04491 – The Kiel Esterel Processor
The Kiel Esterel Processor – A Semi-Custom, Configurable Reactive Processor
— Dagstuhl Seminar —

Xin Li[1], Reinhard von Hanxleden[2]

Real-Time and Embedded Systems Group
Institute of Computer Science and Applied Mathematics
Faculty of Engineering, Christian-Albrechts-Universität zu Kiel, Germany

[1] xli@informatik.uni-kiel.de
[2] rvh@informatik.uni-kiel.de

Abstract. The synchronous language Esterel is an established language for developing reactive systems. It gives an abstract, well-defined and executable description of the application, and can be synthesized into hardware and software. Typically, an Esterel program is first translated into other, lower-level languages (such as VHDL or C), and then compiled further. However, there is also the alternative of executing Esterel-like instructions directly. For example, in the REFLIX and RePIC projects, Roop et al. have augmented traditional processors with custom hardware to execute Esterel instructions. This patch strategy is a convenient approach, but has some shortages.
We present the Kiel Esterel Processor (KEP), a semi-custom, configurable reactive processor for the direct execution of Esterel programs. It consists of a reactive core and scalable peripheral elements. KEP supports standard Esterel statements directly, except (so far) for the concurrency operator. Valued signals and counter functions in Esterel statements are supported by KEP. Due to its control path and its cooperation with elements, KEP obeys exact Esterel (preemption and priority) rules, including for example abort/weak abort (nests).

Keywords. Esterel, synchronous languages, reactive programming, ASIPs

1 Introduction

1.1 Synthesis Options

The synchronous language Esterel has been developed for programming reactive systems [4]. Since its inception in the beginning of the 1980's, the Esterel language has been used for designing reactive embedded systems, including aircraft controllers, automobile dashboards, and RAM controllers. As a system-level language, it gives an abstract, well-defined and executable description of the application, and can be synthesized into low-level languages for further compiling and implementing.

There are several methods to implement an Esterel model [2,4,3,5].

- **Hardware Synthesis**
  Hardware implementations (Figure 1(a)), where an Esterel program is synthesized into a hardware circuit presentation (e. g. VHDL or Verilog HDL), lead to small footprints (low memory requirements) and cheap implementations. However, hardware implementations are not flexible, meaning that even a tiny modification of the program will require a re-synthesis. Furthermore, its resources usage increases rapidly when data path handling is needed.
- **Software Synthesis**
  In a software implementation (Figure 1(b)), an Esterel program is first synthesized into sequential, lower level language code (e. g. C or JAVA), and then compiled to code that can be executed at a target CPU. This is a very flexible solution, and has low costs for the data path and arithmetic operations. However, classical processor architectures cannot handle reactive control constructs, such as abortions, directly, and cannot concurrently observe multiple signals; therefore, handling these control constructs correctly, including priority resolution, turns out to be fairly expensive on classical software implementations. Moreover, the footprint (memory requirement) can be too large for low cost microcontrollers.
- **Hardware/Software Co-design**
  A *Co-design* (Figure 1(c)) implementation partitions a model into hardware and software components. The SW/HW interface is synthesized for internal communication. It combines the advantages of hardware and software implementations methods, but also inherits some shortages of the above. The co-design approach has been explored for example by the POLIS project [1].
- **Patched Processor**
  In the *Patched Processor* implementation approach (Figure 1(d)), a traditional microcontroller core is combined with a custom hardware block that extends the instruction set of the traditional microcontroller by certain new, Esterel-like instructions. An example has been demonstrated by P. S. Roop et al. [6,7,5,8]. An external block and a traditional microcontroller (FLIX or PIC) are combined to create a patched processor (REFLIX or RePIC). The external block captures some additional Esterel style instructions and gives appropriate actions. It results in a smaller footprint and shorter response times than a software implementation, and much more flexibility than hardware implementation.
  Although the patched processor implementation is a promising approach, it does have some shortcomings. First, the additional block cost is relatively high. The patch block's FPGA resources usage is almost equal to that of a full function microcontroller core (PIC16C84). Second, the function of Esterel style instructions is incomplete. For example, the valued signal and

the counter cannot be supported directly. Third, being limited by the existing microcontroller's architecture, it cannot implement fully reactive and priority action according to the original Esterel semantics. For example, the `abort` handling block in the patch provides an `abort` response mechanism which differs with Esterel `abort` or `weak abort`. Finally, the patch is inflexible; in many cases, the patched processor does not fit the function requirement of different embedded applications.

– **Esterel Processor**
  A final alternative is to design a processor for executing Esterel style instructions—and only those instructions—directly. The Kiel Esterel Processor (KEP), which is presented in this report, falls into this category. Unlike the Patched Processor solution, KEP is a truly reactive processor, which has a reactive kernel and adapted semi-custom (scalable) peripheral elements. It follows the exact definition of the Esterel, processes valued signal and counter directly, and reduces or extends the peripheral elements for making a processor series.

Figure 1 shows the differences of architecture of above methods. Table 1 gives a further comparison of the architectures.

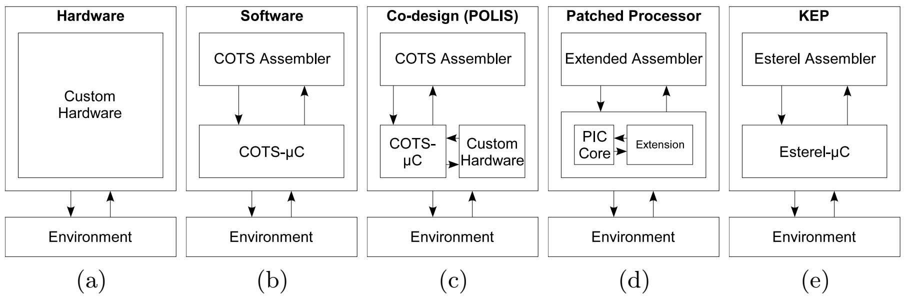

Fig. 1. The architecture comparison of implementation alternatives

Table 1. Comparison of Synthesis Approaches

	Hardware	Software	Co-design	Patched Processor	KEP
Speed	++	–	+	+	+
Flexibility	– –	++	–	+/–	+
Esterel Compliance	++	++	+/–	–	+/–
Cost	++	– –	–	–	+
Appl. Design Cycle	– –	++	+/–	++	++

++ = best; – – = worst. E. g. Cost ++ means very low production costs.

2 KEP (Kiel Esterel Processor)

2.1 Overview

An overview of the KEP architecture is shown in Figure 2. There are two main components.

1. The *Reactive Core*
   - reacts to responses of peripherals elements;
   - makes preemption decisions;
   - decodes the instruction and emits control signals;
   - configures peripherals elements.
2. The *Peripheral Elements*
   - define of the corresponding instruction functions;
   - are scalable (Semi-Custom), with configurable parameters;
   - contain counters.

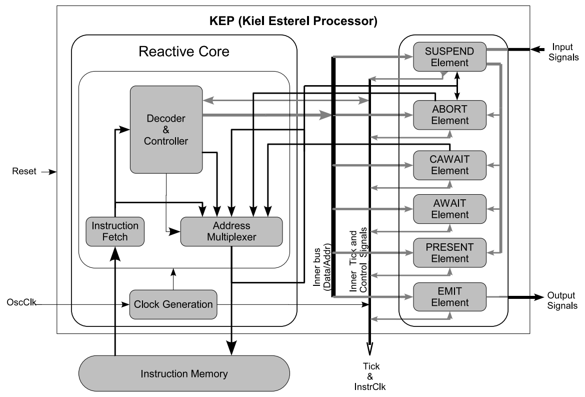

Fig. 2. The architecture overview of KEP

2.2 Instruction Set Architecture

The performance of KEP is predictable. All instructions can be executed in a single instruction cycle, except for the `ABORT` instruction. For `ABORT`, element (re-)configuration initially requires two cycles. However, our `abort` mechanism avoids ABORT Element (re)configuration during program execution. For most cases, it is just required after system resetting and executed once.

Trading off resource usage, speed, and performance, KEP employs a 24-bit wide instruction word with a separate 14-bit wide inner data bus and a 5-bit wide inner address bus. This choice extends the counter data range to 16383, which is enough for most reasonable requirements.

The KEP assembler language contains 15 instructions for representing 9 Esterel statements directly. Some other Esterel statements can be implemented by standard Esterel syntax translation. Due to integrated counters in peripherals elements, most of Esterel modules which do not contain variable data can be implemented by those instructions directly. A major limitation of KEP as of now is that as we execute Esterel-like instructions, it does not support concurrency directly, which stems from the sequential nature of execution on a single processor. One approach to handle this problem is to extend KEP to a multi-processor architecture for executing concurrent programs in parallel; see also Section 4.2. Another approach is to sequentialize concurrent Esterel programs into KEP-Assembler; see also Section 3.3.

Table 2 lists the complete instruction set.

Table 2. Overview of Instruction Set Architecture

Mnemonic, Operands	Description	Corresponding Esterel Statement
`ABORT n,S,endAddr(,startaddr)`	Configure the Watcher in the ABORT Element	`abort...when n S`
`WABORT n,S,endAddr(,startaddr)`	Configure the Watcher in the ABORT Element	`weak abort...when n S`
`AWAIT S`	Configure the AWAIT Element	`await S`
`AWAITN n,S`	Configure the AWAIT Element	`await n S`
`CAWAIT` S_n,S_n`startaddr`	Configure the CAWAIT Element	`await case`
`CAWAITE` S_n,S_n`startaddr`	Configure the CAWAIT Element for the last `case` in the list	`await case`
`EMIT S`	Emit the signal `S` and keep it during the current tick	`emit S`
`EMITD S,n`	Emit signal `S` with value `n` and keeping it during the current tick	`emit S(n)`
`HALT`	Halt the system	`halt`
`NOTHING`	Do nothing	`nothing`
`PRESENT S,elseAddr`	Test signal `S`, go to address `elseAddr` if `S` not present	`present S then ... else ... end present`
`SUSPEND S,endAddr`	Suspend when signal `S` is present	`suspend ... when S`
`SUSTAIN S`	Sustain signal `S`	`sustain S`
`SUSTAIND S,n`	Sustain valued signal `S`	`sustain S(n)`
`GOTO addr`	going to address `addr`	

2.3 Interface Signals

The top-level interface signals appear in Figure 3.

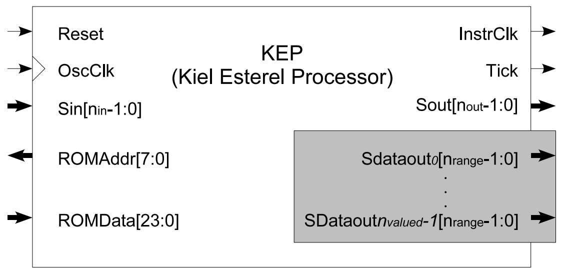

Fig. 3. KEP Interface Signals

2.4 ABORT Element

The `abort` statement is one of the most important statements in Esterel. It offers a preemptive abortion mechanism. An abortion statement kills its body when a delay elapses. For *strong abortion*, performed by `abort`, the body does not receive the control at abortion time. In other words, it ought to kill its body immediately. For *weak abortion*, performed by `weak abort`, the body receives the control for a last time at abortion time [3], i. e., it should receive control and execute the *remaining instantaneous statements*, and then kills its body.

For comparison, the abortion mechanism of RePIC executes the *current instruction* and then responds to the highest (outer) abortion priority whose sensitivity signal is present [5]. That means that the behavior of the RePIC `abort` corresponds to either weak or strong abort, depending on the context, as illustrated by the examples in Figures 4 and 5.

```
module Abort1:
  input A;
  output B,C;
  abort              (1)
    sustain A        (2)
  when S;
  emit B;            (3)
end module
```

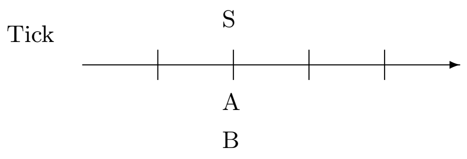

Fig. 4. `Abort1`: In this example, the RePIC behavior corresponds to a `weak abort`

```
module Abort2:
  input A,B,C,D;
  output E,F,G,H;
  weak abort          (1)
    abort             (2)
      await C;        (3)
      emit E;         (4)
      await D;        (5)
      emit F;         (6)
    when A;
    emit G;           (7)
    await D;          (8)
  when B;
  emit H;             (9)
  halt                (10)
end module
```

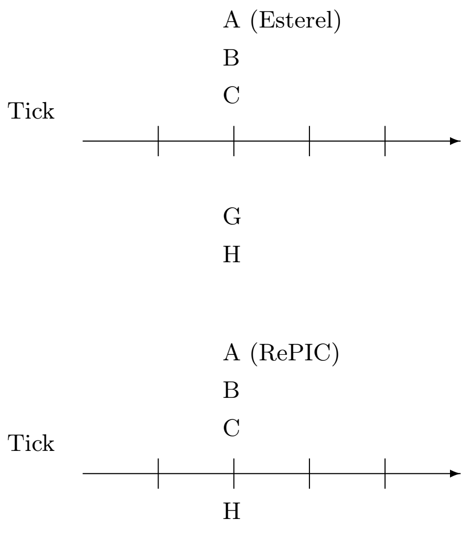

Fig. 5. `Abort2`: In this example, the RePIC behavior corresponds to a strong `abort`

We here present a different method to handle `abort` priority and preemption. Since the abort is active at the moment when its body is entered and it stays active until the entire body has finished execution or the preemption condition is satisfied, for the `Abort2` module, the abortion `A` or `B` is *enabled* when the PC is in its body, and *disabled* when the PC is out of its body. When abortion is *enabled*, the sensitive signal is watched and the module can react to it (is *active*). Otherwise, the signal is ignored.

We call this approach *Inside/Outside Abort Range Watching* (IOARW), and implement this with dedicated Watcher modules. A Watcher contains two functions. The first function watches the program counter (PC) and compares it with the corresponding abortion's start and end address, then decides whether this abortion ought to be *enabled* or not. We name it *Enable Watcher* (EW). The second function watches and counts down the relative signal, then decides if the abortion ought to kill its body or not. We name it *Trigger Watcher* (TW).

Every Watcher contains 5 reconfigurable parameters, i. e. StartAddr, EndAddr, SignalCoder, SignalCount, and AbortWeakFlag. The Watcher is configured by the KEP Core and then runs automatically. StartAddr and EndAddr assign the watching range of the Watcher. SignalCoder indicates which signal ought to be watched. If watched signal is valid on the Tick rising edge and Watcher is in the enabled state, SignalCount should be decremented. The Watcher emits a TW event to the environment when the counter value equals to zero. AbortWeakFlag, which indicates the abortion type, and EndAddr registers can be accessed by the environment. Once the Watcher changes its state from disable to enable, which means that the PC re-enters to the watching range, the SignalCount will reload the counter value automatically. This strategy ensures that the initial presence or absence of signal can be ignored in the starting instant, as required by the (non-immediate) await statement.

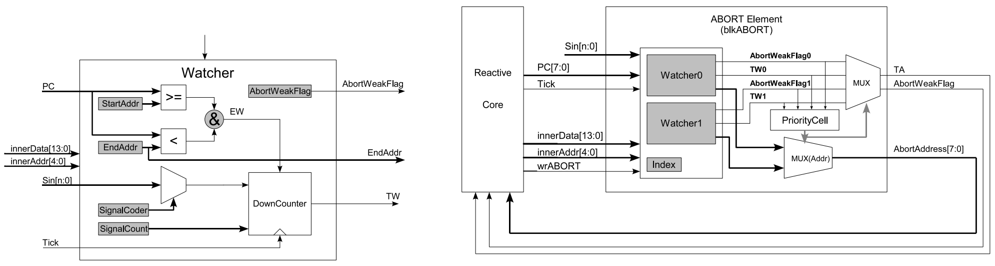

Fig. 6. Structure of the Watcher, and an ABORT Element with 2 Watchers

Due to the independence of every Watcher, a number of Watchers in an ABORT Element can run in parallel. Each Watcher works independently, according to the configured parameters. The distributed Watchers structure makes the architecture of ABORT Element clear, concise and scalable.

Figure 6 shows the architecture of an ABORT Element which includes two Watchers. After KEP resetting, the Index register takes the value of 0, and then the initial program writes `ABORT` parameters into Watchers via the innerData bus and the innerAddr bus of KEP. Considering the `abort` nest structure, we can conclude that the higher priority abortion has wider address range which covers over the lower one. Therefore, the first `ABORT` instruction will be assigned to the highest priority Watcher0. Then the Index will be increased by 1 and it will point to the next Watcher unless the `p` parameter in the instruction is `'1'`. The Watcher works immediately when its configuration processing is finished.

The PriorityCell checks the TW and AbortWeakFlag signals, and uses certain rules to judge which Watcher's output signals should be mapped to the Reactive Core via MUXs that are controlled by the PriorityCell.

The architecture shown in Figure 6 can handle a two-level `abort` nest (either weak or strong abortion). We use `Abort2` to explain how the KEP deals with different abortion types.

Figure 7 shows the mixed `abort/weak abort` nest example `Abort2`. At first, the Reactive Core configures Watcher0 and Watcher1 via two `ABORT` instructions. Watcher0 is enabled when the PC is between [0] and [9], and it responds to signal `B`. Watcher1 is enabled when the PC is between [2] and [7], and it responds to signal `A`. The Core executes the `AWAIT C` [4] instruction, and then stays here to wait for a sensed signal to occur.

```
%Esterel
module Abort2:
  input A,B,C,D;
  output E,F,G,H;
  weak abort          (1)
    abort             (2)
      await C;        (3)
      emit E;         (4)
      await D;        (5)
      emit F;         (6)
    when A;
    emit G;           (7)
    await D;          (8)
  when B;
  emit H;             (9)
  halt                (10)
end module
```

⇒

```
%KEP ASM
%Instruction                Address
INPUT A,B,C,D
OUTPUT E,F,G,H
   WABORT 1,B,A1         [0][1]
     ABORT 1,A,A0        [2][3]
       AWAIT C              [4]
       EMIT E               [5]
       AWAIT D              [6]
       EMIT F               [7]
A0:
     EMIT G                 [8]
     AWAIT D                [9]
A1:
   EMIT H                  [10]
   HALT                    [11]
```

Fig. 7. The program of example `Abort2`

The PC stays at instruction [4] until any of the signals `A`, `B`, or `C` occurs. If `A`, `B`, and `C` occur simultaneously, both Watchers are enabled, so they are both active. The AW0 and AW1 are present at the same time. The AbortWeakFlag0 is '1' and AbortWeakFlag1 is '0' to indicate different abortion types.

The PriorityCell responds to TW occurrences. In this case, strong abortion of `B` has higher priority and gets preemption first. The PriorityCell maps Watcher1's outputs to that of the ABORT Element, so the ABORT Element's TA is '1' to denote there is an active abortion, the AbortAddress equals 8 (the next instruction address behind the body of abortion `A`), and the AbortWeakFlag is '0' for indicating strong abortion type.

The AWAIT Element sets rdAWAIT to '1' to denote that `AWAIT` is terminated. The Reactive Core checks the TA, AbortWeakFlag and rdAWAIT signals synchronously. Since it is a strong abortion, the Core responds immediately and the AbortAddress is mapped to the PC via the Address Mutiplexer. The KEP jumps to [8].

Address [8] is out of Watcher1's watching range, and then the Watcher1 is disabled immediately. Watcher0 is still in enabled state, so the Watcher0's outputs are mapped to ABORT Element's.

The Reactive Core checks the TA and AbortWeakFlag, the AbortWeakFlag of the ABORT Element is '1' because it comes from Watcher0. The Core will not pay attention to the weak abortion event until all following instantaneous instructions (e. g. `EMIT` and `PRESENT`) are finished. So the following concurrent instruction `EMIT G` [8] is executed, the Core fetches the next instruction `AWAIT D` [9].

AWAIT is a sequencing instruction. The Reactive Core will not execute it, but responds to the weak abortion. The KEP jumps to [10] (the next instruction address behind the body of abortion B). Now both of Watchers are disabled. TA is '0' to indicate no active abortion. KEP executes EMIT H [10] and HALT [11].

The above case illustrates how KEP deals with different abortion types. The KEP's abort (nest) execution trace and result are according to the Esterel semantics.

3 Compilation

3.1 KEP Assembler Compiler

The compiler KEPcmp (KEP Assembler Compiler) compiles a KEP assembly language file into executable codes. An important novelty of the KEPcmp is ABORT Element initialization.

We use Abort2a shown in Figure 8 as an illustrating example.

```
%Esterel
module Abort2a:
  input A,B,C,D;
  output E,F,G,H;
  loop
    weak abort          (1)
      abort             (2)
        await C;        (3)
        emit E;         (4)
        await D;        (5)
        emit F;         (6)
      when A;
      emit G;           (7)
      await D;          (8)
    when B;
    emit H;             (9)
    end loop             (10)
end module
```

⇒

```
%KEP ASM
INPUT A,B,C,D
OUTPUT E,F,G,H

A3:
   WABORT 1,B,A1        (1)
     ABORT 1,A,A0       (2)
       AWAIT C          (3)
       EMIT E           (4)
       AWAIT D          (5)
       EMIT F           (6)
A0:
     EMIT G             (7)
     AWAIT D            (8)
A1:
   EMIT H               (9)
   GOTO A3             (10)
```

Fig. 8. The program of example Abort2a

According to KEP's ABORT implementation, the Watchers in the ABORT Element need to be configured only once. So the compiler transforms the source file to the following style:

At first, KEPcmp marks the start address of an abortion body, i. e. $0 and $1, and then those parameters are combined with the original ABORT instructions. The new combined ABORT instruction contains all parameters for configuring the

```
%KEPcmp Temporary 1
INPUT A,B,C,D
OUTPUT E,F,G,H
A3:
$1:WABORT 1,B,A1,$1      (1)
$0:  ABORT 1,A,A0,$0     (2)
       AWAIT C           (3)
       EMIT E            (4)
       AWAIT D           (5)
       EMIT F            (6)
A0:
     EMIT G              (7)
     AWAIT D             (8)
A1:
   EMIT H                (9)
   GOTO A3              (10)
```

⇒

```
%KEPcmp Temporary 2
INPUT A,B,C,D
OUTPUT E,F,G,H
    WABORT 1,B,A1,$1     (1)
    ABORT 1,A,A0,$0      (2)
A3: $1: $0:
           AWAIT C       (3)
           EMIT E        (4)
           AWAIT D       (5)
           EMIT F        (6)
A0:
     EMIT G              (7)
     AWAIT D             (8)
A1:
   EMIT H                (9)
   GOTO A3              (10)
```

Fig. 9. The transformation of example `Abort2a`

Watchers. So the compiler moves them to the beginning of the module, out of the `loop` body.

When KEP runs the first iteration of the loop in `Abort2a` (assume no abortion preemption occurs at this time), 10 instructions need to be executed, and 12 instruction cycles are required at least. The execution trace is the same as that of the traditional execution trace, e. g. RePIC's method. Then it goes back to `AWAIT C` (3). In the next iteration period, the `ABORT` instructions need not be executed anymore. The execution period decreased. However, the KEPcmp also supports the traditional method if desired.

3.2 Esterel Syntax Translation Rules

The Esterel language can be reduced to a set of kernel statements, from which other statements can be derived. There are different ways to define this set of kernel statements; for example, for pure Esterel (which excludes valued signals), one set of kernel statements is `nothing`, `emit`, `pause`, *present*, *suspend*, `;` (sequencing), `loop`, `||` (parallel), `trap`, `exit`, and `signal` [3]. A processor which would offer instructions to implement each of these kernel statements would be powerful enough to process any Esterel program, after translation of all *derived* statements into the corresponding kernel statements (this translation is a rather straightforward, syntactical process). Such an instruction set would be powerful and compact—however, it would not necessarily be the most adequate for the efficient execution of real programs.

Instead of implementing just the kernel statements, the design of the KEP instruction set aimed to allow direct execution of the most frequent Esterel instructions, aiming for maximal efficiency without excessive redundancy. The

KEP does not offer the concurrency operator (||) (see also Section 3.3); however, it does offer valued signals.

Figure 10 gives some examples of how to translate Esterel programs into KEP assembler programs, where p and q are arbitrary code fragments.

```
emit A;
await B;
emit C(20);
await 25 D;
```

⇒

```
EMIT A
AWAIT B
EMITD C,20
AWAITN 25,D
```

```
emit G;
await
     case A do
         emit D;
     case B do
         emit E;
     case C do
         emit F;
end await;
emit G;
```

⇒

```
     EMIT G
     CAWAIT A,A0
     CAWAIT B,A1
     CAWAITE C,A2
A0:
     EMIT D
     GOTO A3
A1:
     EMIT E
     GOTO A3
A2:
     EMIT F
     GOTO A3
A3:
     EMIT G
```

```
loop
    p
end loop
```

⇒

```
A0:
    p
GOTO A0
```

```
await immediate A
```

⇒

```
     PRESENT A,A0
     GOTO A1
A0:
     AWAIT A
A1:
```

Fig. 10. Example translations from Esterel programs into KEP assembler

3.3 Sequentialization

The KEP does offer low-level parallelism for example by operating multiple Watcher modules in parallel (which cannot be done, for example, by a traditional sequential processor). However, as already mentioned, the KEP overall still operates sequentially, and therefore cannot support Esterel's general concurrency operator (||). One approach to overcome this limitation is to eliminate the concurrency operator by applying another source-level transformation which would *sequentialize* the Esterel program before feeding it to the KEP.

In principle, sequentialization is a well-known procedure for synthesizing Esterel programs into software; traditionally, however, this is not done at the Esterel-level, but rather as part of the compilation process. Furthermore, we do not have to eliminate all concurrency from a given Esterel-program; for example, we may still watch multiple signals for an `await case` or for a nested `abort` concurrently. In terms of automata, we do not need a completely flat automaton, we still allow nesting; however, we do not allow concurrent sub-states (AND-states).

To illustrate, Figure 11 and 12 shows how the canonical ABRO module can be sequentialized. The sequentialized ABRO_SEQ module can be translated to KEP assembler program by using the syntax translation rules mentioned in section 3.2. In this example, the sequentialization is rather straightforward and does not lead to a dramatic increase in program size; in general, however, the translation is less straightforward, and may lead to exponential code increase. How practical this approach really is still needs to be investigated.

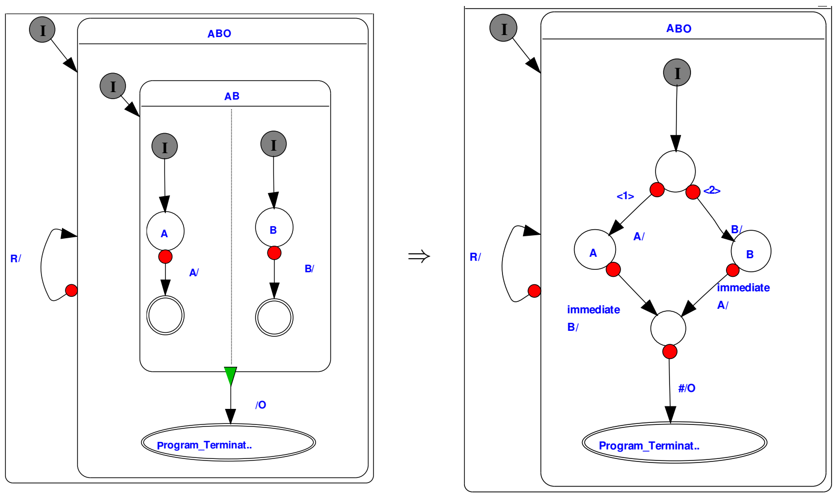

Fig. 11. Sequentialization automation of the ABRO module

4 Conclusion

4.1 Comparison of KEP and RePIC

We created three KEP processors, which have different elements for targeting different applications. KEPV0.1-A and KEPV0.1-B can deal with most of Esterel modules which do not contain variable data, e. g. the RUNNER, the ATM, etc. KEPV0.1-C allows a direct comparison with RePIC.

```
module ABRO_PAR:
 loop
    abort
      [await A
        ||
      await B];
      emit O
    each R;
 end loop
```

⇒

```
module ABRO_SEQ:
 loop
    abort
      await
        case A do
          await immediate B;
        case B do
          await immediate A;
      end await;
      emit O
    halt
    when R;
 end loop
```

Fig. 12. Sequentialization program of the ABRO module

Table 3 shows the basic differences between KEP and RePIC. Table 4 gives a simple performance comparison. The KEP's resource usage is just 20% - 30% of that of RePIC (even just about half of the RePIC patch). The obtained speedup is 1.4x or better. Considering some details of our approach, e. g. ABORT Element configuring method, one can expect further performance improvements.

Table 3. The differences between KEP and RePIC

KEP	RePIC
Reactive kernel	Traditional microcontroller + patch block
Semi-custom (scalable) peripheral elements	Fixed patch block
Supports real `abort / weak abort` (nest)	Supports approximate strong `abort` (nest) type
Configurable `abort` nest levels	Fixed 4 `abort` nest levels
Configurable `await case` branches	Fixed 2 `await case` branches
Up to 31 input signals and 31 output signals	16 input, 12 Output and 13 common I/O
Supports pure and valued signals	Only supports pure signals
Supports counter (e. g. `await 100 S`, `abort...when 50 S`) directly, the range is up to 16383 (14 bits)	Does not support counter
Supports `suspend` (incompletely)	Does not support `suspend`

4.2 Summary

For the direct Esterel program execution, existing projects (REFLIX or RePIC) use a patch strategy, which adds a patch block on a traditional processor. This

Table 4. Comparison of KEP series and RePIC

	KEPV0.1-A	KEPV0.1-B	KEPV0.1-C	RePIC
AWAIT CASE Number	2	2	2	2
ABORT Nest	2	5	4	4
Counter Value Range	1	100	1	1
Input	8	8	16	16
Output	7	7	12	12
Logic Cell Count	400	732	604	2068[1]
Max Osc Freq (MHz)	49.15	43.74	43.56	40.27
Instruction Freq[2] (MHz)	16.4	14.6	14.5	10.1

[1] 1082 Logic Cells for PIC.
[2] RePIC uses four clock cycles for composing an instruction cycle, but the KEP Version 0.1 uses only three clock cycles. When they run on the same clock frequency, the KEP's instruction cycle period is just 75% of that of the RePIC's.

is a convenient approach, but has some shortages. This report presents KEP Version 0.1, a new semi-custom, configurable reactive processor for the direct execution of Esterel programs. It consists of a reactive core and scalable peripheral elements, which operate concurrently to the reactive core. KEP supports standard Esterel statements. Valued signals and counter functions in Esterel statements can be executed by KEP directly. We created a Reactive Core as the kernel of KEP. Due to its control path and its cooperation with elements, KEP obeys exact Esterel (preemption and priority) rules, e. g. `abort/weak abort` (nests). Furthermore, the KEP series offers various processors with different elements, and users can select the suitable one to avoid excess or lack of processor capability.

As an initial prototype, the current KEP can be optimized further. Limitations of the current version are:

– There is no `||` statement.
– Logic and arithmetic expression are not supported.
– The `SUSPEND` cannot work well when it is in an abortion body. It has to be on the top of the nest levels; i. e. a `SUSPEND-ABORT` nest works well, but an `ABORT-SUSPEND` nest will screen the corresponding abortion sensitive signal when the sensitive signals of the `ABORT` and the `SUSPEND` are present simultaneously.

4.3 Outlook

Possible improvements that we plan to investigate are:

– Proper handling of `abort/suspend` nests
– Extend the instruction word to 32-bit to
  • The counter data range to 65535 (16-bit)

  - The instruction memory address to 1024 (10-bit)
  - Flexible Watcher reusing mechanism
  - More signals in total, etc.
- A data handling layer for dealing with data path.
- New instructions, such as `ADD`, `SUB`, `?S`, etc.
- A reconfigurable Logic Element. For example, an `AWAIT A OR B` statement can be implemented by an `AWAIT A_B` instruction and an OR gate in this Element. The inputs of gate are signals `A` and `B`, the output is `A_B`. The reconfigurable structure element can improve the performance of KEP.
- Compiling Esterel to KEP assembler.
- Handling `||`
  - An extension of structure of KEP to fit multi-processor architectures, to handle Esterel program with several parallel threads in the future.
  - Sequentializing `||`

References

1. Felice Balarin, Paolo Giusto, Attila Jurecska, Claudio Passerone, Ellen M. Sentovich, Bassam Tabbara, Massimiliano Chiodo, Harry Hsieh, Luciono Lavagno, Alberto Sangiovanni-Vincentelli, and Kei Suzuki. *Hardware-Software Co-Design of Embedded Systems, The POLIS Approach*. Kluwer Academic Publishers, April 1997.
2. Gerard Berry. Esterel on hardware. *Philosophical Transactions of the Royal Society of London*, 339:87–104, 1992.
3. Gerard Berry. *The Esterel v5 Language Primer*. Draft Book, 1999.
4. Gerard Berry and Georges Gonthier. The Esterel synchronous programming language: Design, semantics, implementation. *Science of Computer Programming*, 19(2):87–152, 1992.
5. 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. School of Engineering Report No. 612, University of Auckland, 2004.
6. P. S. Roop, Z. Salcic, M. Biglari-Abhari, and A. Bigdeli. A New Reactive Processor with Architecture Support for Control Dominated Embedded Systems. In *IEEE International Conference on VLSI Design*, pages 189–194. IEEE CS Press, January 2003.
7. P. S. Roop, Z. Salcic, and M. W. S. Dayaratne. Towards Direct Execution of Esterel Programs on Reactive Processors. In *4th ACM International Conference on Embedded Software (EMSOFT 04)*, Pisa, Italy, September 2004.
8. Z. Salcic, P. S. Roop, M. Biglari-Abhari, and A. Bigdeli. REFLIX: A Processor Core with Native Support for Control Dominated Embedded Applications. *Elsevier Journal of Microprocessors and Microsystems*, 28:13–25, 2004.