INSTITUT FÜR INFORMATIK

Executing Safe State Machines on a Reactive Processor

Falk Starke
Claus Traulsen
Reinhard von Hanxleden

Bericht Nr. 0907
March 2009

CHRISTIAN-ALBRECHTS-UNIVERSITÄT

ZU KIEL

Institut für Informatik der
Christian-Albrechts-Universität zu Kiel
Olshausenstr. 40
D – 24098 Kiel

Executing Safe State Machines on a Reactive Processor

Falk Starke
Claus Traulsen
Reinhard von Hanxleden

Bericht Nr. 0907
March 2009

e-mail:
{fast, ctr, rvh}@informatik.uni-kiel.de

Technical Report

Safe State Machines (SSMs), also known as SyncCharts, are a Statechart dialect with precise synchronous semantics, used to describe the behavior of reactive systems. A natural target for executing SSMs are reactive processors, which have an instruction set architecture (ISA) particularly well-suited for reactive control flow. When synthesizing SSMs into code, this is traditionally done via the synchronous language Esterel. However, this is not always straightforward; transitions in SSMs can jump arbitrarily between states, and there is no Esterel statement that matches this. We here propose to circumvent this by synthesizing SSMs directly onto a reactive ISA that can encode transitions directly as GOTOs. This not only has the potential for smaller and faster code, but preserves the structure of the SSM much better that going via Esterel. Conversely, we note that SSMs appear easier to implement on a reactive processor than Esterel, notably because there is not exception handling required.

Keywords: Safe State Machines, Statecharts, Compilation, Reactive processors

Contents

1 Introduction **1**

2 Related Work **6**

3 Safe State Machines and Esterel **8**
3.1 Safe State Machines 8
3.2 Esterel 9
3.3 Compiling SSMs to Esterel 9

4 The Kiel Esterel Processor **12**
4.1 Architecture 12
4.2 Instruction Set 12

5 Compiling SSM to KEP Assembler **14**

6 Experimental Results **20**

7 Conclusion and Outlook **23**

1 Introduction

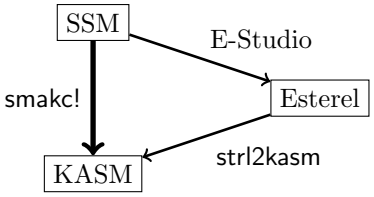
Reactive systems are systems that continuously interact with their environment. The execution of these systems is determined by their internal state and external stimuli. As a reaction, new stimuli and/or a new internal state are generated. To describe the behavior of reactive systems, the family of synchronous languages has been developed, including Esterel [4], Lustre [10] and Signal [9]. These languages offer numerous control flow primitives such as concurrency and preemption that are pertinent to reactive systems, and the *synchrony hypothesis* [4] gives a sound semantical basis to these languages. For a small example that illustrates the uses of concurrency and preemption, consider the Esterel program in Figure 1.2b. The system waits for two inputs A and B. As soon as both inputs have occurred, the output O is emitted. The behavior is reset by the input R. The semantics of the language ensures deterministic behavior, for example there is no race condition between emitting O and reacting to R here, R is guaranteed to get precedence.

In particular for safety-critical systems, it is important that the behavior cannot only be understood by the programmer, but as well by experts in the application area, without further knowledge in computer science. In order to achieve this, graphical notations such as Statecharts were developed. The Statecharts formalism extends the classical formalism of finite-state machines and state transition diagrams by incorporating the notions of hierarchy, orthogonality, compound events, and a broadcast mechanism for communication between concurrent components. Statecharts provide an effective graphical notation, not only for the specification and design of reactive systems, but also for the simulation of the modeled system behavior. Since the original Statecharts proposal [11], numerous dialects of Statecharts have been developed and Statecharts have also been incorporated into the Unified Modeling Language (UML). Today, Statecharts are supported by several commercial tools, *e. g.*, Matlab/Simulink/Stateflow[1], Statemate [11] or Rational Rose[2]. In this paper, we are particularly interested in the Safe State Machines (SSMs) [2] dialect of Statecharts, also known as SyncCharts, which is a graphical variant of Esterel. Figure 1.2a shows an SSM equivalent to the Esterel program in Figure 1.2b.

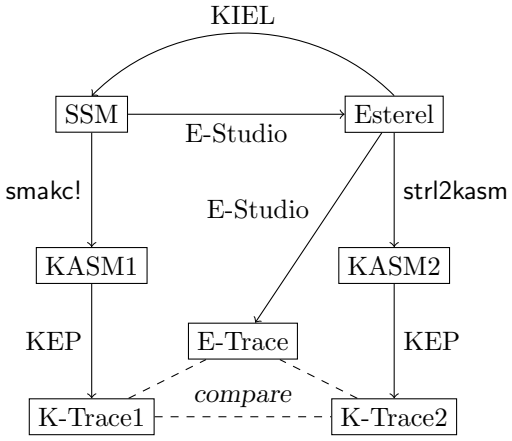
Usually SSMs are transformed to Esterel, which can then be compiled further to either C code, synthesized to hardware or compiled in an additional HW/SW-Codesign approach [15]. Another possibility is to execute Esterel on a *reactive processor*, such as, the Kiel Esterel Processor (KEP) [12], the Emperor [22] or the StarPro [23]. These processors are designed to allow deterministic execution of Esterel programs, by offering special instructions for concurrency and preemption. In particular, sensing for a preemption

[1] http://www.mathworks.com/products/stateflow/

[2] http://www-306.ibm.com/software/awdtools/developer/technical/



(a) Different compilation schemes from SSMs to KEP assembler (KASM). The classical synthesis path involves Esterel as intermediate step, we here propose the direct translation.



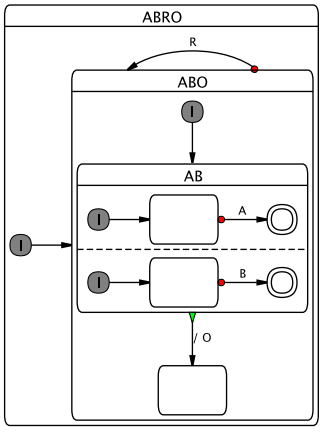
(b) Each model is compiled either directly or via Esterel. To validate the compilation, the executions on the KEP are compared to reference traces generated by E-Studio. The reaction times of both executions are compared to each other.

Figure 1.1: Compilation paths and validation of the compiler

can be performed in parallel to the actual computation, while still preserving determinism. As the KEP assembler code for the ABRO shown in Figure 1.2c illustrates, most Esterel statements can be mapped directly to reactive assembler instructions. Apart from a speedup, this direct correspondence simplifies the timing analysis [5]. The KEP compiler automatically computes the maximum reaction time and initializes the KEP's Tick Manager accordingly, see Figure 1.2c, line 5. In the further examples, we will omit the interface part consisting of I/O signals and the Tick Manager initialization.

This paper investigates how to efficiently execute SSMs on reactive processors. The classical compilation chain starts with an SSM designed with Esterel Studio (E-Studio for short, by Esterel Technologies), from which E-Studio synthesizes Esterel. From here, the strl2kasm [12] compiler generates KEP assembler (KASM). This path is also illustrated in Figure 1.1a. While simple SSMs can be easily transformed into equivalent Esterel programs, this is not true for highly interconnected SSMs. Since Esterel does not have any instructions to jump to a given arbitrary code block, only linear control flow can be easily expressed.

Consider the SSM in Figure 1.3a. It takes two inputs N (nickel) and D (dime). The state encodes how many cents have been entered: 0 for state Zero, 5 for state Five and 10 for state Ten. A GUM is emitted whenever 15 cents have been entered. We assume that the inputs N and D never occur simultaneously. While such a complete graph is an example for highly non-linear control-flow, it illustrates the problems that can occur when such SSMs are transformed into Esterel. Naturally, these difficulties get more pronounced as the number of states and transitions increases. Figure 1.3b shows the



(a) Safe State Machine

```
1  module ABRO:
2  input A, B, R;
3  output O;
4  loop
5    [
6      await A
7    ||
8      await B
9    ];
10   emit O;
11 each R
12 end module
```

(b) Esterel code

```
1  % ——— I/O Signals ————————————
2       INPUT A, B, R
3       OUTPUT O
4  % ——— Initialize Tick Manager ——
5       EMIT _TICKLEN, #11
6
7  A0:  ABORT R, A1        % Begin loop each
8         PAR 1, A2, 1
   % Start concurrent threads
9         PAR 1, A3, 2
10        PARE A4, 1
11 A2:      AWAIT A       % Thread 1
12 A3:      AWAIT B       % Thread 2
13 A4:    JOIN 0          % Join threads
14        EMIT O
15        HALT            % Wait for reset
16 A1:  GOTO A0           % End loop each
```

(c) KEP assembler derived via Esterel (strl2kasm)

Figure 1.2: ABRO—a system that waits for two inputs (A and B), before emitting output O. Shown are equivalent descriptions as SSM and Esterel program, and the KEP assembler generated from Esterel by strl2kasm.

Esterel code that was synthesized by E-Studio[3]. Two auxiliary signals are introduced, to indicate that not the initial state Zero, but either state Five or state Ten should be activated.

The problem illustrated here is that a Statechart transition allows arbitrary control changes, akin to a GOTO, and Esterel only allows structured control. More fundamentally, Statecharts are a means to describe reactive behavior [11], where it may be perfectly natural to transfer from one system state to an arbitrary other system state. The situation is somewhat different in "classical" computer programs, where a structured control flow is desirable [7]. Actually, it is a common misconception among Statechart novices to treat Statecharts as control flow diagrams, where states may merely encode the state of the program counter. This "state" is rather short-lived and relates to the

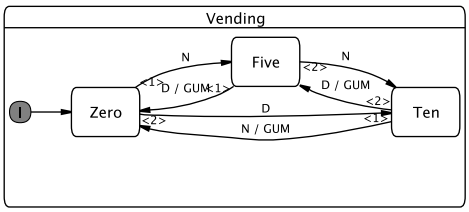
[3]To improve readability of the automatically generated code samples—the Esterel produced by E-Studio as well as the KASM code samples—, we added comments, renamed some labels and slightly polished the formatting.

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

We here propose an alternative path to synthesize code for reactive processors from SSMs, which avoids the detour via Esterel, and performs a direct translation instead. We have investigated this for the KEP execution platform and have developed a state machine to KEP compiler (called smakc!), which produces KEP assembler directly from SSMs—see again Figure 1.1a. For the Vending example, the code produced by smakc! is shown in Figure 1.3c. As can be seen, this code does need any additional signals and directly reflects the original structure. Each state is encoded as a HALT, which is enclosed by (weak or strong) ABORTs that trigger outgoing transitions. For example, state Zero corresponds to the HALT in line 4. The transition from Zero to Five, which depends on signal N, is triggered by the WABORT that is initialized in line 2; if triggered, this transfers control to the label Zero2Five, where a GOTO Five is performed. Note that the nesting order of the ABORTs reflects the transition priorities in the SSM. Hence, the generation of KEP assembler directly from the SSM gives a very simple one-to-one mapping where each assembler instruction can be directly mapped to a state action or a transition. This not only results in compact and efficient code, but also greatly improves code traceability. This supports testing and verification, and in case of safety-critical systems may also aid in certification.

In this paper, we investigate how SSMs can be naturally executed on a reactive processor. On the one hand, this execution is more efficient than the common way to compile an SSM to Esterel and preserves the structure of the SSM in the assembler code. On the other hand, due to the special Instruction Set Architecture (ISA) of reactive processors, the compilation is much simpler than the compilation to a general purpose language like C. Still, the compilation approach presented here could also be beneficial for classical, non-reactive processors, for example if they provide a virtual machine that implements a reactive ISA [14].

In the next section, we will give a short overview about related work. Section 3 introduces SSMs, Esterel, and synchronous languages in more detail. Section 4 gives a short overview about our target platform, a reactive processor and its ISA. Section 5 details our compilation process and Section 6 gives experimental results. We conclude in Section 7.



(a) Safe State Machine

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

(b) Esterel code synthesized from the SSM by E-Studio

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

(c) KEP assembler produced directly from the SSM by smakc!

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

(d) KEP assembler derived via Esterel

Figure 1.3: Vending—an example of tightly interconnected SSM, and various derivatives.

2 Related Work

While Statecharts are an appealing language to describe reactive behaviors, the generation of efficient code is not trivial. Three different methods of compiling Statecharts can be distinguished: compilation into an object oriented language using the state pattern [1], dynamic simulation [21], and flattening into finite state machines. Since flattening can suffer from state explosion, often a combination of flattening and dynamic simulation is used. As Statecharts exist in various different flavors, which code generation scheme is best might also depend on the actual Statechart type and the actual semantics. We focus on SSMs, which have a formal, clean semantics. Since our target architecture directly supports concurrency and hierarchy (by means of preemption) our approach differs from standard Statechart compilation. However, it can be seen as a simulation based approach as used for general purpose processors, were the simulator is implemented in hardware.

A translation from SSMs to Esterel was proposed by André [2] together with the initial definition of SSMs and their semantics. This transformation, with additional unpublished optimizations, is implemented in E-Studio.

Our work is most closely related to the extension of Esterel with GOTO by Tardieu and Edwards [18]. Since they extend the language, they have to consider all possible usages of GOTO, *e. g.*, jumping from one thread into another. Our approach could be directly used to generate efficient extended Esterel (including GOTO) from SSMs, since the structure of the SSM will always generate valid GOTOs. We choose to target KEP assembler, which already has a GOTO statement. One practical reason is that there is no publicly available compiler for extended Esterel; another is that the KEP assembler is close enough to Esterel that this intermediate step would not make much difference in code generation.

The issue of extracting complex signal expressions into simple condition triggers (see Section 5) is related to extracting such expressions into external hardware in hardware/-software co-design [8]. In co-design, the motivation is to accelerate computation, and the challenge is to cleanly extract such expressions into the hw/sw interface. In our setting, the motivation is to provide triggers for the abortion watchers, and the challenge is to ensure that the trigger signals are computed for as long as necessary without blocking progress.

Considering a non-synchronous language like C as alternative synthesis target, the direct generation of C code from SSMs might also be more efficient than the current method to first generate Esterel. But since the compilation of Esterel as well as SSMs is by no means trivial, *e. g.*, because of causality issues, a direct compiler from SSMs to C would be very complex.

Reactive processors are especially designed for reactive systems. In particular, they support preemption and concurrency, as well as timing predictability. All existing reactive processors have been built with the idea of executing Esterel programs in mind [20]. Thus their ISA is very similar to Esterel statements. Besides reactive processors that are based on Esterel, there are also approaches for efficient processors with deterministic timing that use C as an input language like the PReT [13] or the PREDATOR[1] project.

Esterel programs can be compiled to the KEP by using strl2kasm. It first translates Esterel into an intermediate graph structure, called concurrent KEP assembler graph (CKAG), on which the scheduling is performed. The compiler also incorporates a Worst Case Reaction Time (WCRT) analysis [5]. The compiler for the STARPro [23] is very similar to the strl2kasm, but has to insert explicit instructions to check for preemption. The compilation for the EMPEROR [6] differs from both other approaches, since the target is not multi-threaded, but consists of multiple cores that execute in truly parallel fashion.

[1] http://predator-project.eu/

3 Safe State Machines and Esterel

SSMs as well as Esterel are synchronous languages, specially designed to describe reactive systems. Both emphasize on preemption and concurrency, key issues for such systems. Concurrent threads run completely synchronously and can communicate instantaneously back and forth. The execution is divided into logical *instants*, or *ticks*, and communication within or across threads occurs via *signals*. At each tick, a signal is either *present* (emitted) or *absent* (not emitted). While *pure signals* only have a status, *valued signals* carry an additional value. This value is preserved over instants, and it can only change if the signal is present. Besides usual termination, all program parts can be suspended or aborted. Both languages distinguish between *weak abortion*, where the aborted code is executed one last tick, and *strong abortion*, where the code is aborted immediately.

While the semantics of SSMs and Esterel are quite similar, their appeal is different. SSMs, as a graphical language, are very good to give an intuitive understanding of the system, without prior knowledge of the semantical details, while Esterel code can be more compact. However there is also some behavior which can be described more compactly in SSMs. In particular, systems with different modes, where the mode can change from any state to another, are easier to express in SSMs than in Esterel, like the example in Figure 1.3a.

3.1 Safe State Machines

SSMs are a Statechart dialect with a synchronous semantics that strictly conforms to the Esterel semantics. A procedural definition of SSMs is given by André [2]. The basic object in SSMs is a *reactive cell*, which is a state with its outgoing transitions. Reactive cells are combined to *state-transition graphs*, which we will also refer to as *state regions*. A *macro-state* consists of one or more state-transition graphs. Additionally, SSMs can contain *textual macrostates*, which consist of plain Esterel code. States can also have *internal actions*: on entry, on exit and during. SSMs inherit the concept of signals and valued signals from Esterel. Hence a transition trigger can consist of an event, which tests for presence and absence of values, and a conditional, which may compare numerical values. Characteristic for SSMs are the different forms of preemption, expressed by different state transition types. Weak and strong abortion transitions as well as suspension can be applied to macrostates. A macrostate can either be left by an abortion, which has an explicit trigger, or by a *normal termination*, which is taken if the macrostate enters a terminal state. Analogously to Esterel, all transitions can either be immediate or delayed, where a delayed transitions is only taken if the source state

was already active at the start of an instant. In contrast, immediate transitions may be taken as soon as the state becomes active; this enables the activation and deactivation of a state multiple times within one instant. Delayed transitions can also be *count delayed*, *i. e.*, the trigger must have been evaluated to true for a specific number of times, before the transition is enabled. When a state has more than one outgoing transition, a unique *priority* is assigned to each of them, where lower numbers have higher priority. Weak abortions must have lower priority than strong abortions, and if a normal termination exists, it always has the lowest priority.

3.2 Esterel

Esterel is a textual language, with a strict synchronous semantics. For an overview of representative Esterel statements, see Table 3.1. Esterel statements are either *transient*, in which case they do not consume logical time, or *delayed*, in which case execution is finished for the current tick. Per default statements are transient; delayed statements include pause and (non-immediate) await. Esterel's parallel operator || groups statements in concurrently executed threads. The Esterel language distinguishes *kernel statements* (*e. g.*, emit, pause) and *derived statements* (*e. g.*, await, every). Derived statements are in general just syntactic sugar and can be reduced to kernel statements.

Esterel *traps* are used to express exceptions. When an exit is executed, control immediately shifts to the corresponding handler. When multiple parallel traps are raised, the trap with the outermost handler will be executed.

3.3 Compiling SSMs to Esterel

Since SSMs were developed as a graphical version of Esterel, programs in one language can in general be fairly directly translated into the other. The transformation from SSMs to Esterel is implemented in E-Studio, while the opposite transformation is implemented in the KIEL tool [16]. The main difficulty in going from Esterel to SSMs is that SSMs do not directly provide traps (exceptions). Weak abortion type transitions in SSMs are related to traps, but do not provide the immediate control change offered by exceptions. Transitions in SSMs are more restrictive in that abortions only happen at tick boundaries. However, as mentioned before, transitions in SSMs are less restrictive than control flow in Esterel in that transitions can transfer control to arbitrary other states (within the same hierarchy). Put another way, control flow *timing* is more restrictive in SSMs, but control flow *structure* is more restrictive in Esterel. The GOTO offered by reactive processors is not restrictive in either dimension.

The translation of states is trivial, they can be directly expressed by await or abort in Esterel. The translation of transitions is more complex: Esterel provides conditionals, loops and trap exits, but does not allow to jump directly to another block of code. Hence, when we want a transition from state A to state B, A must be in the scope of B. But if we have the reverse transition as well, A and B must be in each others scope, which

is not possible in Esterel's syntax. Instead, the whole region needs to be embedded into a loop, as illustrated in the Vending example (Figure 1.3b). Each state is encoded as an await case statement, with one case for each transition trigger. Signals encode which state is to be taken next; the presence of go2Five or go2Ten indicate a transition to states Five or Ten, respectively; if neither is present, this indicates a transition to state Zero. When a transition is taken, control will jump to the end of the loop, restart the region and determine which state shall be active. For the Vending example, the Esterel code behaves as follows. Initially, go2Five is absent, so control will flow to the second await statement, which corresponds to state Zero. The await is non-immediate, hence the current tick terminates. Assume that signal N is set in the next tick. Then emit go2Five (line 16) is executed. Hence the control will skip the next present (line 21), the loop body terminates, the loop is restarted and control flows to the first await (line 6, state Five).

Now consider the KEP assembler in Figure 1.3d. The structure is quite similar to the Esterel code, and also uses signals go2Five and go2Ten to trigger transitions. As go2Ten is declared for the whole program, the compiler only assigns a register to it, but does not introduce a SIGNAL instruction that resets a signal at the entry of its scope. In contrast, the go2Ten signal is declared inside the loop and must be reset in each iteration (line 1).

As the KEP ISA does not directly support await case, the strl2kasm compiler converts this into a sequence of trap exits embedded in a loop. The transition from Zero to Five is implemented by 10 instructions: L16–L18, L24–26, L39, L1–L4. For comparison, in the KEP assembler in Figure 1.3c, this requires only 5 instructions: L4, L6, L8–10 are executed.

The Esterel program, and the KEP assembler code derived from it, correctly present the semantics of the original SSM, but do not really preserve its structure. This example with only three states is still fairly simple. SSMs with more states and more elaborate transition patterns can result in arbitrarily complicated control flow in the generated Esterel program.

Mnemonic, Operands	**Esterel Syntax**	**Cycles**	**Notes**
PAR *prio*$_1$, *startAddr*$_1$, *id*$_1$ ... PAR *prio*$_n$, startAddr$_n$, *id*$_n$ PARE *endAddr* *startAddr*$_1$: ... *startAddr*$_2$: ... *startAddr*$_n$: ... *endAddr*: JOIN *prio*	[p$_1$ || ⋮ || p$_n$]	$n+1$ 1	For each thread, one PAR is needed to define the start address, thread id and initial priority. The end of a thread is defined by the start address of the next thread, except for the last thread, whose end is defined via PARE. The *prio* argument of JOIN instruction sets the priority of the parent thread. This allows to lower the priority of a parent thread after spawning of child threads.
PRIO *prio*		1	Set current thread priority to *prio*.
[W]ABORT[I, *n*] *S*, *endAddr* ... *endAddr*:	[**weak**] **abort** ... **when** [**immediate**, *n*] *S*	2	
SUSPEND[I,*n*] *S*, *endAddr* ... *endAddr*:	**suspend** ... **when** [**immediate**, *n*] *S*	2	
startAddr: ... EXIT *exitAddr startAddr* ... *exitAddr:*	**trap** *T* **in** ... **exit** *T* ... **end trap**	1	Exit from a trap, *startAddr*/*exitAddr* specifies the trap scope. Unlike the GOTO, the EXIT checks for concurrent EXITs and terminates enclosing ||.
PAUSE	**pause**	1	Wait for a signal. AWAIT TICK is equivalent to PAUSE.
AWAIT [I, *n*] *S*	**await** [**immediate**, *n*] *S*	1	
SIGNAL *S*	**signal** *S* **in** ... **end**	1	Initialize a local signal *S*.
EMIT *S* [, {#*data*|*reg*}]	**emit** *S* [(*val*)]	1	Emit (valued) signal *S*.
SUSTAIN *S* [, {#*data*|*reg*}]	**sustain** *S* [(*val*)]	1	Sustain (valued) signal *S*.
PRESENT *S*, *elseAddr*	**present** *S* **then** ... **end**	1	Jump to *elseAddr* if *S* is absent.
HALT	**halt**	1	Halt the program.
addr: ... GOTO *addr*	**loop** ... **end loop**	1	Jump to *addr*.

Table 3.1: Overview of the KEP instruction set architecture, and their relation to Esterel and the number of processor cycles for the execution of each instruction.

4 The Kiel Esterel Processor

The main idea of the KEP is the direct support for preemption via watchers and parallelism by supporting hardware threads with a priority based scheduling. Originally the KEP was implemented by Xin Li [12] using VHDL. It was reimplemented in Esterel [19], then named KEPe, for better maintainability and to test the practicability of Esterel for hardware design. However, the KEPe does not fully implement all features, for example valued signals are missing. Exceptions (trap/exit) are also not supported by the KEPe. This is a limitation for running Esterel programs, but not for executing SSMs.

4.1 Architecture

The significant parts of the architecture of the KEP are the thread block and the reactive core [12]. The thread block manages the different threads and schedules them according to their priority. It also has to handle abortion of threads by their parent threads. Each thread has an independent program counter (PC) and threads are scheduled according to their respective status and dynamically changing priorities. At the beginning of each instruction cycle the enabled thread with the highest priority is selected and executed. The scheduler is very light-weight. In the KEP, scheduling and context switching do not cost extra instruction cycles, only changing the priority of a thread costs an instruction. The priority-based execution scheme allows on the one hand to enforce an ordering among threads that obeys the constraints given by Esterel's semantics, but on the other hand avoids unnecessary context switches. If a thread lowers its priority during execution but still has the highest priority, it simply keeps executing.

The reactive block provides a configurable number of Watcher units, which detect whether a signal triggering a preemption is present and whether the program counter (PC) is in the corresponding preemption body. Therefore, no additional instruction cycles are needed to test for preemption. The reactive block also provides a signal test and an await cell to handle delays.

4.2 Instruction Set

The instruction set of the KEP is very similar to the Esterel language. The KEP ISA includes all kernel statements, and in addition some frequently used derived statements. The KEP ISA also includes valued signals, which cannot be reduced to kernel statements. The only parts of Esterel v5 that are not part of the KEP ISA are combined signal handling and external task handling.

Due to this direct mapping from Esterel to the KEP ISA, most Esterel statements can be executed in just one instruction cycle. For more complicated statements, well-known translations into kernel statements exist, allowing the KEP to execute arbitrary Esterel programs. Part of the KEP instruction set is shown in Table 3.1.

5 Compiling SSM to KEP Assembler

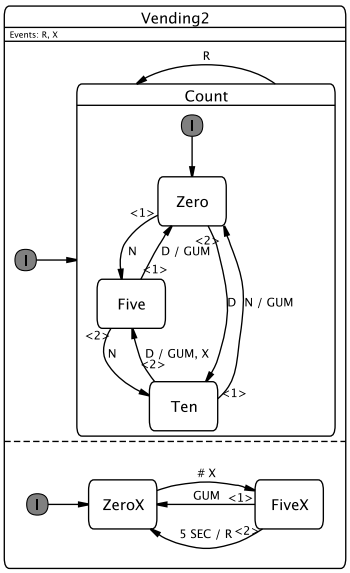
In order to compile an SSM into KEP assembler, several transformations are applied to the SSM, which might alter its structure, or enrich it with additional information. In a last step, the generated SSM can be directly written into KEP assembler. The compilation is carried out by the following steps:

1. *Complex conditionals extraction:* this transforms complex signal expressions to simple signal tests. This transformation results in an SSM with the same behavior, which does not contain conditional expressions.
2. *Dependency analysis:* this detects data and control flow dependencies and adds them to the SSM as special transition type.
3. *Cycle detection:* to ensure schedulability, the dependency graph is inspected for cycles.
4. *Scheduling:* the states are scheduled according to the encountered dependencies, the schedule is implemented by assigning appropriate thread priorities.
5. *Code generation:* this generates the target platform's code.

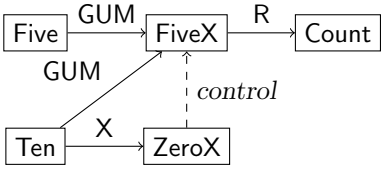
In the following we will illustrate the compilation process for the Vending2 example in Figure 5.1. The left part of the SSM is identical to the SSM in Figure 1.3a. It takes as input the information whether a dime (D) or a nickel (N) was entered. It emits the signal GUM whenever 15 cents where entered. When the user has entered 20 cents, the user is credited 5 cents (indicated by signal X), and has 5 seconds time to enter another 10 cents; otherwise, the right side of the SSM will trigger a reset (signal R), and the credit is lost.

Step 1: Complex conditionals extraction

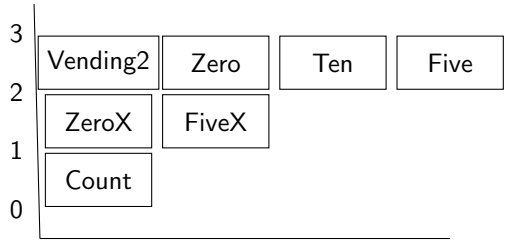
Computing complex conditionals is a common task for compilation. In SSMs, transitions may be triggered by complex signal expressions. A complication when compiling to a reactive processor is that aborts are triggered by simple signals, thus necessitating the creation of a new auxiliary signal that is present whenever the original signal expression evaluates to true. This auxiliary signal is computed in parallel to the state region where the complex expression occurs. In the original state region, the complex expression is then replaced by a test for this one signal. This also has the benefit that the same signal can be reused in case one expression is used multiple times in the same state region.



(a) Safe State Machine



(b) Signal and control dependencies



(c) Packing from which the schedule will be derived

```
1  % ———— state Vending2 (part 1/2) ——
2                  SIGNAL R
3                  SIGNAL X
4                  PAR 3, Count, 1
5                  PAR 2, ZeroX, 2
6                  PARE EndVending2, 0
7  % ———— state Count (part 1/2) ————
8  Count:          WABORT R, Count2Count
9                  PAR 3, Zero, 3
10                 PARE EndCount, 0
11 % ———— state Zero ————————————————
12 Zero:           WABORT N, Zero2Five
13                 WABORT D, Zero2Ten
14                 HALT
15 Zero2Ten:       GOTO Ten
16 Zero2Five:      GOTO Five
17 % ———— state Five ———————————————
18 Five:           WABORT D, Five2Zero
19                 WABORT N, Five2Ten
20                 HALT
21 Five2Ten:       GOTO Ten
22 Five2Zero:      EMIT GUM
23                 GOTO Zero
24 % ———— state Ten ————————————————
25 Ten:            WABORT N, Ten2Zero
26                 WABORT D, Ten2Five
27                 HALT
28 Ten2Five:       EMIT GUM
29                 EMIT X
30                 GOTO Five
31 Ten2Zero:       EMIT GUM
32                 GOTO Zero
33 % ———— state Count (part 2/2) ————
34 EndCount:       JOIN 1
35                 HALT
36 Count2Count:    GOTO Count
37 % ———— state ZeroX ———————————————
38 ZeroX:          AWAITI X
39                 GOTO FiveX
40 % ———— state FiveX ———————————————
41 FiveX:
   WABORT GUM, FiveX2ZeroX_1
42                 LOAD _Count,#5
43                 WABORT SEC, FiveX2ZeroX_2
44                 HALT
45 FiveX2ZeroX_2:  EMIT R
46                 GOTO ZeroX
47 FiveX2ZeroX_1:  GOTO ZeroX
48 % ———— state Vending2 (part 2/2) ——
49 EndVending2:    JOIN 3
50                 HALT
```

(d) KEP Assembler produced by smakc!

Figure 5.1: Vending2—a vending machine with credit.

A remaining issue is for how long this auxiliary signal needs to be computed. If we would simply compute this indefinitely, the whole parallel region would not properly terminate, as the semantics of concurrency here is that a parallel region only terminates if all concurrent regions have terminated (or an external preemption takes place).

As illustrated in Figure 5.2, we handle this issue by introducing further auxiliary signals, by which on the one hand an original signal region communicates to the auxiliary signal computation region that an original region has finished (signals _FINISH_A and _FINISH_B), and on the other hand the auxiliary signal computation region communicates to the original regions that the whole parallel region is ready to terminate (signal _FINISH_ALL).

Step 2: Dependency analysis

The signal dependency detection for SSMs is fairly straightforward. For each signal, we first compute all possible sources and sinks. A state A is a source of a signal S, if it is either emitted in a state action, or if it is emitted on any outgoing transition. A state B is a potential sink of a signal S, if S is tested in any action, on an outgoing transition, or in the suspend trigger of B. A *signal dependency* is a pair of states (A, B) which are concurrent, and for which a signal S exists, such that A is a source of S and B is a sink of S. We do not have to consider dependencies between non-concurrent states, as those states are already in a fixed ordering which cannot be changed through scheduling of threads. However, we must consider states that can reach each other via immediate transitions. If this is the case, the priority of the source state in general has to be at least as high as the priority of the sink state, which corresponds to a *control dependency*. In the Vending2 example, state ZeroX transfers control immediately to FiveX if X is present (the immediate nature of the transition is indicated by the hash mark). Hence ZeroX "inherits" the priority from FiveX. In this example, this turns out to be unnecessarily conservative; however, it would be necessary if FiveX could immediately emit a signal, *e. g.* via an on-entry action or an immediate outgoing transition.

To determine whether two states are concurrent to each other, it suffices to compute the lowest state that contains both states and then to check whether both states occur in the same region of this state.

The dependencies returned by this algorithm are conservative over-approximations, since some structurally possible constellations of simultaneously active states might not be possible in a state machine. But precise checking whether two states can be executed in parallel during run-time is not decidable for SSMs with data[1].

In our vending machine example, we get the dependencies illustrated in Figure 5.1b. Sources for signal GUM are state Five and state Ten, and reader is FiveX. The only source

[1]It is well known that checking whether two-clocks of a Lustre program are semantically equivalent is undecidable [10]. The translation of the Lustre program into an automaton yields a valid SSM. By putting two simple automata in parallel, where the value of the clocks is encoded by two states, we can reduce the problem to check for clock equivalence to the problem to check whether states can be executed in parallel.

for signal R is FiveX, and the only reader is Count. Source for signal X is state Ten, and the only reader s state ZeroX. As explained before, there is also a control dependency from ZeroX to FiveX.

Step 3: Cycle detection

On the dependencies, a simple cycle detection is performed. This analysis could be omitted, since the subsequent scheduling algorithm also 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.

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 a compiler flag.

The Vending2 example does not contain any cycles.

Step 4: Scheduling

The signal dependencies can now be used to schedule the SSM. Each source state must have a higher priority than the corresponding sink state. Since a thread on the KEP can only decrease its priority during one tick, each state must have a priority that is greater or equal to all its successor states. Similar, the priority of a complex state must be greater or equal than all of its child states. With this information, we could use constraint solving or the simplex algorithm to compute a schedule. However, we choose to use ideas from strip packing [3] instead. This allows to easily incorporate pre-scheduled states (*e. g.*, for modularization): we assign a height of 1 to each state, whereas the scheduling of SSMs that have already been scheduled can be reused by assigning the total height of their packing as item height.

Placing the source of a dependency above the sink for each dependency yields a valid sequence of execution for the states that fulfills the dependencies. The packing in Figure 5.1c is such a placement (compare with Figure 5.1b). Note that the states that do not have any dependencies attached to them (in the example Vending2 and Zero) get arbitrarily assigned the top placement. So, in the example, the states Vending2, Zero, Five, and Ten all have the highest priority (3), followed by ZeroX and FiveX (priority 2) and Count (priority 1).

As an optimization, states that are neither source nor sink of a dependency can be omitted from the scheduling entirely, and later during code generation simply get the current thread priority assigned. This saves some more priority switch instructions.

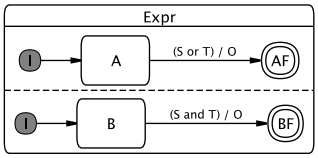
Step 5: Code generation

Once we have scheduled the SSM, generating code can be done by a simple templating mechanism. Each state first initializes one abort for each outgoing transition and possibly a suspend, when a suspend trigger for this state exists. The body of a simple state contains all on entry actions, followed by a loop that contains all on inside action or a HALT if no on inside actions exists. For a complex state, the body spawns a parallel thread for each region, before recursively inserting code for the sub states.

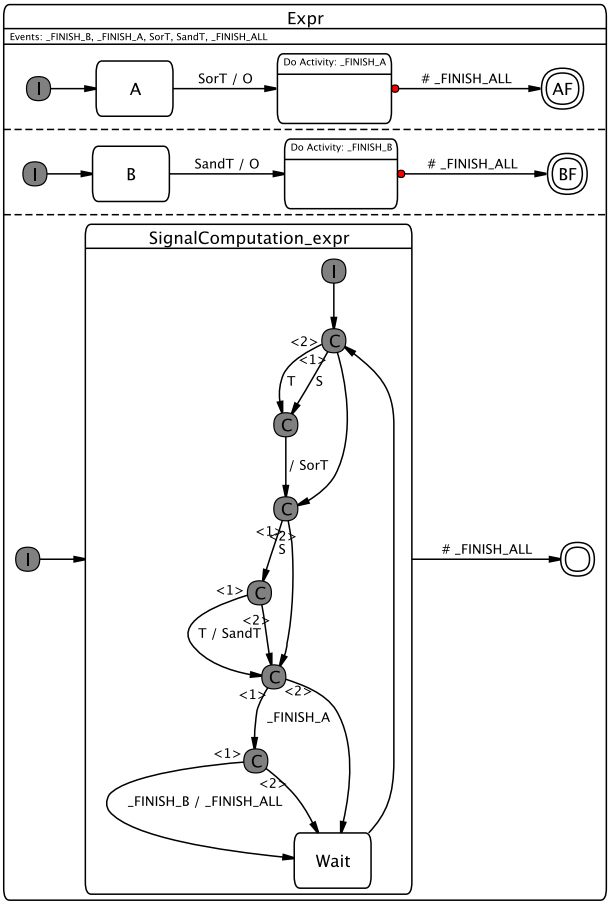
At the end of the abort scope for each transition, first code that emits the effect of the transition is generated. Thereafter, the correct priority for the target state is set, before a GOTO to the target state is executed. For optimization, an empty state that only has one outgoing transition is translated to an AWAIT instruction instead of an abort. Conditional pseudo-states are implemented by present tests. It might be the case that the outgoing transitions of a macrostate must be executed with a lower priority than its inner states. Therefore, we might first execute the inner state in a separate, embedded thread running at high priority. Then, the macro-state executes the JOIN statement in each tick at a lower priority, which can trigger a preemption. Note that the watchers are also triggered by executing the inner states.

This thread embedding illustrated in the code in Figure 5.1d, where Count, running at priority 3 (specified as first argument to PAR instruction in line 4), spawns (lines 9/10) an internal thread running also at priority 3. This internal thread executes the sub-states of Count, which, via signal GUM and state FiveX (running subsequently at priority 2) may indirectly trigger the preemption of Count. Once the sub-states of Count and the states concurrent to Count have executed, Count regains control with the JOIN statement in line 34. That JOIN is first executed with priority 3, but also sets the priority of the Count thread to 1. (This implies that the JOIN must be executed twice in the initial tick, once before the child threads, once after the child threads.) The Count thread then remains at priority 1 until it is terminated. The JOIN also tests the preemption signal R. If R is present, the corresponding watcher (line 8) triggers a reset via the GOTO Count (line 36).

Note that the strl2kasm compiler has to solve the same problem, and does so slightly differently. It does not use thread embedding, but instead would convert each HALT instruction to a PAUSE embedded in a loop that lowers and raises priorities accordingly. In Figure 5.1d, the HALT instructions in the thread embedded in Count would become something like *Label*: PRIO 1; PRIO 3; PAUSE; GOTO *Label*. This saves the extra thread, but typically results in larger, slower code. For comparison, the thread embedding approach does not manipulate priorities in this explicit fashion at each tick, but instead uses the capabilities provided in hardware by the JOIN instruction.



(a) Original SSM



(b) Expanded expressions

Figure 5.2: Example for expanding signal expressions.

6 Experimental Results

We used the K(r)epevalbench[1] to verify the KEP assembler produced by smakc! and to measure its reaction times. Experiments were carried out on several SSM examples. The SSMs were created in E-Studio, which was also used to compile them into Esterel code and to generate traces (E-traces) that covered all SSM transitions. We compiled the SSM and the generated Esterel program into KEP assembler. The two KEP programs could then be directly compared to each other in terms of size and reaction times, see Figure 1.1b.

We did so by executing the programs on the KEP with the E-traces as inputs, generating output traces (K-Traces). The K(r)epevalbench compared the generated outputs to the outputs in the E-traces, thereby confirming the correctness of the compilation. It also recorded the minimum, maximum and average reaction times of the generated code, which we compared to the reaction times for the compilation via Esterel. Hence we used E-Studio as a reference point both for validation (correctness of smakc!) and for benchmarking (competitiveness of smakc! with code synthesized via E-Studio).

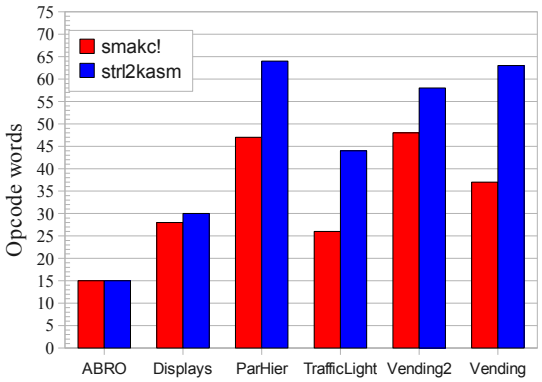
The object code size, measured in 40-bit instruction words, for compilation by the two compilers is listed in Figure 6.1a. Our tests showed that the direct code generation in general produced comparable code sizes. The smakc! compiler generates smaller programs than strl2kasm. Still the code could be further optimized by removing superfluous thread embeddings and GOTOs, this requires a slightly more sophisticated dependency and control flow analysis. The smakc! compiler is particular at an advantage when the levels of nested hierarchy and transition count per state grow. A possible explanation is that the transformation to Esterel described by André and implemented in E-Studio has difficulties to efficiently encode exactly these two cases.

Average and maximum reaction times of the code are shown in Figure 6.1b. Again, the results are comparable, but usually slightly better for the direct compilation, in particular in the automata-like examples. A difficult encoding using a lot of Esterel instructions also accounts for more KEP assembler instructions, increasing the reaction times of the resulting code. The reactions of the code generated by smakc! can sometimes be slower, in particular due to the thread embedding, where additional JOIN statements are executed in each tick.

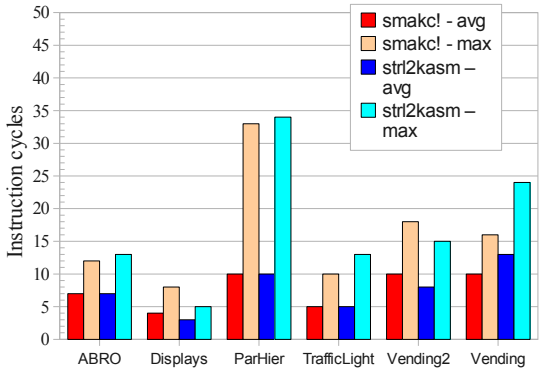
While the direct compilation is only better in some cases, the figures are in general comparable. The strl2kasm compiler makes a more involved dependency analysis. Incorporating this into the smakc should help to avoid many unnecessary embeddings of threads in macrostates, and hence improve both code size and reaction times.

The clean structure of the assembler code generated from the templating process

[1] http://www.informatik.uni-kiel.de/rtsys/kep/



(a) Comparison of code size.



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

Figure 6.1: Comparison between smakc and strl2kasm.

applied to the SSM permits a very simple implementation of a tracing tool. With approximately only three hours of coding work, we wrote a small program that reads an assembler file generated with our compiler and an SSM source code file, both in a single pass. The SSM source code uses the textual, human-readable KIT format [17]. The program generates a map that maps an assembler code line to the corresponding line in the SSM source. This map can be used by the K(r)epevalbench when tracing a the run of a KEP program. As illustrated in Figure 6.2, the current state of a system can be highlighted in the textual description of the SSM. An alternative, which is not implemented yet but should be fairly straightforward, would be to highlight the current state not in the textual format, but directly in the graphical SSM.

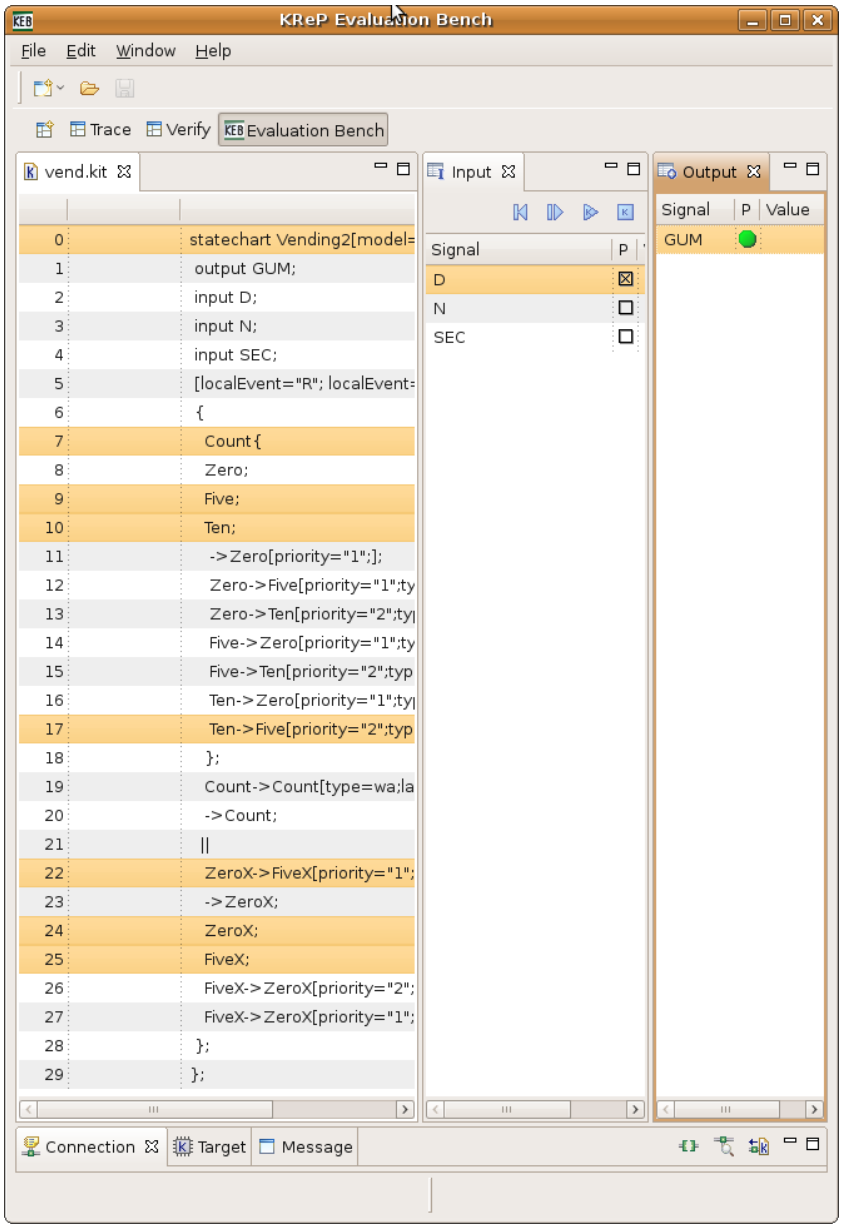


Figure 6.2: Executing the Vending2 SSM (Figure 5.1a) using the evaluation bench. The left panel shows the KIT-source for the SSM, the center and right panels show input and output signals, respectively. The system is currently in states Ten and ZeroX, and entering a dime (signal D) triggers transitions to Five and FiveX, respectively.

7 Conclusion and Outlook

We have shown that SSMs can be translated to a reactive processor in a fairly straightforward, efficient manner. While reactive processors are (so far) designed for Esterel, their special hardware instructions match the needs for SSM execution. This makes the compilation much easier and more efficient than the compilation to a general purpose architecture via a language like C.

Our experiments showed that direct compilation has significant performance advantages only in some cases. In regular applications, there is usually a linear, or circular main control flow with some branches. This type of "classic" program can be efficiently compiled to the KEP using the compilation via a high level language, in this case Esterel. Applications closer to hardware behavior description with a high level of interconnection however do perform better when directly compiled to a processor's assembler, making use of a GOTO for encoding transitions.

Another benefit is that the structure of the SSM is preserved on the assembler level. This greatly simplifies the traceability from the original state-machine to the KEP assembler, by giving a one to one correspondence between assembler instructions and state or transitions, respectively. We currently take advantage of this by tracing the execution of reactive programs in textual SSM descriptions; as mentioned above, it should be straightforward to map these traces directly to graphical SSMs as well.

Overall, we conclude that SSMs are a very natural input language for reactive processors. Even though at this point we did not reach substantial performance increases compared to the route via Esterel, we noticed that several issues that can be rather tedious when going via Esterel, like causality analysis and scheduling, become relatively straightforward when coming from SSMs. This not only helps the aforementioned traceability, but also simplifies the job of the compiler writer. On the one hand, reactive ISAs directly provide the flexible control flow expressed in SSMs; on the other hand, SSMs do not entail certain features of Esterel (notably traps) that are complicated to support in a reactive ISA.

There are several opportunities for optimizations, as already mentioned in Section 6. Consider also again the code in Figure 5.1d. The GOTO Five in line 16 could be safely eliminated, as it transfers control to a state that follows immediately. One could also order states in a way that further exposes such optimization possibilities. The GOTO ZeroX in line 46 could also be safely eliminated, for a slightly different reason: here we have two transitions to the same target, and the lower-priority transition does not have any action associated with it. One might also try to optimize the priority assignment further, *e. g.* by exploiting thread-id precedences to enforce dependencies.

We would like to extend the input to SSM with reference and textual macro states, which would also expand the range of available benchmarks. Reference states could

allow the efficient reuse of already compiled state-machines. However, note that modular compilation for synchronous programs is not possible in general, due to the instantaneous communication. So, a conservative approximation of possible signal dependencies must be taken. Textual macrostates would require to use the existing compiler from Esterel to the KEP inside our compiler.

Other, but similar, target platforms could be addressed as well, like the STARPro or Esterel with GOTO. One might also explore the design of reactive processors with an ISA that does not try to support full Esterel, but only the subset needed for SSMs. This essentially would remove traps, which would simplify the logic required in the reactive core. On the same token, one could develop a virtual machine running on classical general purpose processor that provides such a reduced ISA. This would allow to take advantage of the smakc! compilation approach on a much broader range of platforms than just reactive processors.

Bibliography

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

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

[3] John Augustine, Sudarshan Banerjee, and Sandy Irani. Strip packing with precedence constraints and strip packing with release times. In *Proceedings of the Eighteenth Annual Acm Symposium on Parallelism in Algorithms and Architectures (SPAA'06)*, pages 180–189, New York, NY, USA, 2006. ACM.

[4] Gérard Berry and Georges Gonthier. The Esterel synchronous programming language: Design, semantics, implementation. *Science of Computer Programming*, 19(2):87–152, 1992.

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

[6] M. W. Sajeewa Dayaratne, Partha S. Roop, and Zoran Salcic. Direct Execution of Esterel Using Reactive Microprocessors. In *Proceedings of Synchronous Languages, Applications, and Programming (SLAP'05)*, April 2005.

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

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

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

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

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

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

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

[14] Becky Plummer, Mukul Khajanchi, and Stephen A. Edwards. An Esterel virtual machine for embedded systems. In *International Workshop on Synchronous Languages, Applications, and Programming (SLAP'06)*, Vienna, Austria, March 2006.

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

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

[17] Steffen Prochnow and Reinhard von Hanxleden. Statechart development beyond WYSIWYG. In *Proceedings of the ACM/IEEE 10th International Conference on Model Driven Engineering Languages and Systems (MoDELS'07)*, Nashville, TN, USA, October 2007.

[18] Olivier Tardieu and Stephen A. Edwards. Instanteneous transitions in esterel. In *Proceedings of Model Driven High-Level Programming of Embedded Systems (SLA++P'07)*, Braga, Portugal, March 2007.

[19] Malte Tiedje and Claus Traulsen. Designing a reactive processor with Esterel v7. In *Proceedings of the Workshop on Model-Driven High-Level Programming of Embedded Systems (SLA++P'08)*, Budapest, Hungary, April 2008.

[20] Reinhard von Hanxleden, Xin Li, Partha Roop, Zoran Salcic, and Li Hsien Yoong. Reactive processing for reactive systems. *ERCIM News*, 66:28–29, October 2006.

[21] Andrzej Wasowski. On efficient program synthesis from Statecharts. In *Proceedings of the 2003 ACM SIGPLAN Conference on Language, Compilers, and Tools for Embedded Systems (LCTES'03)*, volume 38, issue 7, June 2003. ACM SIGPLAN Notices.

[22] Li Hsien Yoong, Partha Roop, Zoran Salcic, and Flavius Gruian. Compiling Esterel for distributed execution. In *International Workshop on Synchronous Languages, Applications, and Programming (SLAP'06)*, Vienna, Austria, March 2006.

[23] Simon Yuan, Sidharta Andalam, Li Hsien Yoong, Partha S. Roop, and Zoran Salcic. STARPro—a new multithreaded direct execution platform for Esterel. In *Proceedings of Model Driven High-Level Programming of Embedded Systems (SLA++P'08)*, Budapest, Hungary, April 2008.