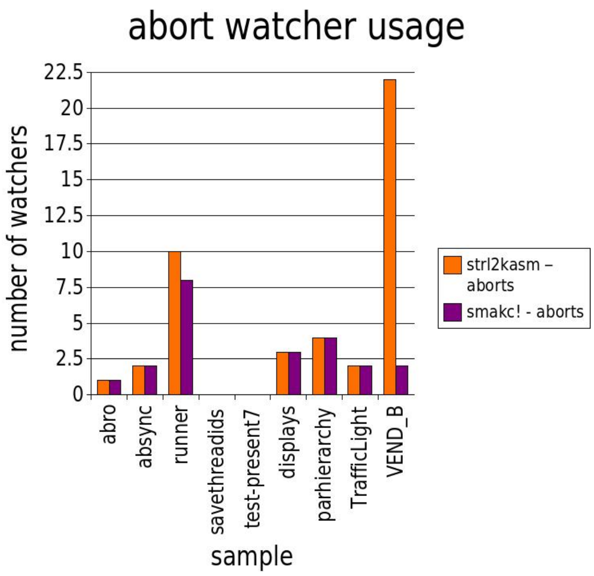


Figure 8.15.: Usage of abort watchers in the example programs. Exits were counted as aborts in VEND_B.

```
module threenode:
input R,S,T;
signal sc_cache in
 signal sc_go_1_S, sc_go_2_T in
  emit sc_cache;
  loop
   present
   case [sc_go_2_T] do
    % state T
    await
    case [R] do
     emit sc_cache
    case [S] do
     emit sc_cache;
     emit sc_go_1_S
    case [T] do
     emit sc_cache;
     emit sc_go_2_T
    end await
   case [sc_go_1_S] do
    % state S
    await
    case [R] do
     emit sc_cache
    case [T] do
     emit sc_cache;
     emit sc_go_2_T
    case [S] do
     emit sc_cache;
     emit sc_go_1_S
```

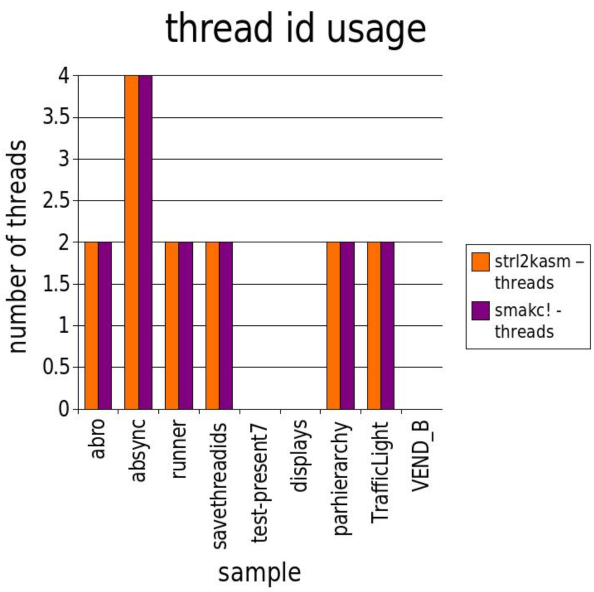


Figure 8.16.: Usage of threads — *strl2kasm* and *smakc!* equally good

```
      end await
     else
      % state R
      await
      case [S] do
       emit sc_cache;
       emit sc_go_1_S
      case [T] do
       emit sc_cache;
       emit sc_go_2_T
      case [R] do
       emit sc_cache
      end await
     end present
    end loop
  end signal
end signal
end module
```

The power of the goto statement becomes clear with this example, as smakc! has the advantage in transforming the transitions of the states. The code generated by smakc! is found on page 65, the *strl2kasm* code on page 67.

Figure 8.17.: A diff by *meld* on the KASM code for the thread id saver example. On the left the *strl2kasm* code, on the right *smakc!* generated code.

```
%%% ---BEGIN KEP CODE---
INPUT R
INPUT S
INPUT T
EMIT _TICKLEN,#0
BEGINSTARTUPR:
ABORT S, ENDABORT_RSS_P1
ABORT T, ENDABORT_RTT_P2
ABORT R, ENDABORT_RRR_P3
HALT
ENDABORT_RRR_P3:
GOTO BEGINSTARTUPR
ENDABORT_RTT_P2:
GOTO BEGINSTARTUPT
ENDABORT_RSS_P1:
GOTO BEGINSTARTUPS
BEGINSTARTUPS:
ABORT R, ENDABORT_SRR_P1
ABORT T, ENDABORT_STT_P2
ABORT S, ENDABORT_SSS_P3
HALT
ENDABORT_SSS_P3:
GOTO BEGINSTARTUPS
ENDABORT_STT_P2:
GOTO BEGINSTARTUPT
ENDABORT_SRR_P1:
GOTO BEGINSTARTUPR
BEGINSTARTUPT:
ABORT R, ENDABORT_TRR_P1
```

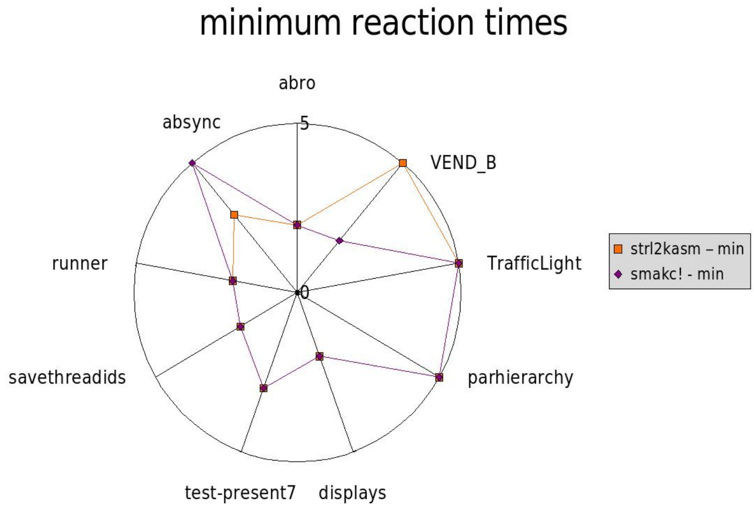


Figure 8.18.: Shortest tick length in the sample programs.

```
ABORT S, ENDABORT_TSS_P2
ABORT T, ENDABORT_TTT_P3
HALT
ENDABORT_TTT_P3:
GOTO BEGINSTARTUPT
ENDABORT_TSS_P2:
GOTO BEGINSTARTUPS
ENDABORT_TRR_P1:
GOTO BEGINSTARTUPR
HALT
%%% ---END KEP CODE---
```

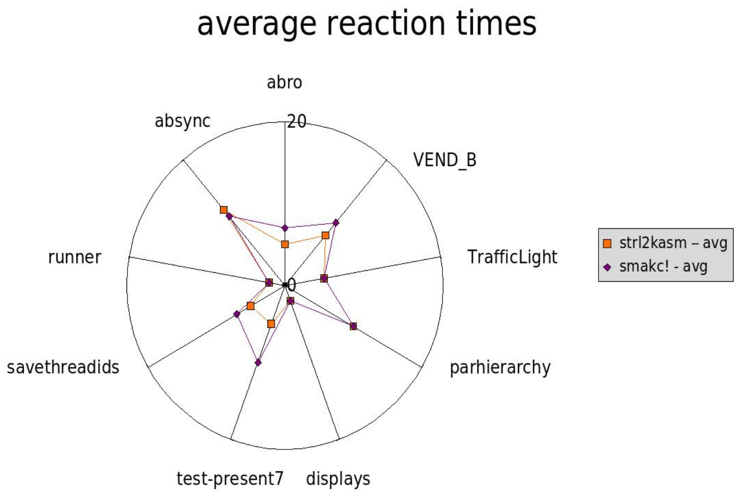


Figure 8.19.: Average tick length in the sample programs.

```
%%% Esterel Module: threenode

%%%-----I/O SIGNALS-----
INPUT R,S,T
%%% ERROR: NO OUTPUT SIGNALS, DEFINE DUMMY:
OUTPUT _NO_OUTPUT_PORT_ERROR
%%%-----TOP LOCAL SIGNALS-----
SIGNAL SC_CACHE,SC_GO_1_S,SC_GO_2_T
%%%-----INTERFACE STATEMENTS-----
EMIT _TICKLEN,#13

EMIT SC_CACHE
A0:
PRESENT SC_GO_2_T,A1
A3:
A4:
A5:
A6:
A7:
PAUSE
PRESENT R,A8
EXIT AC,A4
A8:
PRESENT S,A9
EXIT AC_0,A5
A9:
PRESENT T,A10
EXIT AC_1,A6
A10:
GOTO A7
AC_1:
EMIT SC_CACHE
EMIT SC_GO_2_T
EXIT AWAIT_CASE,A3
```

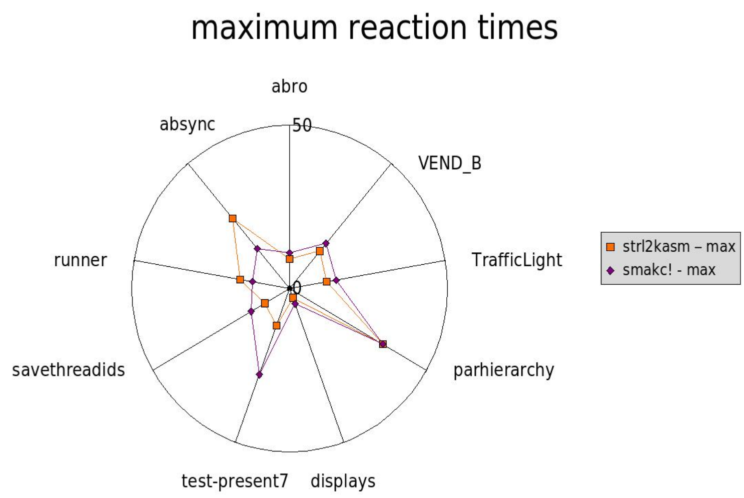


Figure 8.20.: Length of the longest tick in the sample programs.

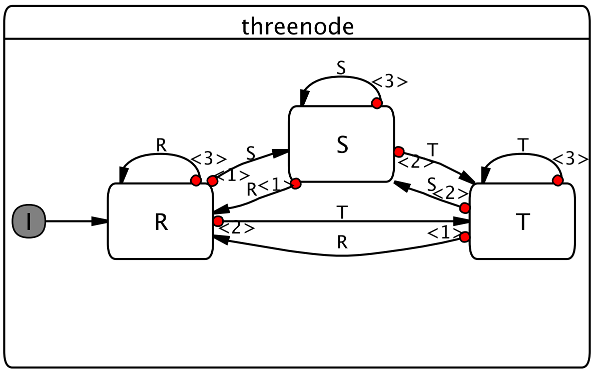


Figure 8.21.: The complete graph on three nodes as SSM.

```
AC_0:
EMIT SC_CACHE
EMIT SC_GO_1_S
EXIT AWAIT_CASE,A3
AC:
EMIT SC_CACHE
EXIT AWAIT_CASE,A3
AWAIT_CASE:
GOTO A2
A1:
PRESENT SC_GO_1_S,A11
A13:
A14:
A15:
A16:
A17:
PAUSE
```

```
PRESENT R,A18
EXIT AC_2,A14
A18:
PRESENT T,A19
EXIT AC_3,A15
A19:
PRESENT S,A20
EXIT AC_4,A16
A20:
GOTO A17
AC_4:
EMIT SC_CACHE
EMIT SC_GO_1_S
EXIT AWAIT_CASE_0,A13
AC_3:
EMIT SC_CACHE
EMIT SC_GO_2_T
EXIT AWAIT_CASE_0,A13
AC_2:
EMIT SC_CACHE
EXIT AWAIT_CASE_0,A13
AWAIT_CASE_0:
GOTO A12
A11:
A21:
A22:
A23:
A24:
A25:
PAUSE
PRESENT S,A26
EXIT AC_5,A22
A26:
PRESENT T,A27
EXIT AC_6,A23
A27:
PRESENT R,A28
EXIT AC_7,A24
A28:
GOTO A25
AC_7:
EMIT SC_CACHE
EXIT AWAIT_CASE_1,A21
AC_6:
EMIT SC_CACHE
EMIT SC_GO_2_T
EXIT AWAIT_CASE_1,A21
AC_5:
EMIT SC_CACHE
EMIT SC_GO_1_S
EXIT AWAIT_CASE_1,A21
AWAIT_CASE_1:
A12:
A2:
GOTO A0
```

The *strl2kasm* compiler tries to do its best by avoiding nested aborts and using trap/exit combinations instead. However, due to the rather awkward Esterel code, there is not much to save in performance in this example. The listing (opcode) size amounts to 104 instructions for the *strl2kasm* code, whereas smakc! produces only 64. This small example shows that SSMs with highly interconnected states produce bad code when compiled to Esterel. However, in real-life applications, such extreme interconnection is quite unlikely, since programs usually follow a control flow which is somewhat linear.

8.3.6. Summary of the comparison

In all Esterel based examples, the *strl2kasm* compiler by Boldt produced more efficient code than smakc!. When starting from state machines, it was often more beneficial to use smakc!, a compiler especially devised for SSMs. However, the gain was quite small except for an occasional anomaly. This is mainly due to the target ISA, which was especially designed for Esterel. The transformation of state machines into KASM code is, after all, a transformation into an Esterel-like code. This makes the produced code very similar in most cases.

The "parhierarchy" example which caused *strl2kasm* to produce more code than smakc! was in fact an example especially geared to translate badly into Esterel (a combination of cyclic control flow and hierarchy, mixed with parallelism produces large code). This is also the punch line of the comparison: the story of fooling *strl2kasm* into producing bad KASM code is the story of fooling Esterel Studio into producing bad Esterel code.

Saving watchers and threadids is very similar. The missing watcher id saving could probably also be quickly implemented in *strl2kasm*.

Reaction times are not significantly different on average, and except for some weird anomalies *strl2kasm* performs slightly better than smakc!.

9. Conclusion and outlook

9.1. Conclusion

In our work we have designed and implemented a compiler to directly translate SSMs to KASM, the assembler language of the KEP, a reactive processor. This was already possible by first translating a SSM to the Esterel language and then using an existing compiler to transform Esterel code to the KEP ISA. Directly transforming SSMs seemed to be a promising approach, as the KEP, just as most other processor architectures, supports a `goto` statement, by which it is very easy to express transitions of a state machine.

The resulting compiler was tested against taking the other two transformations in series as described above. The experiments show that the direct translation is more efficient in most cases; however only very marginally. A notable result: the bottleneck of transforming SSMs to the KEP does not lie in KASM code generators, but in SSM to Esterel code generators. If the transformation from SSMs to Esterel were more efficient, the transformation going from SSMs to Esterel to KASM would probably be equally good or even better than the direct compilation. The Esterel language augmentation *Esterel+Goto* [43] might be the solution, but currently we are missing support for this language extension in all stages of KASM generation as well as in verification frameworks.

Examples that make it hard for the Esterel to KASM compiler are in fact examples that make it hard for the SSM to Esterel compiler. A real gain over multistage compilation in contrast to direct compilation of SSMs is only achieved in some rare special cases.

The true advantage of using a direct SSM to target architecture compiler lies in design verification and debugging. The code produced by smakc! is much easier to understand than code generated using Esterel as an intermediate language, as the multistage compilation tends to obfuscate the code. In smakc! generated code, the original states of a state machine can easily be found. For developers, this means that adding debugging code becomes easier.

So, in short, the answer to the question if one should use a direct SSM to target architecture compiler or go via Esterel as intermediate language is: If you are interested in automated verification, easy debugging and recognizing states in the code, then you should use a direct compiler. If you are only interested in code efficiency and don't care what it looks like, then spare yourself the time and stick with Esterel as intermediate language.

9.2. Open questions and problems

We must know. We will know.

David Hilbert, not knowing that the day before Gödel proved that we can't know

Apply Lukoschus' cycle elimination algorithm to SSMs Lukoschus and von Hanxleden [31] devised a method for removing cyclic signal dependencies in Esterel programs known to be constructive. The interesting part: it is a source code transformation. Since Esterel programs and SSMs are equivalent, this transformation could be applied to SSMs as well and would fit in nicely as a smakc! transformation.

Estimated workload: A full semester's worth of a student research project.

Improve abort and thread watcher usage Usage (or rather, reusage) of abort watcher ids and thread ids is not optimal, neither in *strl2kasm* nor in smakc!, quite obviously because no knowledge is used whatsoever about the possibility of two states executing in parallel or not. If two states (semantically) cannot execute in parallel, they can use the same abort watchers. The question about the best possible approximation ratio is even more interesting.

Estimated workload: Seminar paper or student research project.

Improve the signal dependency algorithm So far, the signal dependency algorithm considers all sources and sink of the same signal a dependency if they are concurrent to each other. Just as above, this does not take additional information into account if it is semantically possible for source and sink to also execute concurrently.

Estimated workload: Seminar paper or student research project.

Implement onExit actions smakc!s SSM datastructure supports onExit actions, but they are not processed in code generation. This is due to the extremely difficult semantics. onExit actions are executed whenever a state is exited, by means of an outbound transition or if any containing macrostate is exited or preempted. Let e be an onExit action. Then this could be implemented by adding e to every containing state, and adding a new signal S_e. The new signal shall denote the firing of e in that instant. In assembler code generation, onExit actions would then be implemented as first testing if S_e is present, and if not, executing e and then emitting S_e.

Estimated workload: Implementation work for 4 hours per week for a full semester.

Linker Since SSMs feature a reference macrostate, smakc! was originally supposed to get a linker to be able to do modular compilation. The linker would insert the appropriate state machines for the reference macrostates and perform signal renaming on the interface. Due to time restrictions, this was not implemented yet. Since macrostates can also be textual states, this would allow linking SSMs and finished KASM produced from any source.

Estimated workload: Implementation work for 4 hours per week for a full semester.

Distributed compilation Almost all parts of smakc! can be executed on different machines, as almost all transformations are inherently distributable. For example, a state machine containing three parallel branches could be split into four parts (the three parallel parts and the containing state). The three parallel parts are full state machines on their own already, and the containing macrostate can be made to one by inserting reference macrostates instead. Then all parts could be compiled separately to be put back together by the above-mentioned linker when finished. This would allow compilation of state machines with more than 2500 states (which is approximately the current limit). Such distributed or modular compilation suffers additional complexity from data dependencies between different program parts.

Estimated workload: Implementation work for 8 hours per week for a full semester.

Other model output smakc! can currently read from KIELER's XMI format and from KIEL's kit format. It might be interesting to implement transformations that act as output modules. For example, a state machine transformed by smakc! could be transformed back into kit and then written to secondary output for graphical display (in KIELER for example). smakc! handles data dependencies as (interlevel) transitions internally so this might be a nice way to display dependencies for the user.

Estimated workload: Implementation work for 2 hours per week for a full semester.

KEP3 KASM output The KEP3 has an ALU, in contrast to the KEPe. This enables the compilation of valued signal SSMs however, the smakc! internal data model would also have to be adjusted accordingly (emissions will have to be added, as seen in the KIELER SSM datastructure).

Estimated workload: Implementation work for 2 hours per week for a full semester.

A. smakc! user guide

smakc! requires several other projects to be referenced in the classpath, depending on the features you want to use. Input modules require the respective API classes for loading and processing the format in question, that is, `kieler.ssm.jar` and the Eclipse Ecore API (`emf.common`, `emf.ecore`, `emf.ecore.xmi`, [1]) for KIELER SSMs and `kiel.kit.jar` for the legacy kit format. The `org.fast.utilities.jar` provides classes and algorithms used all throughout smakc!. The `velocity-dep-1.5.jar` contains the current Apache Velocity engine as well as all other packages it depends on. If you are using the jar bundle of smakc!, then `smakc.jar` contains the main compiler. If you wrote your own extensions, for example, enabling more input formats (see this chapter below, Appendix B), do not forget to add those jars to the classpath too.

smakc! supports several arguments, some of which are command line specific and others can only be used when directly calling the compiler API. We will discuss the call arguments in the following two sections.

A.1. Using command line smakc!

If you are using the `smakc` shellscript, you do not need to worry about options. The reasonable options are already set, all you have to provide are the paths to the state machines you wish to compile. If you want to call smakc! directly however, you should know about the most important options. smakc! is located in the launchers package, thus, you have to call `launchers.smakc`.
Options are:

`-if` or `--input-file`: List any number of files here. The files should be accessible by you and they should contain valid state machines in an input format that smakc! understands. It is important to note that smakc! will attempt to load all input SSMs simultaneously, so you should make sure that the Java VM has enough memory.

`-od` or `--output-directory`: Write compiled state machines to the specified directory instead of the source directories. The compiler ignores all arguments after the first one. If this value is not specified, output is written to standard output.

`-f` or `--force`: Causes smakc! to swallow some error messages and continue with compilation. This includes:

Ignores missing, inaccessible or corrupted input state machines and continues with the rest, does not abort compilation if a state machine contains cyclic dependencies and continues with the rest, ignores arguments in excess of one in negation operators instead of aborting (uses only the first one).

`-v` or `--verbose`: Produces more output (if you really want to see what is going on in smakc! during compilation).

`-tr` or `--apply-transformations`: Provide a list of classes implementing the `Transformation` interface, writing the class names only. The transformations will be loaded by reflection from `smakc.compiler.transformations.<name>`, where `<name>` is the class name of the transformation. The compiler does not check if the transformations are loadable beforehand and will abort compilation with an error message if the reflection loading fails. Transformations have to be given in function call notation, so calling smakc! with `-tr A B C` will apply transformation C first, then B, then A. Compilation will also fail if a transformation does not get the correct input class piped from the previous one.

`-ta` or `--target-architecture`: Provide a list of paths to code templates. The paths will automatically be searched in the templates subdirectory in the classpath. Code will be generated for each target architecture specified.

A.2. Using the smakc! API

Using the API is somewhat different from using the command line. The main class is not the `launchers.smakc` class, it is the smakc.compiler.Compiler class. The Compiler class takes only two arguments: a `java.lang.Collection<State>`, a set of input SSMs, and a `java.util.Properties` as configuration. The configuration holds additional call arguments, and is handed through every transformation by the compiler, so transformations that alter the configuration also indirectly influence the subsequent transformations.

smakc! is generally much more flexible when configured through the API.

`-f` or `--force`,

`-v` or `--verbose`,

`-od` or `--output-directory`,

`-tr` or `--apply-transformations`: The same as on the command line.

return value: The `compile` method returns a `Collection` of `Object`s representing the outcome of all transformations applied to the input state machines.

`-o2` **or** `--secondary-output`: Some transformations produce secondary output, for example, the cycle detection (it displays the states on a dependency cycle). This secondary output is written to the streams found in the provided Map for that SSM. Thus, this field should contain a `Map<String, OutputStream>`.

The streams default to standard output if nothing is specified.

`-ol` **or** `--logging-output`: Same as the secondary output map. The streams found here will be used for logging output. Defaults to standard error output.

B. smakc! developer guide

Extending smakc! is as easy as copying files, thanks to the reflection-loading and templating mechanism.

B.1. Adding more input formats

Allowing smakc! to process a new format for state machines requires you to implement smakc!s SSM interfaces. Some of these interfaces are actually abstract classes, enforcing certain rules, for example: state and signed names must be unique.

A new state machine format just needs to be wrapped into classes capable of accessing new format on the one side and implementing smakc!s SSM format on the other (which is why such classes are called wrapper classes). Once you have all of those classes, simply add them to the package smakc.statemachineproviders.<file extension>. When launching smakc! be sure to have the original foreign format classes and your wrapper class in the classpath.

In either case, the most important thing is to implement the smakc.statemachineproviders.StateMachineLoader interface. The implementation must be called LoaderImpl, and must be located at smakc.statemachineproviders.<file extension>.LoaderImpl. You do not need to modify smakc! code. Simply add your own code to the classpath.

B.2. Adding more output formats

This is even easier than adding more input or transformations. Just create a new set of Velocity templates in a path listed on the smakc! classpath. You must have a template that starts with the word start. Code generation will begin with that template. See chapter 7 and [35] for full Velocity documentation. Velocity uses contexts for passing on data through the templating process and to subtemplates. The CodeWriter transformation initializes the context with the field "State" set to the root state of the statemachine and the "ContainmentTree" is set to contain a special datastructure, the state containment tree, and adds ListGenerator, MapGenerator and StackGenerator factories into fields of the same name. These factories generate List, Map and Stack classes respectively, allowing the use of any amount of these datastructures during the templating process. The start template should be used for initialization only.

The ContainmentTree datastructure was especially created for efficiency reasons in smakc!. Basically, it is a simple tree structure in which each state points to a containing macrostate. However, the ContainmentTree can present a view of itself as a Collection of states. If a certain state is specified, the ContainmentTree can be accessed as a collection of the states on the path from the specified state up to the root state. Accessing the state containment path multiple times becomes more efficient this way, since the ContainmentTree generates the Collection view by internal tricks and does not initialize new Collections every time. This of course causes problems when accessing the datastructure from more than one concurrent Java thread.

To make smakc! use your set of templates, specify the path to them with the `-ta` compiler option.

B.3. Adding transformations

Adding your own transformations is similar to adding input formats. You have to implement an interface (this time *smakc.interfaces.compiler.Transformation*) and your transformation must be located at `smakc.compiler.transformations.<name>`. Implementing the transformation interface is very straightforward. Depending on what types you provided when you wrote the class declaration (let us say you wrote `MyTrans implements Transformation <StateA, StateB>`) you have to transform an object of type `StateB` to an object of type `StateA`. The only other method is the configure method, inherited from Configurable. Here, the compiler provides you with the configuration that should be used.

Please keep in mind that for reasons of coding style and abstraction you should never assume more information about the input state machines than is given by the interface.

C. Esterel examples used in the two-way compare

ABRO The most famous Esterel example.

```
module ABRO:

input A, B, R;

output O;

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

end module
```

smakc! KASM

```
%%% ---BEGIN KEP CODE---
INPUT A
INPUT B
INPUT R
OUTPUT O
EMIT _TICKLEN,#0
BEGINSTARTUPSTATEMENTLIST39STATE:
ABORT R, ENDABORT_STATEMENTLIST39STATERSTATEMENTLIST39STATE_P1, 1
PAR 1, BEGINSTARTUP157, 1
PAR 1, BEGINSTARTUP191, 2
PARE SUBSTATESENDPARALLELSTATEMENTLIST40STATE, 0
BEGINSTARTUP157:
AWAIT A
GOTO BEGINSTARTUP161
BEGINSTARTUP161:
GOTO SUBSTATESENDPARALLELSTATEMENTLIST40STATE
BEGINSTARTUP191:
AWAIT B
GOTO BEGINSTARTUP195
BEGINSTARTUP195:
GOTO SUBSTATESENDPARALLELSTATEMENTLIST40STATE
SUBSTATESENDPARALLELSTATEMENTLIST40STATE:
JOIN 1
EMIT O
GOTO BEGINSTARTUP234
BEGINSTARTUP234:
GOTO SUBSTATESENDSTATEMENTLIST39STATE
SUBSTATESENDSTATEMENTLIST39STATE:
HALT
ENDABORT_STATEMENTLIST39STATERSTATEMENTLIST39STATE_P1:
GOTO BEGINSTARTUPSTATEMENTLIST39STATE
HALT
%%% ---END KEP CODE---
```

```
%%% Esterel Module: ABRO

%%%−−−−−I/O SIGNALS−−−−−
INPUT A,B,R
OUTPUT O
%%%−−−−−INTERFACE STATEMENTS−−−−−
EMIT _TICKLEN,#11

A0:
ABORT R,A1
PAR 1,A2,1
PAR 1,A3,2
PARE A4,1
A2:
AWAIT A
A3:
AWAIT B
A4:
JOIN 0
EMIT O
HALT
A1:
GOTO A0
```

Absync An example of code generated by EStudio (and later optimized by hand).

```
module absync:
input A    ;
output AB   ;
input B    ;
input P    ;
input R    ;

signal  Disarm
in
 nothing;
 loop
  % state ABSync
  abort
   weak abort
    signal  Arm
    in
     [
      nothing;
      % state  idle
      await case  [Arm] do
      nothing;
      % state  Counting
      suspend
       nothing;
       loop
        % state  count
        await case 2  [ tick ]  do
        emit Disarm
        end await
       end loop
      when [P]
      end await
     ||
      nothing;
      % state  WaitAB
      [
       nothing;
       % state  wA
       await case  [A] do
       emit Arm
       end await
      ||
       nothing;
       % state  wB
       await case  [B] do
       emit Arm
       end await
      ];
      emit AB;
      % state  done
      halt
     ]
    end signal
   when [Disarm]
   do
    nothing
```

```
      end weak abort
    when [R]
    do
      nothing
    end abort
  end loop
end signal
end module
```

smakc! KASM

```
%%% ———BEGIN KEP CODE———
INPUT A
INPUT B
INPUT P
INPUT R
OUTPUT AB
EMIT _TICKLEN,#0
SIGNAL DISARM
BEGINSTARTUPABORT49STATE:
SIGNAL ARM
ABORT R, ENDABORT_ABORT49STATERABORT49STATE_P1, 1
ABORT DISARM, ENDABORT_PARALLELSTATEMENTLIST55STATEDISARM9605_P1, 2
PAR 2, BEGINSTARTUP9201, 1
PAR 1, BEGINSTARTUPPARALLELSTATEMENTLIST77STATE, 2
PARE SUBSTATESENDPARALLELSTATEMENTLIST55STATE, 0
BEGINSTARTUP9201:
AWAIT ARM
GOTO BEGINSTARTUPSTATEMENTLIST63STATE
BEGINSTARTUPSTATEMENTLIST63STATE:
SUSPEND P, ENDSUSPENDSTATEMENTLIST63STATE
BEGINSTARTUP9247:
LOAD _COUNT,#2
AWAIT TICK
GOTO BEGINSTARTUP9247
HALT
ENDSUSPENDSTATEMENTLIST63STATE:
BEGINSTARTUPPARALLELSTATEMENTLIST77STATE:
PAR 1, BEGINSTARTUP9406, 3
PAR 1, BEGINSTARTUP9480, 4
PARE SUBSTATESENDPARALLELSTATEMENTLIST77STATE, 0
BEGINSTARTUP9406:
AWAIT A
GOTO BEGINSTARTUP9411
BEGINSTARTUP9411:
GOTO SUBSTATESENDPARALLELSTATEMENTLIST77STATE
BEGINSTARTUP9480:
AWAIT B
GOTO BEGINSTARTUP9485
BEGINSTARTUP9485:
GOTO SUBSTATESENDPARALLELSTATEMENTLIST77STATE
SUBSTATESENDPARALLELSTATEMENTLIST77STATE:
JOIN 1
EMIT AB
GOTO BEGINSTARTUP9563
BEGINSTARTUP9563:
HALT
SUBSTATESENDPARALLELSTATEMENTLIST55STATE:
JOIN 1
HALT
ENDABORT_PARALLELSTATEMENTLIST55STATEDISARM9605_P1:
GOTO BEGINSTARTUP9605
BEGINSTARTUP9605:
GOTO SUBSTATESENDABORT49STATE
SUBSTATESENDABORT49STATE:
GOTO BEGINSTARTUPABORT49STATE
ENDABORT_ABORT49STATERABORT49STATE_P1:
```

```
GOTO BEGINSTARTUPABORT49STATE
HALT
HALT
%%% ---END KEP CODE---
```

strl2kasm KASM

```
%%% Esterel Module: absync

%%%-----I/O SIGNALS-----
INPUT A,B,P,R
OUTPUT AB
%%%-----TOP LOCAL SIGNALS-----
SIGNAL DISARM
%%%-----INTERFACE STATEMENTS-----
EMIT _TICKLEN,#0

NOTHING
A0:
ABORT R,A1
WABORT DISARM,A3
SIGNAL ARM
PAR 1,A5,1
PAR 1,A6,2
PARE A7,1
A5:
NOTHING
AWAIT ARM
SUSPEND P,A10
NOTHING
A11:
LOAD _COUNT,#2
ABORT TICK,A12
A13:
PRIO 2
PAUSE
PRIO 1
GOTO A13
A12:
EMIT DISARM
PRIO 1
GOTO A11
A10:
NOTHING
A6:
NOTHING
PAR 1,A14,3
PAR 1,A15,4
PARE A16,1
A14:
NOTHING
AWAIT A
EMIT ARM
A15:
NOTHING
AWAIT B
EMIT ARM
A16:
JOIN 0
EMIT AB
HALT
A7:
JOIN 0
```

```
A3:
NOTHING
A4:
GOTO A2
A1:
NOTHING
A2:
GOTO A0
```

Runner Another (in-)famous example for pure Esterel.

```
module RUNNER :

input METER;
input SECOND;
input MORNING;
input LAP;
input STEP;
input HEART_BEAT;

output WALK;
output RUN;
output JUMP;
output GO_TO_WORK;
output GO_TO_HOSPITAL;

relation SECOND # METER # STEP;
relation MORNING => SECOND;
relation LAP => METER;

trap HEART_ATTACK in
    every MORNING do
        do
            loop
                do
                    emit WALK
                upto 100 METER;
                signal HEART_ATTACK in
                    do
                        do
                            every STEP do emit JUMP end
                        upto 15 SECOND;
                        emit RUN;
                    watching HEART_ATTACK timeout exit HEART_ATTACK end
                ||
                    copymodule CHECK_HEART
                end
            each LAP
        upto 2 LAP;
        emit GO_TO_WORK
    end
handle HEART_ATTACK do
    emit GO_TO_HOSPITAL
end
.

module CHECK_HEART :
input SECOND;
input HEART_BEAT;
output HEART_ATTACK;

loop
    await SECOND do emit HEART_ATTACK end
each HEART_BEAT
.
```

smakc! KASM

```
%%% ---BEGIN KEP CODE---
INPUT METER
INPUT SECOND
INPUT MORNING
INPUT LAP
INPUT STEP
INPUT HEART_BEAT
OUTPUT WALK
OUTPUT RUN
OUTPUT JUMP
OUTPUT GO_TO_WORK
OUTPUT GO_TO_HOSPITAL
EMIT _TICKLEN,#0
SIGNAL HEART_ATTACK_
SIGNAL HALTTRAP51_
WABORTI HALTTRAP51_, ENDABORT_EVERY52STATEHALTTRAP51_21102_P1, 1
WABORTI HEART_ATTACK_, ENDABORT_EVERY52STATEHEART_ATTACK_21476_P2, 2
AWAIT MORNING
GOTO BEGINSTARTUPSTATEMENTLIST53STATE
BEGINSTARTUPSTATEMENTLIST53STATE:
ABORT MORNING, ENDABORT_STATEMENTLIST53STATEMORNINGSTATEMENTLIST53STATE_P1, 3
LOAD _COUNT,#2
ABORT LAP, ENDABORT_LOOPEACH55STATELAP21425_P1, 4
BEGINSTARTUPSTATEMENTLIST56STATE:
SIGNAL HEART_ATTACK
ABORT LAP, ENDABORT_STATEMENTLIST56STATELAPSTATEMENTLIST56STATE_P1, 5
LOAD _COUNT,#100
AWAIT METER
GOTO BEGINSTARTUPPARALLELSTATEMENTLIST64STATE
BEGINSTARTUPPARALLELSTATEMENTLIST64STATE:
PAR 2, BEGINSTARTUPSTATEMENTLIST66STATE, 1
PAR 1, BEGINSTARTUPAWAIT80STATE, 2
PARE SUBSTATESENDPARALLELSTATEMENTLIST64STATE, 0
BEGINSTARTUPSTATEMENTLIST66STATE:
ABORT HEART_ATTACK, ENDABORT_STATEMENTLIST66STATEHEART_ATTACK21266_P1, 6
LOAD _COUNT,#15
ABORT SECOND, ENDABORT_EVERY68STATESECOND21242_P1, 7
AWAIT STEP
GOTO BEGINSTARTUPINIT21186
BEGINSTARTUPINIT21186:
PRESENT TICK, ENDSIGNALTEST_TICK_INIT21186
EMIT JUMP
GOTO BEGINSTARTUP21186
ENDSIGNALTEST_TICK_INIT21186:
HALT
BEGINSTARTUP21186:
AWAIT STEP
GOTO BEGINSTARTUPINIT21186
HALT
ENDABORT_EVERY68STATESECOND21242_P1:
EMIT RUN
GOTO BEGINSTARTUP21242
BEGINSTARTUP21242:
GOTO SUBSTATESENDSTATEMENTLIST66STATE
SUBSTATESENDSTATEMENTLIST66STATE:
GOTO BEGINSTARTUP21170
```

```
ENDABORT_STATEMENTLIST66STATEHEART_ATTACK21266_P1:
EMIT HEART_ATTACK_
GOTO BEGINSTARTUP21266
BEGINSTARTUP21170:
GOTO SUBSTATESENDPARALLELSTATEMENTLIST64STATE
BEGINSTARTUP21266:
HALT
BEGINSTARTUPAWAIT80STATE:
ABORT HEART_BEAT, ENDABORT_AWAIT80STATEHEART_BEATAWAIT80STATE_P1, 8
AWAIT SECOND
GOTO BEGINSTARTUP21317
BEGINSTARTUP21317:
GOTO SUBSTATESENDAWAIT80STATE
SUBSTATESENDAWAIT80STATE:
HALT
ENDABORT_AWAIT80STATEHEART_BEATAWAIT80STATE_P1:
GOTO BEGINSTARTUPAWAIT80STATE
SUBSTATESENDPARALLELSTATEMENTLIST64STATE:
JOIN 1
HALT
HALT
ENDABORT_STATEMENTLIST56STATELAPSTATEMENTLIST56STATE_P1:
GOTO BEGINSTARTUPSTATEMENTLIST56STATE
HALT
ENDABORT_LOOPEACH55STATELAP21425_P1:
EMIT GO_TO_WORK
GOTO BEGINSTARTUP21425
BEGINSTARTUP21425:
GOTO SUBSTATESENDSTATEMENTLIST53STATE
SUBSTATESENDSTATEMENTLIST53STATE:
HALT
ENDABORT_STATEMENTLIST53STATEMORNINGSTATEMENTLIST53STATE_P1:
GOTO BEGINSTARTUPSTATEMENTLIST53STATE
HALT
ENDABORT_EVERY52STATEHEART_ATTACK_21476_P2:
EMIT GO_TO_HOSPITAL
GOTO BEGINSTARTUP21476
ENDABORT_EVERY52STATEHALTTRAP51_21102_P1:
GOTO BEGINSTARTUP21102
BEGINSTARTUP21102:
HALT
BEGINSTARTUP21476:
GOTO SUBSTATESENDMODULE_RUNNER
SUBSTATESENDMODULE_RUNNER:
HALT
%%% ---END KEP CODE---
```

strl2kasm KASM

```
%%% Esterel Module: RUNNER

%%%-----I/O SIGNALS-----
INPUT METER,SECOND,MORNING,LAP,STEP,HEART_BEAT
OUTPUT WALK,RUN,JUMP,GO_TO_WORK,GO_TO_HOSPITAL
%%%-----INTERFACE STATEMENTS-----
EMIT _TICKLEN,#18

A0:
AWAIT MORNING
A3:
ABORT MORNING,A4
LOAD _COUNT,#2
ABORT LAP,A5
A6:
ABORT LAP,A7
LOAD _COUNT,#100
ABORT METER,A8
EMIT WALK
HALT
A8:
SIGNAL HEART_ATTACK_0
PAR 1,A10,1
PAR 1,A11,2
PARE A12,1
A10:
ABORT HEART_ATTACK_0,A13
LOAD _COUNT,#15
ABORT SECOND,A14
ABORT STEP,A15
A16:
PRIO 2
PAUSE
PRIO 1
GOTO A16
A15:
A17:
PRIO 1
ABORT STEP,A18
EMIT JUMP
A19:
PRIO 2
PAUSE
PRIO 1
GOTO A19
A18:
PRIO 1
GOTO A17
A14:
EMIT RUN
A13:
EXIT HEART_ATTACK,A0
A11:
A20:
ABORT HEART_BEAT,A21
ABORT SECOND,A22
```

```
A23:
PRIO 2
PAUSE
PRIO 1
GOTO A23
A22:
EMIT HEART_ATTACK_0
A24:
PRIO 1
PAUSE
GOTO A24
A21:
PRIO 1
GOTO A20
A12:
JOIN 0
A7:
GOTO A6
A5:
EMIT GO_TO_WORK
HALT
A4:
GOTO A3
HEART_ATTACK:
EMIT GO_TO_HOSPITAL
HALT
```

test-present7 A simple program testing present statements.

```
module test_present7:
input A;
input B;
input C;
output W;
output X;
output Y;
output Z;

loop
  present A then
    pause
  end present;
  present B else
    emit W
  end present;
  present
    case A do
      pause;
      emit X
    case B
    case C do
      emit Y
  end present;
  present A then
    present B then
      pause
    else
      emit Z
    end present;
    present C else
      pause
    end present
  else
    present
      case A
      case B do
        emit X
      case C
    end present
  end present;
  pause
end loop

end module
```

smakc! KASM

```
%%% ---BEGIN KEP CODE---
INPUT A
INPUT B
INPUT C
OUTPUT W
OUTPUT X
OUTPUT Y
OUTPUT Z
EMIT _TICKLEN,#0
BEGINSTARTUP582:
PRESENT A, ENDSIGNALTEST_A_582
GOTO BEGINSTARTUP584
ENDSIGNALTEST_A_582:
PRESENT TICK, ENDSIGNALTEST_TICK_582
GOTO BEGINSTARTUP636
ENDSIGNALTEST_TICK_582:
HALT
BEGINSTARTUP584:
AWAIT TICK
GOTO BEGINSTARTUP636
BEGINSTARTUP636:
PRESENT B, ENDSIGNALTEST_B_636
GOTO BEGINSTARTUP682
ENDSIGNALTEST_B_636:
PRESENT TICK, ENDSIGNALTEST_TICK_636
EMIT W
GOTO BEGINSTARTUP682
ENDSIGNALTEST_TICK_636:
HALT
BEGINSTARTUP682:
PRESENT A, ENDSIGNALTEST_A_682
GOTO BEGINSTARTUP685
ENDSIGNALTEST_A_682:
PRESENT B, ENDSIGNALTEST_B_682
GOTO BEGINSTARTUP797
ENDSIGNALTEST_B_682:
PRESENT C, ENDSIGNALTEST_C_682
EMIT Y
GOTO BEGINSTARTUP797
ENDSIGNALTEST_C_682:
PRESENT TICK, ENDSIGNALTEST_TICK_682
GOTO BEGINSTARTUP797
ENDSIGNALTEST_TICK_682:
HALT
BEGINSTARTUP685:
AWAIT TICK
GOTO BEGINSTARTUP797
BEGINSTARTUP797:
PRESENT A, ENDSIGNALTEST_A_797
GOTO BEGINSTARTUP800
ENDSIGNALTEST_A_797:
PRESENT TICK, ENDSIGNALTEST_TICK_797
GOTO BEGINSTARTUP929
ENDSIGNALTEST_TICK_797:
HALT
BEGINSTARTUP800:
```

```
PRESENT B, ENDSIGNALTEST_B_800
GOTO BEGINSTARTUP802
ENDSIGNALTEST_B_800:
PRESENT TICK, ENDSIGNALTEST_TICK_800
EMIT Z
GOTO BEGINSTARTUP865
ENDSIGNALTEST_TICK_800:
HALT
BEGINSTARTUP802:
AWAIT TICK
GOTO BEGINSTARTUP865
BEGINSTARTUP865:
PRESENT C, ENDSIGNALTEST_C_865
GOTO BEGINSTARTUP930
ENDSIGNALTEST_C_865:
PRESENT TICK, ENDSIGNALTEST_TICK_865
GOTO BEGINSTARTUP877
ENDSIGNALTEST_TICK_865:
HALT
BEGINSTARTUP930:
AWAIT TICK
GOTO BEGINSTARTUP582
BEGINSTARTUP877:
AWAIT TICK
GOTO BEGINSTARTUP930
BEGINSTARTUP929:
PRESENT A, ENDSIGNALTEST_A_929
GOTO BEGINSTARTUP930
ENDSIGNALTEST_A_929:
PRESENT B, ENDSIGNALTEST_B_929
EMIT X
GOTO BEGINSTARTUP930
ENDSIGNALTEST_B_929:
PRESENT C, ENDSIGNALTEST_C_929
GOTO BEGINSTARTUP930
ENDSIGNALTEST_C_929:
PRESENT TICK, ENDSIGNALTEST_TICK_929
GOTO BEGINSTARTUP930
ENDSIGNALTEST_TICK_929:
HALT
HALT
%%% ---END KEP CODE---
```

strl2kasm KASM

```
%%% Esterel Module: test_present7

%%%−−−−−I/O SIGNALS−−−−−
INPUT A,B,C
OUTPUT W,X,Y,Z
%%%−−−−−INTERFACE STATEMENTS−−−−−
EMIT _TICKLEN,#16

A0:
PRESENT A,A1
PAUSE
A1:
PRESENT B,A2
GOTO A3
A2:
EMIT W
A3:
PRESENT A,A4
PAUSE
EMIT X
GOTO A5
A4:
PRESENT B,A6
GOTO A7
A6:
PRESENT C,A8
EMIT Y
A8:
A7:
A5:
PRESENT A,A9
PRESENT B,A11
PAUSE
GOTO A12
A11:
EMIT Z
A12:
PRESENT C,A13
GOTO A14
A13:
PAUSE
A14:
GOTO A10
A9:
PRESENT A,A15
GOTO A16
A15:
PRESENT B,A17
EMIT X
GOTO A18
A17:
A18:
A16:
A10:
PAUSE
GOTO A0
```

savethreadids A program to test if thread ids are saved by compilers.

```
module savethreadids:
input A;
input B;
  [
   await A;
   ||
   await B;
  ];
  [
   await A;
   ||
   await B;
  ];
end module
```

smakc! KASM

```
%%% ---BEGIN KEP CODE---
INPUT A
INPUT B
EMIT _TICKLEN,#0
PAR 1, BEGINSTARTUP17STATE.2, 1
PAR 1, BEGINSTARTUP25STATE.4, 2
PARE SUBSTATESEND4STATE.1, 0
BEGINSTARTUP17STATE.2:
AWAIT A
GOTO BEGINSTARTUP21STATE.3
BEGINSTARTUP21STATE.3:
GOTO SUBSTATESEND4STATE.1
BEGINSTARTUP25STATE.4:
AWAIT B
GOTO BEGINSTARTUP29STATE.5
BEGINSTARTUP29STATE.5:
GOTO SUBSTATESEND4STATE.1
SUBSTATESEND4STATE.1:
JOIN 1
GOTO BEGINSTARTUP56STATE.6
BEGINSTARTUP56STATE.6:
PAR 1, BEGINSTARTUP71STATE.7, 1
PAR 1, BEGINSTARTUP79STATE.9, 2
PARE SUBSTATESEND56STATE.6, 0
BEGINSTARTUP71STATE.7:
AWAIT A
GOTO BEGINSTARTUP75STATE.8
BEGINSTARTUP75STATE.8:
GOTO SUBSTATESEND56STATE.6
BEGINSTARTUP79STATE.9:
AWAIT B
GOTO BEGINSTARTUP83STATE.10
BEGINSTARTUP83STATE.10:
GOTO SUBSTATESEND56STATE.6
SUBSTATESEND56STATE.6:
JOIN 1
HALT
HALT
%%% ---END KEP CODE---
```

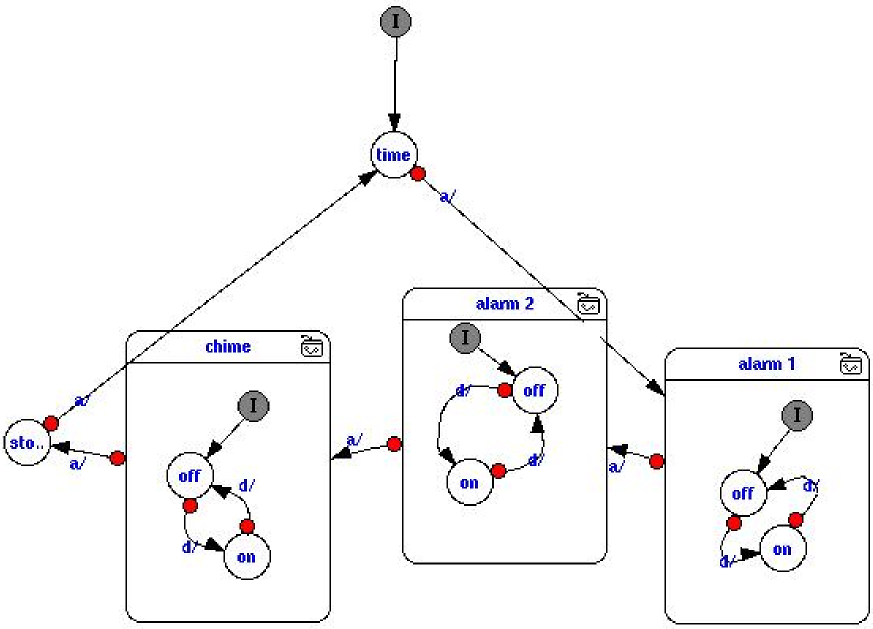
strl2kasm KASM

```
%%% Esterel Module: savethreadids

%%%-----I/O SIGNALS-----
INPUT A,B
%%% ERROR: NO OUTPUT SIGNALS, DEFINE DUMMY:
OUTPUT _NO_OUTPUT_PORT_ERROR
%%%-----INTERFACE STATEMENTS-----
EMIT _TICKLEN,#9

PAR 1,A0,1
PAR 1,A1,2
PARE A2,1
A0:
AWAIT A
A1:
AWAIT B
A2:
JOIN 0
PAR 1,A7,1
PAR 1,A8,2
PARE A9,1
A7:
AWAIT A
A8:
AWAIT B
A9:
JOIN 0
HALT
```

D. SSM examples used in the two-way compare

The SSMs for the two-way comparison were created with Esterel Studio v5. Esterel Studio was also used to export them to Esterel. The KIEL tool, on the other hand, transformed Esterel Studio scg SSMs to the kit format for processing by smakc!.

"displays" A simple three-part SSM.

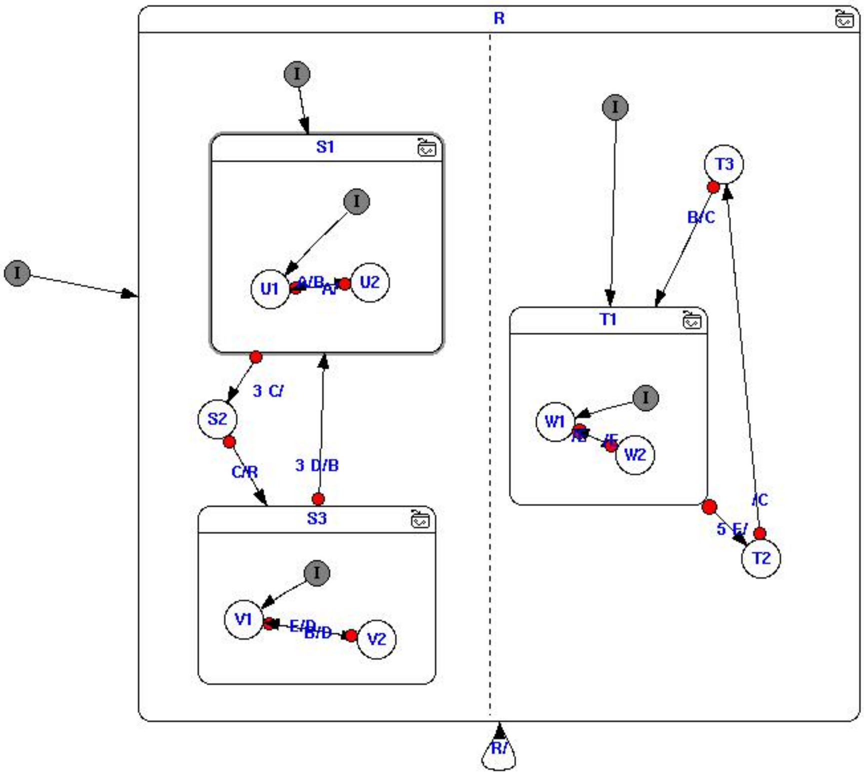


smakc! KASM

```
%%% ---BEGIN KEP CODE---
INPUT A
INPUT D
EMIT _TICKLEN,#0
BEGINSTARTUPTIME:
AWAIT A
GOTO BEGINSTARTUP1042STATE.5
BEGINSTARTUP1042STATE.5:
ABORT A, ENDABORT_1042STATE.5A1015STATE.4_P1, 1
BEGINSTARTUPOFF:
LOAD _COUNT,#2
AWAIT D
GOTO BEGINSTARTUPOFF
HALT
ENDABORT_1042STATE.5A1015STATE.4_P1:
GOTO BEGINSTARTUP1015STATE.4
BEGINSTARTUP1015STATE.4:
ABORT A, ENDABORT_1015STATE.4ACHIME_P1, 2
BEGINSTARTUP1019STATE.8:
LOAD _COUNT,#2
AWAIT D
GOTO BEGINSTARTUP1019STATE.8
HALT
ENDABORT_1015STATE.4ACHIME_P1:
GOTO BEGINSTARTUPCHIME
BEGINSTARTUPCHIME:
LOAD _COUNT,#2
ABORT A, ENDABORT_CHIMEATIME_P1, 3
BEGINSTARTUPOFF1:
LOAD _COUNT,#2
AWAIT D
GOTO BEGINSTARTUPOFF1
HALT
ENDABORT_CHIMEATIME_P1:
GOTO BEGINSTARTUPTIME
HALT
%%% ---END KEP CODE---
```

```
%%% Esterel Module: displays

%%%−−−−−I/O SIGNALS−−−−−
INPUT A,D
%%% ERROR: NO OUTPUT SIGNALS, DEFINE DUMMY:
OUTPUT _NO_OUTPUT_PORT_ERROR
%%%−−−−−TOP LOCAL SIGNALS−−−−−
SIGNAL SC_CACHE
%%%−−−−−INTERFACE STATEMENTS−−−−−
EMIT _TICKLEN,#5

A0:
AWAIT A
ABORT A,A3
A5:
AWAIT D
AWAIT D
GOTO A5
A3:
ABORT A,A10
A12:
AWAIT D
AWAIT D
GOTO A12
A10:
ABORT A,A17
A19:
AWAIT D
AWAIT D
GOTO A19
A17:
AWAIT A
A18:
A11:
A4:
GOTO A0
```

"parhierarchy" A state machine especially designed to exploit the weaknesses of Esterel code generation.



smakc! KASM

```
%%% ---BEGIN KEP CODE---
INPUT A
INPUT B
INPUT C
INPUT D
INPUT E
INPUT R
EMIT _TICKLEN,#0
BEGINSTARTUPR:
WABORT R, ENDABORT_RRR_P1, 1
PAR 3, BEGINSTARTUPS1, 1
PAR 1, BEGINSTARTUPT1, 2
PARE SUBSTATESENDR, 0
BEGINSTARTUPS1:
LOAD _COUNT,#4
ABORT C, ENDABORT_S1CS3_P1, 2
BEGINSTARTUPU1:
LOAD _COUNT,#2
AWAIT A
GOTO BEGINSTARTUPU1
HALT
ENDABORT_S1CS3_P1:
EMIT R
GOTO BEGINSTARTUPS3
BEGINSTARTUPS3:
LOAD _COUNT,#3
ABORT D, ENDABORT_S3DS1_P1, 3
BEGINSTARTUPV1:
AWAIT E
GOTO BEGINSTARTUPV2
BEGINSTARTUPV2:
AWAIT B
GOTO BEGINSTARTUPV1
HALT
ENDABORT_S3DS1_P1:
EMIT B
GOTO BEGINSTARTUPS1
BEGINSTARTUPT1:
LOAD _COUNT,#5
ABORT E, ENDABORT_T1ET2_P1, 4
BEGINSTARTUPW1:
AWAIT TICK
GOTO BEGINSTARTUPW2
BEGINSTARTUPW2:
AWAIT TICK
GOTO BEGINSTARTUPW1
HALT
ENDABORT_T1ET2_P1:
GOTO BEGINSTARTUPT2
BEGINSTARTUPT2:
AWAIT TICK
GOTO BEGINSTARTUPT3
BEGINSTARTUPT3:
AWAIT B
GOTO BEGINSTARTUPT1
SUBSTATESENDR:
```

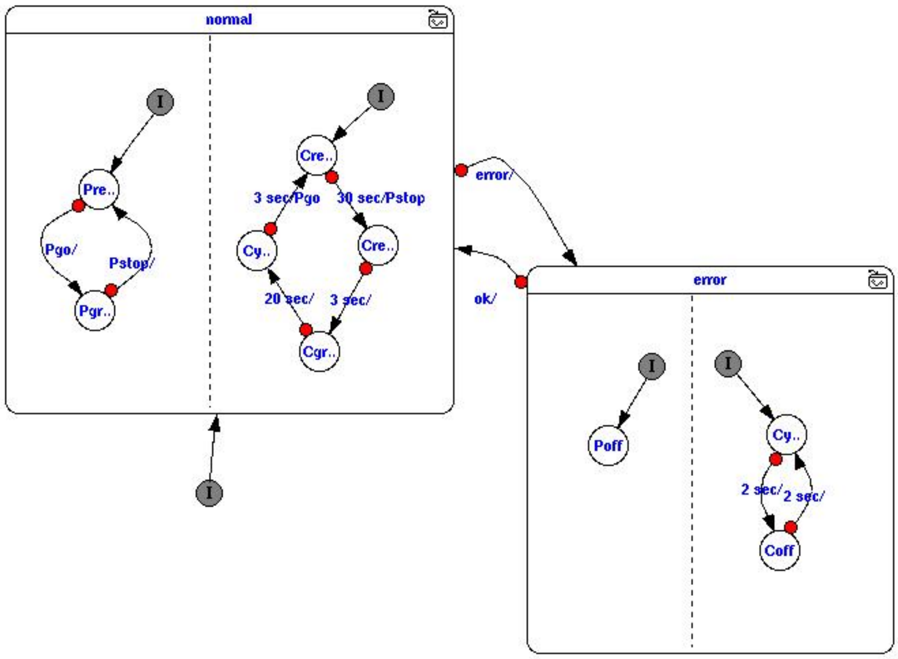
```
JOIN 1
HALT
ENDABORT_RRR_P1:
GOTO BEGINSTARTUPR
HALT
%%% ---END KEP CODE---
```

strl2kasm KASM

```
%%% Esterel Module: parhierarchy

%%%−−−−−I/O SIGNALS−−−−−
INPUT A,B,C,D,E,R
%%% ERROR: NO OUTPUT SIGNALS, DEFINE DUMMY:
OUTPUT _NO_OUTPUT_PORT_ERROR
%%%−−−−−TOP LOCAL SIGNALS−−−−−
SIGNAL SC_CACHE
%%%−−−−−INTERFACE STATEMENTS−−−−−
EMIT _TICKLEN,#0

A0:
WABORT R,A1
PAR 1,A3,1
PAR 1,A4,2
PARE A5,1
A3:
A6:
LOAD _COUNT,#3
ABORT C,A7
A9:
AWAIT A
AWAIT A
EMIT B
GOTO A9
A7:
AWAIT C
EMIT R
LOAD _COUNT,#3
ABORT D,A16
A18:
AWAIT E
EMIT D
AWAIT B
EMIT D
GOTO A18
A16:
EMIT B
A17:
A8:
GOTO A6
A4:
A23:
LOAD _COUNT,#5
ABORT E,A24
A26:
PAUSE
EMIT E
PAUSE
EMIT E
GOTO A26
A24:
PAUSE
EMIT C
AWAIT B
EMIT C
```

```
A25:
GOTO A23
A5:
JOIN 0
A1:
A2:
GOTO A0
```

"Traffic Light" A famous example for a pure SSM.



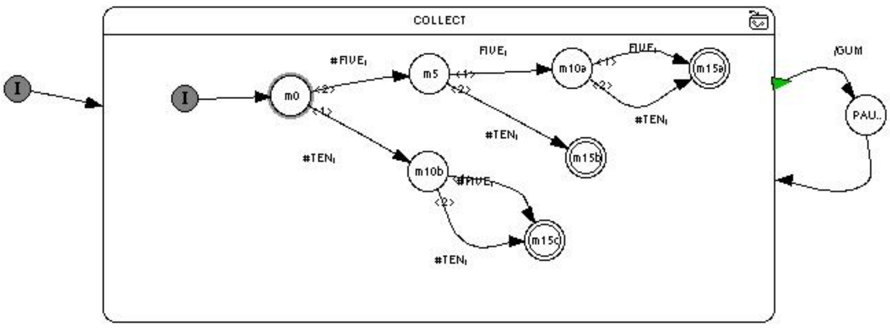
smakc! KASM

```
%%% ---BEGIN KEP CODE---
INPUT SEC
INPUT ERROR
INPUT OK
INPUT PSTOP
INPUT PGO
EMIT _TICKLEN,#0
BEGINSTARTUPNORMAL:
ABORT ERROR, ENDABORT_NORMALERRORERROR_P1, 1
PAR 2, BEGINSTARTUPPRED, 1
PAR 1, BEGINSTARTUPCRED, 2
PARE SUBSTATESENDNORMAL, 0
BEGINSTARTUPPRED:
AWAIT PGO
GOTO BEGINSTARTUPPGREEN
BEGINSTARTUPPGREEN:
AWAIT PSTOP
GOTO BEGINSTARTUPPRED
BEGINSTARTUPCRED:
LOAD _COUNT,#30
AWAIT SEC
GOTO BEGINSTARTUPCREDYEL
BEGINSTARTUPCREDYEL:
LOAD _COUNT,#26
AWAIT SEC
GOTO BEGINSTARTUPCRED
SUBSTATESENDNORMAL:
JOIN 1
HALT
ENDABORT_NORMALERRORERROR_P1:
GOTO BEGINSTARTUPERROR
BEGINSTARTUPERROR:
ABORT OK, ENDABORT_ERROROKNORMAL_P1, 2
PAR 1, BEGINSTARTUPPOFF, 1
PAR 1, BEGINSTARTUPCYELLOW, 2
PARE SUBSTATESENDERROR, 0
BEGINSTARTUPPOFF:
HALT
BEGINSTARTUPCYELLOW:
LOAD _COUNT,#4
AWAIT SEC
GOTO BEGINSTARTUPCYELLOW
SUBSTATESENDERROR:
JOIN 1
HALT
ENDABORT_ERROROKNORMAL_P1:
GOTO BEGINSTARTUPNORMAL
HALT
%%% ---END KEP CODE---
```

strl2kasm KASM

```
%%% Esterel Module: traffic_light

%%%-----I/O SIGNALS-----
INPUT SEC,ERROR,OK,PSTOP,PGO
%%% ERROR: NO OUTPUT SIGNALS, DEFINE DUMMY:
OUTPUT _NO_OUTPUT_PORT_ERROR
%%%-----TOP LOCAL SIGNALS-----
SIGNAL SC_CACHE
%%%-----INTERFACE STATEMENTS-----
EMIT _TICKLEN,#15

A0:
ABORT ERROR,A1
PAR 1,A3,1
PAR 1,A4,2
PARE A5,1
A3:
A6:
AWAIT PGO
AWAIT PSTOP
GOTO A6
A4:
A11:
LOAD _COUNT,#30
AWAIT SEC
EMIT PSTOP
LOAD _COUNT,#3
AWAIT SEC
LOAD _COUNT,#20
AWAIT SEC
LOAD _COUNT,#3
AWAIT SEC
EMIT PGO
GOTO A11
A5:
JOIN 0
A1:
ABORT OK,A20
PAR 1,A22,1
PAR 1,A23,2
PARE A24,1
A22:
HALT
A23:
A26:
LOAD _COUNT,#2
AWAIT SEC
LOAD _COUNT,#2
AWAIT SEC
GOTO A26
A24:
JOIN 0
A20:
A21:
A2:
GOTO A0
```

“VEND_B” A chewing gum vending machine. A gum costs 15 cents, but you don’t get any change returned.



```
%%% ---BEGIN KEP CODE---
INPUT FIVE
INPUT TEN
OUTPUT GUM
EMIT _TICKLEN,#0
BEGINSTARTUPM0:
WABORTI TEN, ENDABORT_M0TENM10B_P1
WABORTI FIVE, ENDABORT_M0FIVEM5_P2
HALT
ENDABORT_M0FIVEM5_P2:
GOTO BEGINSTARTUPM5
ENDABORT_M0TENM10B_P1:
GOTO BEGINSTARTUPM10B
BEGINSTARTUPM10B:
WABORTI FIVE, ENDABORT_M10BFIVEPAUSE_P1
WABORTI TEN, ENDABORT_M10BTENPAUSE_P2
HALT
ENDABORT_M10BTENPAUSE_P2:
EMIT GUM
GOTO BEGINSTARTUPPAUSE
ENDABORT_M10BFIVEPAUSE_P1:
EMIT GUM
GOTO BEGINSTARTUPPAUSE
BEGINSTARTUPPAUSE:
AWAIT TICK
GOTO BEGINSTARTUPM0
BEGINSTARTUPM5:
WABORT FIVE, ENDABORT_M5FIVEM10A_P1
WABORTI TEN, ENDABORT_M5TENPAUSE_P2
HALT
ENDABORT_M5TENPAUSE_P2:
EMIT GUM
GOTO BEGINSTARTUPPAUSE
ENDABORT_M5FIVEM10A_P1:
GOTO BEGINSTARTUPM10A
BEGINSTARTUPM10A:
WABORT FIVE, ENDABORT_M10AFIVEPAUSE_P1
WABORTI TEN, ENDABORT_M10ATENPAUSE_P2
HALT
ENDABORT_M10ATENPAUSE_P2:
EMIT GUM
GOTO BEGINSTARTUPPAUSE
ENDABORT_M10AFIVEPAUSE_P1:
EMIT GUM
GOTO BEGINSTARTUPPAUSE
HALT
%%% ---END KEP CODE---
```

strl2kasm KASM

```
%%% Esterel Module: VEND_B

%%%-----I/O SIGNALS-----
INPUT TEN,FIVE
OUTPUT GUM
%%%-----TOP LOCAL SIGNALS-----
SIGNAL SC_CACHE
%%%-----INTERFACE STATEMENTS-----
EMIT _TICKLEN,#15

A0:
A1:
A2:
A3:
PRESENT TEN,A4
EXIT AC,A2
A4:
PRESENT FIVE,A6
EXIT AC_0,A3
A6:
A7:
PAUSE
PRESENT TEN,A8
EXIT AC,A2
A8:
PRESENT FIVE,A9
EXIT AC_0,A3
A9:
GOTO A7
AC_0:
A10:
A11:
A12:
PRESENT TEN,A13
EXIT AC_2,A12
A13:
A14:
PAUSE
PRESENT FIVE,A15
EXIT AC_1,A11
A15:
PRESENT TEN,A16
EXIT AC_2,A12
A16:
GOTO A14
AC_2:
EXIT AWAIT_CASE_0,A10
AC_1:
A17:
A18:
A19:
PRESENT TEN,A20
EXIT AC_4,A19
A20:
A21:
PAUSE
```

```
PRESENT FIVE,A22
EXIT AC_3,A18
A22:
PRESENT TEN,A23
EXIT AC_4,A19
A23:
GOTO A21
AC_4:
EXIT AWAIT_CASE_1,A17
AC_3:
EXIT AWAIT_CASE_1,A17
AWAIT_CASE_1:
EXIT AWAIT_CASE_0,A10
AWAIT_CASE_0:
EXIT AWAIT_CASE,A1
AC:
A24:
A25:
A26:
PRESENT FIVE,A27
EXIT AC_5,A25
A27:
PRESENT TEN,A29
EXIT AC_6,A26
A29:
A30:
PAUSE
PRESENT FIVE,A31
EXIT AC_5,A25
A31:
PRESENT TEN,A32
EXIT AC_6,A26
A32:
GOTO A30
AC_6:
EXIT AWAIT_CASE_2,A24
AC_5:
EXIT AWAIT_CASE_2,A24
AWAIT_CASE_2:
EXIT AWAIT_CASE,A1
AWAIT_CASE:
EMIT GUM
PAUSE
GOTO A0
```

E. List of acronyms and abbreviations

KEP	Kiel Esterel Processor
smakc!	state machine to KEP compiler
ISA	instruction set architecture
SCC	strongly connected component
CFG	control flow graph
LP	linear problem
DDG	data dependency graph
WCRT	worst case reaction time
SSM	Safe State Machine
KASM	KEP assembler language
FSM	finite state machine
KIEL	Kiel Integrated Environment for Layout
KIELER	KIEL for the Eclipse rich client platform

References

[1] Eclipse modeling framework project (EMF), http://www.eclipse.org/modeling/emf/. 45, 75

[2] Embedded and realtime systems group, University of Kiel: K(r)epevalbench (http://www.informatik.uni-kiel.de/rtsys/kep). 51, 61

[3] KIEL for the Eclipse rich client platform (KIELER), http://www.informatik.uni-kiel.de/rtsys/kieler/. 7, 45

[4] Kiel Integrated Environment for Layout (KIEL), http://rtsys.informatik.uni-kiel.de/ rt-kiel/. 7, 55

[5] Sun Microsystems: Sun Java Real-Time System (http://java.sun.com/javase/technologies/realtime/). 2

[6] Charles André. SyncCharts: A visual representation of reactive behaviors. Technical report, University of Nice-Sophia Antipolis, 1995. 5, 7

[7] Charles André. *Semantics of S.S.M.(Safe State Machine)*. University of Nice-Sophia Antipolis / CNRS, 2003. 3

[8] John Augustine, Sudarshan Banerjee, and Sandy Irani. Strip packing with precedence constraints and strip packing with release times. In *SPAA '06: Proceedings of the eighteenth annual ACM symposium on Parallelism in algorithms and architectures*, pages 180–189, New York, NY, USA, 2006. ACM. 38, 41

[9] Gérard Berry. The constructive semantics of pure Esterel. Draft book, 1999. 2

[10] Gérard Berry. The Esterel v5 language primer. Technical report, Centre de Mathématiques Appliquées Ecole des Mines and INRIA, 1999. 10, 52

[11] Gérard Berry and Laurent Cosserat. The Esterel synchronous programming language and its mathematical semantics. *Lecture Notes in Computer Science*, 197:389–448, 1985. 2

[12] Gérard Berry and Georges Gonthier. The Esterel synchronous programming language: design, semantics, implementation. *Sci. Comput. Program.*, 19(2):87–152, 1992. 2, 4, 6

[13] R. Castelló, R. Mili, and I. G. Tollis. An algorithmic framework for visualizing statecharts. *Lecture Notes in Computer Science*, 1984:43–44, 2001. 3

[14] C.M. Edmund Chow, Joyce S.Y. Tong, M.W. Sajeewa Dayaratne, Partha S Roop, and Zoran Salcic. RePIC a new processor architecture supporting direct Esterel execution. Technical report, Department of Electrical and Computer Engineering, University of Auckland, New Zealand, 2004. 4, 6

[15] Etienne Clossea, Michel Poizea, Jacques Puloua, Patrick Veniera, and Daniel Weil. Saxo-rt: Interpreting Esterel semantic on a sequential execution structure. *Electronic Notes in Theoretical Computer Science*, 65:80–94, 2002. 6

[16] Edsger W. Dijkstra. Letters to the editor: go to statement considered harmful. *Commun. ACM*, 11(3):147–148, 1968. 5

[17] Stephen A. Edwards. Compiling Esterel into sequential code. *Design Automation Conference, 2000. Proceedings 2000. 37th*, pages 322–327, 2000. 4, 6

[18] Stephen A. Edwards. An Esterel compiler for large control-dominated systems. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 21:169–183, 2002. 4, 6

[19] Stephen A. Edwards. Tutorial: Compiling concurrent languages for sequential processors. *ACM Trans. Des. Autom. Electron. Syst.*, 8(2):141–187, 2003. 6

[20] John Ellson, Emden Gansner, Lefteris Koutsofios, North Stephen C., and Gordon Woodhull. Graphviz - open source graph drawing tools. *Lecture Notes in Computer Science*, 2265:594–597, 2002. 3

[21] J. Ferrante and M. Mace. On linearizing parallel code. In *POPL '85: Proceedings of the 12th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, pages 179–190, New York, NY, USA, 1985. ACM. 38

[22] J. Ferrante, M. Mace, and B. Simons. Generating sequential code from parallel code. In *ICS '88: Proceedings of the 2nd international conference on Supercomputing*, pages 582–592, New York, NY, USA, 1988. ACM. 38

[23] David Harel. Statecharts: A visual formalism for complex systems. *Scientific Computer Programming*, 8(3):231–274, 1987. 2

[24] David Harel and Amir Pnueli. On the development of reactive systems. *NATO ASI Series*, 13:477–498, 1985. 2

[25] T. C. Hu. Parallel sequencing and assembly line problems. *Operations Research*, 9(6):841–848, 1961. 38

[26] ILOG. ILOG CPLEX: High-performance software for mathematical programming and optimization (http://www.ilog.com/products/cplex/). 37

[27] N. Karmarkar. A new polynomial-time algorithm for linear programming. *Combinatorica*, 4(4):373–395, 1984. 36

[28] Donald E. Knuth. Structured programming with go to statements. *ACM Comput. Surv.*, 6(4):261–301, 1974. 5

[29] Xin Li, Marian Boldt, and Reinhard von Hanxleden. Mapping Esterel onto a multi-threaded embedded processor. In *ASPLOS-XII: Proceedings of the 12th international conference on Architectural support for programming languages and operating systems*, pages 303–314, New York, NY, USA, 2006. ACM. 4, 7, 15, 29, 42, 51

[30] Xin Li and Reinhard von Hanxleden. A concurrent reactive Esterel processor based on multi-threading. In *SAC '06: Proceedings of the 2006 ACM symposium on Applied computing*, pages 912–917, New York, NY, USA, 2006. ACM. 4, 15, 51

[31] Jan Lukoschus and Reinhard von Hanxleden. Removing cycles in Esterel programs. *EURASIP Journal on Embedded Systems, Special Issue on Synchronous Paradigms in Embedded Systems*, 2007. `http://www.hindawi.com/getarticle.aspx?doi=10.1155/2007/48979`. 29, 72

[32] A. Pnueli and R. Rosner. On the synthesis of a reactive module. In *POPL '89: Proceedings of the 16th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 179–190, New York, NY, USA, 1989. ACM. 2

[33] Steffen Prochnow, Claus Traulsen, and Reinhard von Hanxleden. Synthesizing Safe State Machines from Esterel. In *Proceedings of ACM SIGPLAN/SIGBED Conference on Languages, Compilers, and Tools for Embedded Systems (LCTES'06)*, Ottawa, Canada, 2006. 7, 51

[34] Steffen Prochnow and Reinhard von Hanxleden. Statechart development beyond WYSIWYG. *Lecture Notes in Computer Science*, 4735:635–649, 2007. 3

[35] Apache Velocity Project. The Apache Velocity Project (http://velocity.apache.org/). 18, 47, 49, 53, 79

[36] Partha S. Roop, Zoran Salcic, and M.W. Sajeewa Dayaratne. Towards direct execution of Esterel programs on reactive processors. In *EMSOFT '04: Proceedings of the 4th ACM international conference on Embedded software*, pages 240–248, New York, NY, USA, 2004. ACM. 4

[37] Zoran Salcic, Partha S. Roop, Morteza Biglari-Abhari, and Abbas Bigdeli. REFLIX: a processor core with native support for control-dominated embedded applications. *Microprocessors and Microsystems*, 28(1):13 – 25, 2004. 4

[38] Ingo Schiermeyer. Reverse-fit: A 2-optimal algorithm for packing rectangles. In *ESA '94: Proceedings of the Second Annual European Symposium on Algorithms*, pages 290–299, London, UK, 1994. Springer-Verlag. 38

[39] Arne Schipper. Layout and visual comparison of statecharts. Diploma thesis, Christian-Albrechts-Universität Kiel, 2008. 3

[40] A. Steinberg. A strip-packing algorithm with absolute performance bound 2. *SIAM J. Comput.*, 26(2):401–409, 1997. 38

[41] QNX Software Systems. Qnx realtime operating system (http://www.qnx.com). 7

[42] Olivier Tardieu. Goto and concurrency: Introducing safe jumps in Esterel. *Electronic Notes in Theoretical Computer Science*, 153(4):55 – 70, 2006. Proceedings of the Third International Workshop on Synchronous Languages, Applications, and Programs (SLAP 2004). 8

[43] Olivier Tardieu and Stephen A. Edwards. Instantaneous transitions in Esterel. In *Proceedings of the Workshop on Model-Driven High-Level Programming of Embedded Systems (SLA++P)*, 2007. 8, 71

[44] Esterel Technologies. Esterel studio (http://www.esterel-technologies.com/). 5, 55

[45] Malte Tiedje. Beschreibung des Kiel Esterel Prozessors in Esterel. Diploma thesis, Christian-Albrechts-Universität Kiel, 2008. 4, 8, 51

[46] Stephen A. White. *Introduction to BPMN*. IBM Corporation, 2002. 2

[47] Li Hsien Yoong, Partha S. Roop, Zoran Salcic, and Flavius Gruian. Compiling Esterel for distributed execution. In *SLAP*, 2006. 6

[48] Simon Yuan, Sidharta Andalam, Li Hsien Yoong, Partha S. Roop, and Zoran Salcic. Starpro - a new multithreaded direct execution platform for Esterel. In *SLA++P*, 2008. 4, 6

[49] Jia Zeng, Cristian Soviani, and Stephen A. Edwards. Generating fast code from concurrent program dependence graphs. In *LCTES '04: Proceedings of the 2004 ACM SIGPLAN/SIGBED conference on Languages, compilers, and tools for embedded systems*, pages 175–181, New York, NY, USA, 2004. ACM. 5, 38

Index