

Direct Execution of Esterel Using Reactive Microprocessors

M. W. Sajeewa Dayaratne¹ Partha S Roop² Zoran Salcic³

*Department of Electrical and Computer Engineering
University of Auckland
New Zealand*

Abstract

This paper proposes a new reactive multiprocessor for direct execution of Esterel programs. While Esterel programs have been compiled to both hardware and software, each has its own limitations. Hardware compilers, while being efficient, are quite inflexible. Software compilers, on the other hand, are flexible, but produce both bulky and inefficient code. This is because of the generation of generic code independent of the underlying architecture. This paper proposes an intermediate approach, using *architecture specific compilation*. Firstly, a suitable multiprocessor (called EMPEROR) is designed where each individual processor has native Esterel-like instructions. These processors are synchronised exactly like Esterel using a Thread Control Unit (TCU) that generates the Esterel tick event using *barrier synchronisation*. Subsequently, a compiler is developed that takes advantage of the features available on EMPEROR to generate direct machine code from the Esterel source. Benchmarking results using the proposed approach reveals significant improvement in code quality compared to conventional compilation, both in terms of code size and execution time.

Key words: Esterel, Reactive Processors, Architecture Specific Compilation

1 Introduction

Embedded systems can be defined as application specific digital systems which normally reside within a larger electronic or mechanical environment. Embedded systems constantly interact with their environment, either by reacting to external events or by routine processing of input information.

¹ Email: sajeewa@ieee.org

² Email: p.roop@auckland.ac.nz

³ Email: z.salcic@auckland.ac.nz

Embedded systems are currently designed and developed using numerous technologies. These include automated design environments that use Hardware Description Languages(HDL) such as VHDL [23,26] and Verilog [28], and system level design languages such as SpecC [29] and SystemC [8].

Another set of languages known as *synchronous languages* have also gained recent popularity in the area of embedded systems design [3]. These includes languages such as as SIGNAL [20,15], Argos [22], LUSTRE [17]. Synchronous languages are based on a solid mathematical foundation, thus allowing designs written in these languages to be formally verified using some form of mathematical proof. This make them ideal for designing safety critical systems. This paper focuses on one such synchronous language known as Esterel [6]. Esterel has built in features for describing reactive behaviour which makes it an ideal language for designing reactive embedded systems. The main features of Esterel are:

Synchronous Model of Time An Esterel program samples its environment at discrete time intervals known as *ticks*(this is analogous to the system clock in a hardware circuit). All *instantaneous instructions* complete their execution within a given *tick*.

Preemption and Priority Resolution Esterel has instructions that allows a body of code to be preempted based on an event or a combination of events. This mechanism is known as an *abort* and nesting of abort bodies creates static priorities of events.

Concurrency Esterel has support for synchronous concurrency. It employs a synchronous broadcast mechanism for communication between concurrent threads. This implies that an event generated in one thread is instantaneously (within the same tick) broadcast to all other threads.

Zero Delay Model The Esterel language has a zero delay model of time. This means that the output resulting from an action occurs in the same logical time instant as the corresponding input event.

The features of the language are best described using an example as shown in Figure 1. Any Esterel program such as this one can be made up of a collection of modules, each of which is a basic programming unit. Every module has an interface declaration part(lines 2 & 3) which declares the signals and sensors in the environment. Signals carry control information and values and are either present or absent in a given *tick* (a pure signal carries only control information and has no value). Sensors, on the other hand, are valued objects that can be read at any point in time.

Following the interface declaration part is the body of the program. Almost all real-life Esterel programs have an infinite loop, as shown in this example(line 5-30), surrounding the main body of the program. This describes a non-terminating control loop, which is common for most embedded devices.

The execution of Esterel programs is driven by a logical *tick* event. On each *tick*, the program samples the environment and performs certain reactions

```

1 module slap:
2   input A, B, C;
3   output S, T, U, V, W, X, Y, Z;
4   signal S1, S2, S3 in
5     loop
6       [await tick;
7       emit S2;
8       present S1 then emit X end present;
9       await B;
10      emit S3
11      ||
12      await S2;
13      emit S1;
14      await A;
15      emit Y;
16      await tick;
17      emit V
18      ||
19      await S1;
20      emit Z;
21      await tick;
22      emit W;
23      abort
24        emit T;
25        emit S;
26        await C ;
27        sustain U
28      when S3;
29      emit V;]
30   end loop
31 end signal
32 end module

```

Fig. 1. A Sample Esterel Program

based on the environmental input signals. However, all reactions occur during the same *tick* as the corresponding input event. Thus a zero delay, in terms of logical time, between an input and its corresponding output may be observed.

This example consists of three threads. Threads in Esterel are delimited using the `||` symbol, which describes synchronous parallel execution. The first

thread starts with an *await tick* statement. The *await* statement in Esterel is a delay construct. It delays execution of the program until the given signal(s) have occurred. Being a delay statement, it also marks the end of the current *tick* and therefore is also known as a *tick delimiting* statement [11].

The *await tick* statement is a variant of the standard *await* mentioned earlier. Instead of waiting for particular signal, this instruction simply waits for the completion of one Esterel *tick*. The *pause* statement in Esterel is an abbreviation for the *await tick* statement. When this statement (line 6) is executed the thread simply waits for the completion of one Esterel *tick*, prior to continuing with the remainder of the thread.

The next statement in this example is an *emit* statement. *Emit* is used in Esterel to output both internal and external signals. Internal signals are defined using the *signal* statement as shown in line 4. These signals are only used for communication between threads and are not visible to the external world. External signals are defined in the interface declaration described previously.

In contrast to the *await* instruction, *emit* is an instantaneous instruction. The execution of *emit* is similar to that of a combinational logic block in hardware. Due to its instantaneous nature, lines 7 and 8 are executed at the same logical time instant.

The next instruction of interest is the *present* statement on line 8. This is used in Esterel to provide support for conditional execution based on the *presence* or *absence* of certain signals. In this example the *present S1* statement checks if signal S1 is present during the current *tick* and emits signal X if this is true.

Preemption, which is another important feature of Esterel can also be illustrated by using this example (lines 23 to 28). The *abort* statement maybe used to create preemptive Esterel programs. The example in Figure 1 shows the syntax of the *abort* instruction. The two statements, *abort* and *when S3*, encompass the *abort body*. In this example the abort body contains only four statements, but this can be as large as desired and can even contain multiple threads.

An abort body is terminated when the signal or a signal expression (specified by the *when* part of abort) becomes *present* (or if execution leaves the abort body). In the current example this happens when signal S3 is emitted. At this instant, the body of the abort is preempted and its execution continues from line 29. If the specified abortion condition does not happen before the last statement of the body is executed, the body dies naturally and execution continues from the statement immediately following the abort body. It is possible for abort instructions to be nested within each other. This allows implicit static priorities to be assigned to each preemption condition. In Esterel, the outermost abort has the highest priority and the priority level decreases on each subsequent inner abort.

Due to features of Esterel such as concurrency, synchronous broadcast, preemption and the zero delay model of time, compiling Esterel becomes a

complicated process. Conventional Esterel compilers either generate software code or a hardware circuit for a given Esterel program. However, an intermediate approach is also possible. This is the central theme of the work presented in this paper. The idea is based on *architecture specific compilation* [5]. First a customised microprocessor architecture is developed which provides native support for some of the key features of Esterel. An Esterel compiler is then developed which is able to take advantage of the features of the architecture when generating machine code.

The remainder of this paper is structured as follows. Section 2 presents some of the current compilation techniques available for Esterel. The motivation behind our approach for executing Esterel is then discussed in Section 3. This is followed by a presentation of the proposed multiprocessor architecture in Section 4. Section 5 presents our approach for compiling Esterel on this architecture. This is followed by some benchmarking results in Section 6. Some conclusions and avenues for future work are presented in Section 7.

2 Related Work

The efficient implementation of Esterel designs has been a topic of interest within the research community for some time. Traditionally, Esterel models of embedded systems were either synthesised into software for a standard microprocessor [2,10,1,21,7], or used to generate hardware circuits [12,14] (Figure 2). A combination of both software synthesis and hardware circuit generation was also possible through codesign-design, using tools such as Polis.

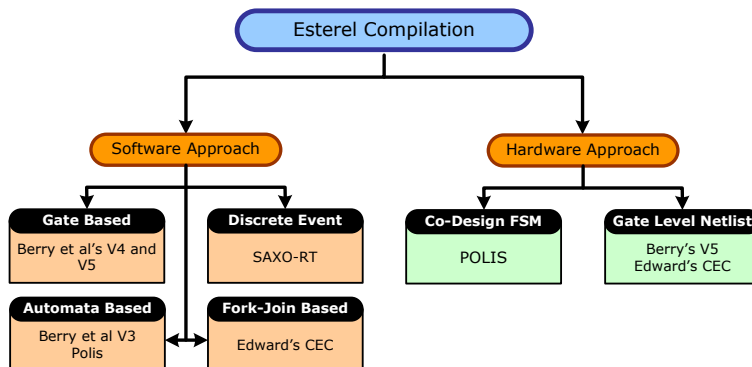


Fig. 2. Current Esterel Compilation Techniques

Software compilers for Esterel can be categorised into four types based on their algorithms and data structures. These include automata based compilers such as Berry's V3 [4], gate based compilers such as Berry's V4 and V5, discrete event based compilers such as SAXO-RT [7] and fork-join based compilers such as Edward's CEC [10,12].

Automata based compiler translates Esterel by first exhaustively simulating the entire state space of the program through a product automata. This allows it to identify what can possibly happen in each state. It then generates

minimal code for each of the states based on the results of the above simulation. Due to this minimalistic approach, the generated code is very fast for simulation. The main drawback of this method, however, is the exponential growth of the code size as the number of threads in a program increases. This is mainly due to the compiler producing a separate branching program for each state [11]. Therefore such a technique becomes impractical for large programs.

The Polis compiler, although also an automata based compiler, tries to reduce the problem of increasing code size by sharing code between states using state space reduction techniques such as binary decision diagrams (BDD) [16]. Although this method reduces the code size compared to standard automata compilers, it is limited to programs where it is practical to enumerate the states [11].

The second type of software compilers are the gate based compilers such as Berry's V4 and V5. These compilers first translate a given Esterel program into a netlist of boolean logic gates, and then generates a leveled compiled-code simulator for it [11]. The generated code uses minimal instructions for mapping code from the original Esterel source. Furthermore, unlike automata compilers, no code duplication occurs in these compilers, leading to much more compact code. The main drawback, however, is the significant increase in execution time. This is mainly due to the fact that the generated code contains many idle sections which consume computation time [11].

The third type of compilers are the discrete event based compilers such as SAXO-RT [7] developed by France Telecom. SAXO-RT breaks the Esterel program into smaller functional blocks which are then dispatched and executed by a fixed scheduler.

Fork-join based Esterel compilers such as Edward's Columbia Esterel Compiler (CEC) [10,12] fall into the fourth category of Esterel compilers. CEC, on an average, produces the most efficient and compact code among all the current Esterel compilers. It employs an algorithm that synthesises a sequential program from a concurrent Esterel program by using fork and join statements within the generated C code [10,11]. One of the main reasons for the good performance of CEC is due to efficient internal representation of Esterel programs using concurrent control flow graph (CCFG).

The main inefficiencies of the above compilers are due to the requirement of indirectly simulating Esterel concurrency (through some form of scheduling), inefficient mapping of signals and sensors to ports on the microcontroller (through additional *reaction code*) and indirect mapping of Esterel preemption constructs through interrupts, polling or some suspend-resume based approach.

3 Motivation for Reactive Processors

Although synchronous languages are ideal for describing and verifying reactive embedded systems, executable models generated by current compilers are either very inefficient or inflexible. The generation of efficient and compact code becomes an important issue for resource constrained embedded systems.

The main objective of conventional Esterel compilers is to generate the most efficient and compact source code possible. However, since the Esterel compiler is unaware of specific features of the processor architecture at compile time, it becomes difficult to optimise the generated code for a given microprocessor.

An example of this problem can be seen during the compilation of Esterel’s *await* instruction, which may be implemented (in some cases) as a busy waiting (signal polling) loop until the given signal has occurred. As microprocessors have no special support for such signal polling, this instruction gets typically implemented using a collection of instructions which repeatedly check for the existence of the given signal. This significantly affects both the the code size and the execution time of the program. A suspend-resume based implementation of *await* is also possible for multithreaded programs. This will still be quite inefficient compared to a direct machine instruction based implementation as advocated here.

One of the main contributors to the compiler overhead is the preemption mechanism in Esterel. Standard compilers map the abort based preemption construct to machine instructions by either exploiting the interrupt mechanism on the processor or by checking for preemption conditions on each *tick*. However, unlike interrupts, Esterel’s *abort* never resumes execution from the point it was preempted. Therefore, execution is a lot simpler than that of an interrupt service routine as there is no need to save the program context.

Traditional architectures, however, do not have a mechanism similar to that of an *abort*. This results in an increase in code size and execution time of such programs. If multiple concurrent *aborts* or nested *aborts* are present, the code size is further increased due to the additional code that is needed to handle abort priority resolution.

Based on these observations it is evident that compilation to an architecture that supports features of Esterel has the potential to produce more efficient and compact code than currently possible. This can be done by exploiting the features of the architecture during compile time.

4 EMPEROR: Reactive Multiprocessor

RePIC [24] was an initial attempt at creating an architecture which had support for some of the reactive features of Esterel. A preliminary investigation and design of RePIC was done in [9]. Subsequently, a detailed design, implementation and benchmarking of a reactive processor supporting single

threaded Esterel programs was done [24]. RePIC was based on the commercial PIC microprocessor. PIC is a proven robust design that is specifically targeted at control-dominated embedded applications [2].

RePIC only supported single threaded Esterel programs and was unable to provide an architecture which would allow mapping of complete Esterel. A big contribution to the complexity in compiling Esterel comes from the need to handle concurrent execution and synchronous communication semantics of the language. In order to support full Esterel, a truly parallel architecture is needed. This led to the development of an application specific multiprocessor called *EMPEROR* (*Embedded MultiProcessor supporting Esterel Reactive Operations*).

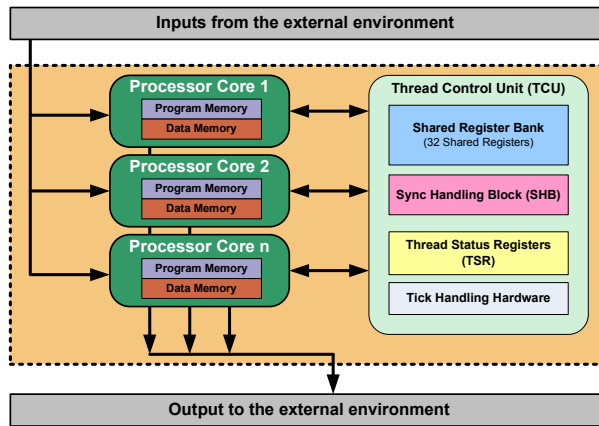


Fig. 3. Overview of the EMPEROR Architecture

The *EMPEROR* architecture consists of a number of reactive processor cores (based on RePIC) connected to each other through a *Thread Control Unit (TCU)* as shown in Figure 3. Each processor core has a reactive instruction set architecture similar to that of RePIC. The design of RePIC, however, has been significantly modified to cater to additional requirements which arise in a multiprocessor environment such as EMPEROR.

4.1 The Processor Core

In comparison with the PIC processor the following modifications were done during the design of the EMPEROR processor core:

- 8 new instructions were added to provide the features required to execute concurrent Esterel programs in a multiprocessor environment.
- A synchronous Abort Handling Block (AHB) was added to allow the processor core to preempt instruction execution. This was a slight modification to the existing AHB within RePIC to allow for *weak* and *strong* aborts of Esterel.
- Hardware was added to support the latching of input and output signals. These are only cleared at the completion of each Esterel *tick*.

- External address and data busses were added to enable communication between processors and the shared register bank within the TCU.
- New ports were added to enable control information to be passed between processors and the TCU.
- Support for *valued signals* was added through the use of a single bit signal presence line and 8 bit signal input and output ports.

All Esterel threads are required to execute in lock step with a logical event known as a *tick*. In order to support such functionality in EMPEROR, a *barrier synchronisation* mechanism is employed. In such a synchronisation mechanism, two or more concurrent processes (threads) block after reaching a previously defined barrier. Once all processes have blocked, the barrier is raised and execution continues [18].

The barrier in the case of EMPEROR marks the completion of an Esterel *tick*. When a given processor reaches a tick delimiting instruction (which signals the completion of the *local tick* for that processor) ⁴, it signals the TCU of it's state and pauses until the barrier is raised. Once all processors have reached the barrier the TCU generates a resume event and raises the barrier. This allows the processors to continue execution of code belonging to any subsequent *tick*.

4.2 The Thread Control Unit

The TCU provides support for some of the core features of Esterel. It is responsible for handling communication and synchronisation between these processors, which enables the preservation of Esterel's execution semantics. It also contains a shared register bank, used for passing shared signals between threads, and hardware components, which provide support for the synchronisation instructions executed within EMPEROR.

The TCU connects all processor cores with each other and provides communication and synchronisation between them. It contains a dedicated communication port for each processor core, and in its current configuration supports up to **32 processors**.

The internal functions of the TCU can be broadly broken down into three separate modules, each dealing with a certain aspect of synchronous execution. The first module provides hardware support for executing *FORK* and *JOIN* instructions. These instructions are used to provide fork and join support for each processor. The second module contains the shared register bank used for inter-thread communication. This component also provides support for the *SYNC* instruction, which is used to synchronise reading and writing of signals. The third module is known as the *tick generator*. This module is responsible for keeping track of the *local tick* signals arriving from each processor. It also generates the *global tick* signal when the barrier synchronisation point has

⁴ The *AWAIT* instruction is used as a local tick delimiter in EMPEROR.

been reached by all participating processors.

5 The EMPEROR Esterel Compiler (EEC)

Although EMPEROR supports some of the key reactive features of Esterel in hardware, traditional compilers are unable to take advantage of these features during compile time since they are designed to generate architecture independent C code. Therefore, in order to take advantage the features, a new compiler called the EMPEROR Esterel Compiler (or EEC) was created.

Most traditional Esterel compilers analyse a given Esterel source file and generates a functionally equivalent program using C. EEC uses a different approach (based on architecture-specific compilation) and it is able to directly translate a given Esterel program onto machine code. This allows for a more natural translation from Esterel to EMPEROR assembly with minimal overheads.

This compilation technique is partly inspired by the Columbia Esterel Compiler(CEC) [12]. One of the main reasons for the good performance of CEC is due to its unique internal representation of Esterel.

CEC uses Concurrent Control Flow Graphs (CCFG) as the intermediate representation of an Esterel program. The control flow of a CCFG, by definition, is concurrent. It uses *fork* and *join* nodes to provide new branches for each thread in a given program. The graph uses a set of primitive nodes to support the control dominated features of the Esterel language. Consider, for example, the CCFG corresponding to two threads of Figure 1 which appears in Figure 4⁵.

The CCFG representation is semantically very close to both the Esterel source and the generated code which has a close resemblance to the original Esterel program. However, when compiling on an architecture such as EMPEROR, which has hardware support for features of Esterel, the resemblance of the generated code to Esterel can be further increased. This can be achieved by improving on certain shortcomings of traditional CCFGs, such as:

- CCFGs are executed from top to bottom for each Esterel ‘tick’. This process can be non intuitive when generating sequential programs. Moreover, decisions on the state of execution of the program (which node or branch to execute in the next tick) is done by using state variables. Many state variables are required for complex programs leading to the complexity of CCFG.
- CCFGs use generic nodes to implement special features of Esterel (such as *abort* and *await*). However such mappings are not needed if these features are available in hardware.

⁵ Only two threads are used since the complexity of the CCFG for all three threads inhibits the clarity of this document

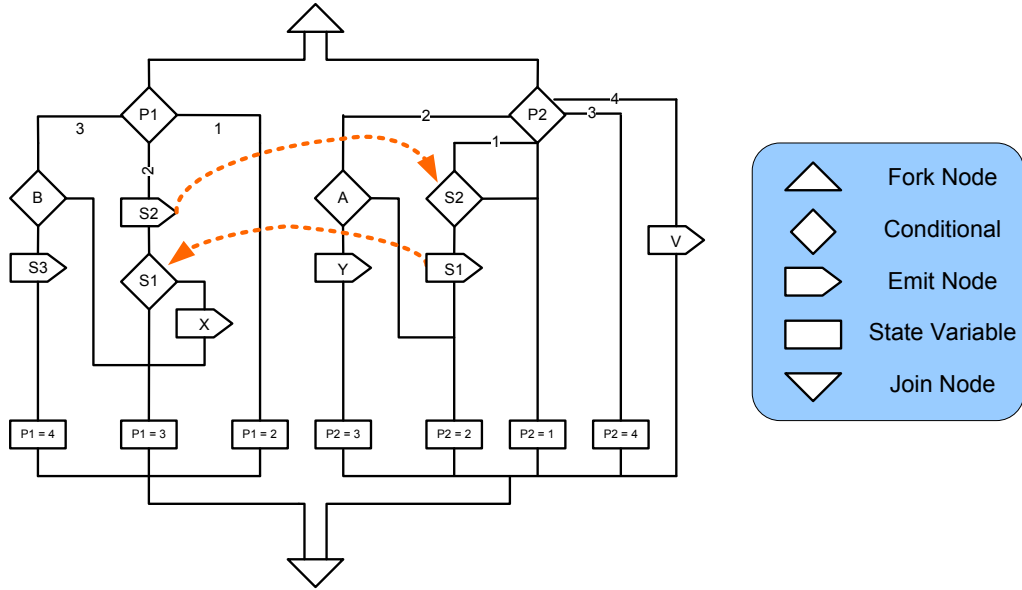


Fig. 4. The Concurrent Control Flow Graph for two threads from the example in Figure 1

In this paper a simplification of CCFG called Unrolled Concurrent Control Flow Graphs (UCCFG) is introduced. UCCFG has a closer resemblance to the Esterel source than CCFG and its execution semantics is simpler. UCCFGs use the reactive instruction set of EMPEROR to generate a simplified control flow graph. This is done by avoiding the need to loop through the graph and hence removes the overhead caused by state variables.

Such unrolling becomes possible due to the addition of architecture specific nodes in the UCCFG. For example, the EMPEROR architecture contains an abort handling block for implementing Esterel aborts. Therefore, the EMPEROR UCCFG can provide a primitive *abort* node to implement the abort instruction. Furthermore, special nodes are also introduced for *await* and *sustain*.

Execution in EMPEROR is done using tick delimiting instruction to mark local ticks, which are already denoted differently in UCCFG. Hence there is no need for looping back at the end of every tick and restarting the graph. This also removes the need for state variables. This is possible as the actual tick synchronisation among Esterel threads is performed in the architecture and the need for such synchronisation in the data structure is removed.

Figure 5(b) depicts the UCCFG for the Esterel program from Figure 1. The control flows in a clear and intuitive manner and maps directly to the original Esterel program. It's important to note that a given UCCFG is specific to the underlying processor architecture, in this case, EMPEROR.

UCCFGs are made up of all the traditional CCFG nodes in combination with the new set of architecture specific nodes:

abort start & abort end The *abort start* and *abort end* nodes correspond

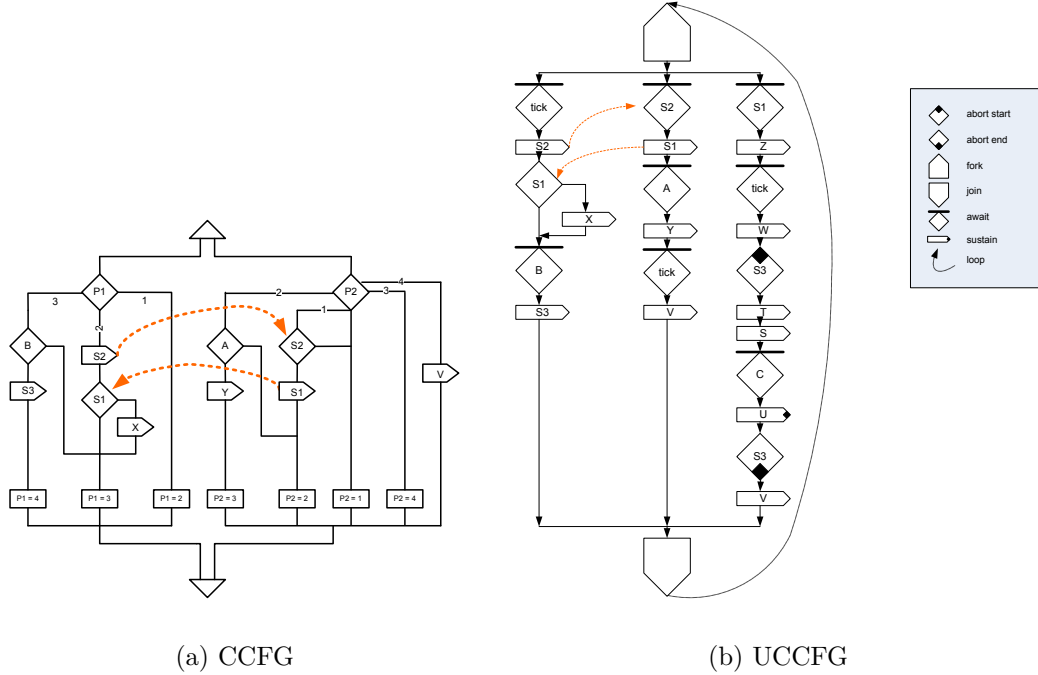


Fig. 5. The CCFG & UCCFG corresponding to the program in Figure 1

to the *abort . . . when* construct in Esterel. This can be directly mapped from the Esterel source code and is implemented by the corresponding machine instructions in the processor. The type of the abort (*weak* or *strong*) is stored as a parameter of the abort start node. Such an implementation on a CCFG would require checking of the abort condition on every iteration of the graph.

await In CCFGs awaits are implemented by checking for the relevant signal at the start of each cycle. In EMPEROR, however, this is implemented by a blocking instruction which waits until the given signal occurs. Therefore in the UCCFG, awaits can be implemented using their own nodes, which stop execution until the specified signal occurs. An await can be an await for a signal or a pause (which delays for one tick called, *await tick*).

sustain The *sustain* instruction in Esterel emits the given signal during every *tick* in which it is executed. In a CCFG this is achieved by emitting the given signal during each cycle. A special instruction has been added to EMPEROR to handle sustains. This instruction causes the processor to emit the given signal until terminated by an enclosing *abort* instruction.

The execution of the above UCCFG begins at the topmost *fork* node. Control is then forked on to three separate threads. The the first thread begins with a special await node, known as an *await tick*. Such nodes do not wait for a particular signal, but wait for the completion of one Esterel tick. The second node in this thread is an emit node, which is emitting the signal S2 in this example. The emit node behaves in the same way as its counterpart

in the CCFG. The following two nodes in this thread are also taken directly from the CCFG. The first node checks for the presence of signal S1 and if present, executes the emit X node. The await B node is another addition to the UCCFG. It pauses control until the signal B becomes present.

The second thread does not contain any new nodes and executes in a similar manner to the first. However, the third and final thread has two more of the new additions to UCCFGs. These are the *abort* and *sustain* nodes. The *start abort* and *end abort* nodes used here only affect the control flow when the abortion signals are present. When the specified signal, in this case S3, is emitted control simply jumps to the *end abort* node. Such transparent control flow is possible due to the support from the Abort Handling Block in EMPEROR.

Once the UCCFG has been extracted for a given Esterel program, dependencies between signal reads and signal writes can also be identified. For example, there is a dependency between the *emit S1* node in thread 2 and the *present S1* which checks for the same signal in thread 1. Such dependencies are shown using the arrows and arcs in Figure 5(b), similar to CCFGs.

Once the UCCFG has been generated from the Esterel source, the EEC compiler is able to use it to produce machine instructions. The compiler performs this conversion in two stages:

- (i) The UCCFG has to be split into separate threads so that the code generated for each thread is executed on a separate processor. This is done by inserting fork and join instructions based on the nodes in the UCCFG.
- (ii) The compiler then translates the nodes to their corresponding machine instructions.

During this process the compiler also looks for any read / write dependencies identified earlier using the dependency arcs (which are part of the UCCFG). These dependencies are used in conjunction with the splitting of the UCCFG to produce code for each processor core. Synchronization between dependent threads is achieved by inserting appropriate SYNC instructions (this is explained later).

The nodes of a UCCFG are mapped to EMPEROR assembly as follows.

emit This node is used to represent a signal emission. The *emit* node is the simplest node to translate. It results in a direct mapping to the EMPEROR EMIT instruction.

present The *present* node is used for conditional control flow. Using this node we are able to ensure that certain parts of the graph are only executed if the specified signal is present. This node is mapped onto the two EMPEROR instructions, PRESENT and GOTO. If this node was identified as having a read-write dependency, then an additional SYNC instruction is added to ensure synchronisation with the signal writer.

fork this node represents the forking of control. The *fork* node can be

mapped directly onto the EMPEROR FORK instruction. The identifier(ID) which was assigned to the node during the UCCFG generation is used as the *sync ID*.

join Similar to the *fork* node, the *join* node gets mapped to the EMPEROR JOIN instruction. Likewise, the join identifier is used as the *sync ID* parameter for this instruction.

await The *await* node waits until a given signal has occurred. This makes it a signal reader as well as a ‘tick’ delimiter. Therefore the mapping to EMPEROR assembly needs to cater for both these functions. This is achieved by mapping this node onto 3 assembly instructions, AWAIT, PRESENT and GOTO. The AWAIT instruction is responsible for generating the *local tick* event, while the PRESENT and GOTO instructions are used to implement the signal check. If the said signal is not present the GOTO instruction jumps back to the AWAIT, and this loop continues until the signal waited on is present. This nicely implement either one tick or multiple tick delay for await depending on whether the signal is present in the current instant or becomes present in a future instant. Similar to the *present* node, an additional SYNC instruction is added if a read-write dependency exists.

sustain The *sustain* node emits the specified signal continuously during each subsequent *tick*. Once control reaches this node, it cannot progress any further unless preempted by an enclosing *abort* node. Therefore the mapping of sustain becomes straightforward, resulting in a continuous loop which emits the specified signal in each tick. This is achieved by the use of an AWAIT and EMIT combination which is repeated through the use of a GOTO instruction.

loop arc The loop back arc in a UCCFG is simply mapped to a GOTO instruction which sends execution back to a specified node.

abort start/abort end These nodes are used to represent the start and end of an abort body. When an abort start node is encountered it is translated onto the EMPEROR LDAADDR and ABORT instructions. These instructions are used to register the details of this *abort* within the Abort Handling Block. The continuation address, which is required by LDAADDR, can only be calculated based on the position of the *abort end* node. Therefore two passes of the graph are required for compilation. Apart from aiding in calculating the continuation address, the *abort end* node does not insert any additional instructions.

Moreover, during the translation of nodes between *abort start* and *abort end* an additional CHKABORT instruction is inserted appropriately. If the enclosing abort was a ‘strong abort’ then the CHKABORT instruction is inserted immediately following any AWAIT instruction. If it was a ‘weak abort’ then CHKABORT is inserted prior to any AWAIT instruction. The CKHABORT ensures Esterel-like execution of *abort* by allowing either the body to be terminated instantly (strong abort) or allowing the body one

last instant of execution during the instant of abortion (weak abort). The compiler makes an exception to this rule in the instance where an await instruction immediately follows a weak abort. In this case a CHKABORT instruction is not inserted before the AWAIT (since a new tick begins with the body and the previous tick was completed outside the body).

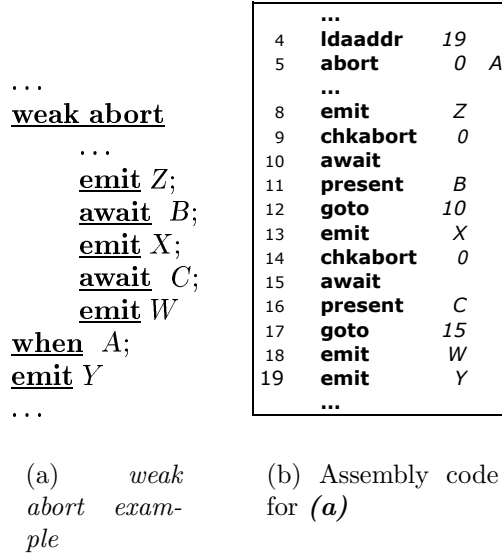


Fig. 6. Sample Code for Esterel *weak* Aborts

Consider the code fragment shown in Figure 6. The Esterel code in Figure 6(a) shows a sample program containing a *weak abort*. The abort body is preempted when signal A is present. Assume that this program is currently executing *await B*.

If both B and A are present in the next *tick* the following observation can be made. The program executes *emit X* followed by *emit Y*. This happens because *weak aborts* allow the body to execute during the abortion instance.

The EMPEROR assembly code shown in Figure 6(b) illustrates how this behaviour is implemented in EMPEROR. The CHKABORT instruction is added prior to the AWAIT instruction in Figure 6(b), this ensures that abort conditions are only checked just prior to completing the tick, thus allowing the body to execute during the abortion instant.

Figure 7 shows a graphical representation of this conversion process for the Esterel example in Figure 1. The mapping of all the three threads from the UCCFG in Figure 5(b) are shown together with the corresponding EMPEROR assembly code.

6 Benchmarking Results

EMPEROR and EEC performance have been evaluated using several programs from the Auckland Reactive Benchmark (ARE-Bench) [24]. This benchmark

the same set of applications. Note that each processor in the EMPEROR multiprocessor is RePIC, which is a PIC extension with additional reactive instructions.

The compactness of the code produced by EEC was compared with four other popular Esterel compilers. These include, the Columbia Esterel Compiler(CEC) 0.3 [12], Esterel Studio 4.0 [14] and the Esterel compiler V3 and V5 [4]. All optimisation were turned on, both at the Esterel to C Code generation stage and also during the C to assembly translation.

Benchmark	Maximum Number of Threads	Maximum Nesting of Aborts	Columbia Esterel Compiler 0.3	Esterl Studio 4.0	Esterel Compiler V5	Esterel Compiler V3	Emperor
Driver	1	1	510	1540	1588	1575	47
Elevator	1	0	378	1147	1192	474	65
Filter	1	0	109	308	303	231	12
LONG_ACC_CAL	1	0	101	504	518	366	30
LONG_ACC_DER	1	1	225	554	578	431	36
PumpController	1	2	692	811	945	346	28
SpeedSense	1	0	193	493	501	352	25
Startup	1	1	140	869	1143	305	29
TCPReceive	1	0	301	498	499	293	24
TCPTransmit	1	0	681	588	593	260	28
TrafficLight	1	2	107	1008	1188	373	41
VER_ACC_CAL	1	0	260	306	301	229	14
VER_ACC_DIAG	1	0	260	675	661	522	32
abcd	4	0	1399	2208	2383	2267	162
ATDS-100	3	0	11289	17827	19735	5816	326
Runner	2	0	594	979	1035	500	36
mejia	8	1	3264	5124	5243	8435	186
ww	12	1	6248	7234	8754	10889	364
tcint	6	1	11147	11432	10833	409670	278

Table 1
Code Size Generated (Number of WORDS) by Various Compilers

Table 1 presents a numerical summary of the compilation results from ARE-Bench. For a majority of the benchmarks, the EMPEROR compiler (EEC) produced code which was on an average 80% more compact than that generated by the other compilers. CEC performed the best among all compilers examined (other than EEC). However, the EMPEROR Esterel compiler produced code which was in the range of 60% to 97% more compact than that produced by CEC.

Although CEC produces the most compact code for most applications, no single compiler performed uniformly best across all benchmarks. In order to present a fairer comparison, EEC generated code was compared with the best performing compiler for each benchmark application. These results are presented in Table 2. These results show that EEC performs significantly better regardless of the compiler being compared. The percentage reduction in code size is consistently high across all applications, with an average reduction of 87.4%.

Benchmark	Best Performing Compiler	Generated Code Size	Emperor Code Size	% Reduction in Code Size
Driver	CEC	510	47	90.78%
Elevator	CEC	378	65	82.80%
Filter	CEC	109	12	88.99%
LONG_ACC_CAL	CEC	101	30	70.30%
LONG_ACC_DER	CEC	225	36	84.00%
PumpController	Esterel V3	346	28	91.91%
SpeedSense	CEC	193	25	87.05%
Startup	CEC	140	29	79.29%
TCPReceive	Esterel V3	293	24	91.81%
TCPTransmit	Esterel V3	260	28	89.23%
TrafficLight	CEC	107	41	61.68%
VER_ACC_CAL	Esterel V3	229	14	93.89%
VER_ACC_DIAG	CEC	260	32	87.69%
abcd	CEC	1399	162	88.42%
ATDS-100	Esterel V3	5816	326	94.39%
Runner	Esterel V3	500	36	92.80%
mejia	CEC	3264	186	94.30%
ww	CEC	6248	364	94.17%
tcint	Esterel V5	10833	278	97.43%
			Average	87.42%

Table 2
Code Size Generated by EMPEROR vs the Best Esterel compiler for each benchmark

The compilation results presented in the previous section clearly identifies EEC as generating the most compact machine code for our benchmark applications. However, the compactness of the code cannot be used as the sole measure of the quality of the produced code. In order to properly determine the performance impact of the EMPEROR architecture, the execution time for these benchmarks were also analysed.

The EMPEROR architecture was used to calculate the execution time of the code produced by the new compiler, while the unmodified PIC processor was used during calculations for the conventional compilers.

Due to the reactive nature of these applications, the execution time for a given program depends heavily on the sequence and timing of input signals. In order to compare the execution times for EMPEROR and its non reactive counterpart(PIC), certain assumptions were made regarding these inputs. The first was an assumption of ideal environmental input signal reads. This meant that a given signal will be present the moment it is to be read by either a *present* or an *await* instruction. Preemption signals were also considered to occur at a random moment in time. The same time instance was used across the comparison for both EMPEROR and PIC.

EMPEROR showed a significant speed up over the fastest Esterel compiler across all benchmark programs. This is clearly visible in the graphical representation of these results as shown in Figure 8. Here the leftmost y-axis displays the execution time in μs while the rightmost y-axis shows the speedup. The x-axis on the top displays the name of the Esterel compiler which generated the fastest code for the current benchmark. An overall average speedup of 12.4 was achieved using EMPEROR.

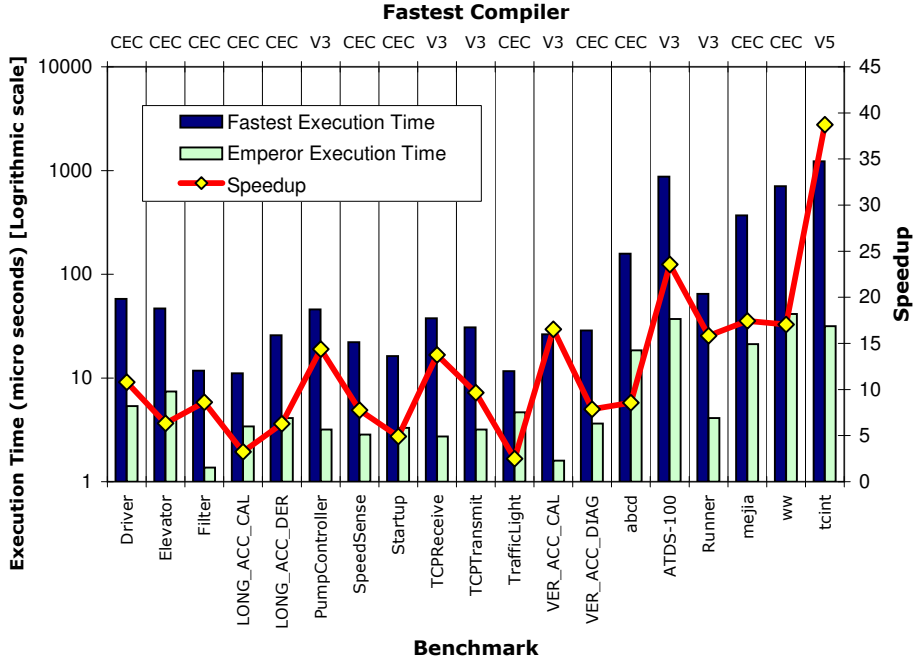


Fig. 8. Execution time comparison between EMPEROR and the fastest Esterel compiler currently available (Note that V3 performs best for programs with few threads)

7 Conclusions

Traditional compilation techniques for Esterel either generate generic software code or hardware netlist. While the advantage of software code is flexibility of compilation to any architecture, hardware code offers efficiency. Both have their limitations. Since the software code is generic and emulates Esterel concurrency, it is often bulky and inefficient. The hardware code, on the other hand, is inflexible as new hardware needs to be generated for every Esterel program and any changes to an existing program. This paper proposes an intermediate solution using *architecture specific* [5] compilation. The main idea is to first design an architecture that has native support for features of Esterel. Subsequently, a compiler is designed to take specific advantage of this new architecture. Based on this idea, a reactive multiprocessor architecture called EMPEROR was designed. A compiler called EEC (the EMPEROR Esterel Compiler) was also developed to generate code from Esterel to directly execute on EEC. Several Esterel programs from ARE-Bench were used to evaluate the performance of EMPEROR and EEC. Benchmarking results indicate that the proposed approach offers a significant increase in code quality, both in terms of code compactness and execution time.

Although EMPEROR provides support for true Esterel concurrency, the current architecture has certain limitations. The architecture currently only supports compilation of non cyclic programs. An existing compiler such as Esterel V5 is used to eliminate any cyclic programs prior to using EEC.

While Esterel does not allow instantaneous loops, some instantaneous statements may be executed more than once in a cycle if another concurrent thread throws an exception using a trap (for details see page 175, Figure 8 in [11]). This feature, known as *reincarnation*, is not currently handled by EEC which compiles traps using weak aborts.

A further limitation arises based on the number of threads in the input program. EMPEROR requires all concurrent Esterel threads to be assigned to individual processor cores. Therefore the maximum number of simultaneous threads that can be executed in EMPEROR is governed by the number of available processor cores (which is limited to 32 in the current design). This limitation may be overcome using:

- Multiple concurrent threads may be assigned to a single processor together with a customised scheduler. The scheduler would be responsible for the interleaved execution of threads during each Esterel *tick*. This would involve emulating the existing interface to the Thread Control Unit by sharing the same set of control and data lines among all threads within the processor. Alternatively, the TCU may be redesigned to support execution of multiple threads within a single processor core.
- Another approach to overcome this limitation would be to employ an automata compilation technique to minimise the number of threads. This can be done prior to invoking the EEC compiler. A similar technique would also be to merge smaller threads together.

Some directions for future research include codesign using reactive processors, formal verification of EMPEROR like architectures and multithreaded reactive processors.

References

- [1] Balarin, F. and M. Chiodo, *Software synthesis for complex reactive embedded systems*, in: *Computer Design, 1999. (ICCD '99) International Conference on*, 1999, pp. 634–639.
- [2] Balarin, F., M. Chiodo, P. Giusto, H. Hsieh, A. Jurecska, L. Lavagno, A. Sangiovanni-Vincentelli, E. Sentovich and K. Suzuki, *Synthesis of software programs for embedded control applications*, *Computer-Aided Design of Integrated Circuits and Systems*, IEEE Transactions on **18** (1999), pp. 834–849.
- [3] Benveniste, A., P. Caspi, S. A. Edwards, N. Halbwachs, P. Le Guernic and R. de Simone, *The synchronous languages 12 years later*, *Proceedings of the IEEE* **91** (2003), pp. 64–83.
- [4] Berry, G. and G. Gonthier, *The Esterel synchronous programming language: Design, semantics, implementation*, *Science of Computer Programming* **19** (1992), pp. 87–152.

- [5] Bhartacharyya, S., R. Leupers and P. Marwedel, *Software synthesis and code generation for signal processing systems*, Circuits and Systems II: Analog and Digital Signal Processing, IEEE Transactions on [see also Circuits and Systems II: Express Briefs, IEEE Transactions on] **47** (2000), pp. 849–875.
- [6] Boussinot, F. and R. de Simone, *The ESTEREL language*, Proceedings of the IEEE **79** (1991), pp. 1293–1304.
- [7] Closse, E., M. Poize, J. Pulou, P. Venier and D. Weil, *SAXO-RT: Interpreting esterel semantic on a sequential execution structure*, Electronic Notes in Theoretical Computer Science **65** (2002).
- [8] Corp, V. S., *System C website* (2003).
URL <http://www.systemc.org>
- [9] Edmund, C., T. Joyce, M. W. S. Dayaratne, P. S. Roop and Z. Salcic, *RePIC: A new processor architecture supporting direct esterel execution*, Technical Report 612, University of Auckland, School of Engineering (2004).
- [10] Edwards, S. A., *Compiling Esterel into sequential code*, in: *Hardware/Software Codesign, 1999. (CODES '99) Proceedings of the Seventh International Workshop on*, 1999, pp. 147–151.
- [11] Edwards, S. A., *An Esterel compiler for large control-dominated systems*, Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on **21** (2002), pp. 169–183.
- [12] Edwards, S. A., *CEC: The Colombia Esterel Compiler* (2003).
URL <http://www1.cs.columbia.edu/~sedwards/cec/>
- [13] Edwards, S. A., *The estbench esterel benchmark suite* (2003).
URL <http://www1.cs.columbia.edu/~sedwards/software.html>
- [14] Esterel-Technologies, *Esterel technologies website*.
URL <http://www.esterel-technologies.com>
- [15] Fleureau, J., P. Le Parc and L. Marce, *SIGNAL: a language for low level implementation*, in: *Southeastcon '95. 'Visualize the Future'. Proceedings.*, IEEE, 1995, pp. 176–182.
- [16] Fujita, M., H. Fujisawa and N. Kawato, *Evaluation and improvement of boolean comparison method based on binary decision diagrams*, in: *Computer-Aided Design, 1988. ICCAD-88. Digest of Technical Papers.*, IEEE International Conference on, 1988, pp. 2–5.
- [17] Halbwachs, N., P. Caspi, P. Raymond and D. Pilaud, *The synchronous data flow programming language LUSTRE*, Proceedings of the IEEE **79** (1991), pp. 1305–1320.
- [18] Hawick, K. and S. Elmohamed, *High performance computing and communications glossary* (2004).
URL <http://www.npac.syr.edu/nse/hpccgloss/hpccgloss.html>

- [19] Lee, L.-C. and H. Lin, *Embedded processor core design to support real-time operating systems*, Technical Report Part 4 Project Report, University of Auckland, Faculty of Engineering (2003).
- [20] LeGuernic, P., T. Gautier, M. Le Borgne and C. Le Maire, *Programming real-time applications with SIGNAL*, Proceedings of the IEEE **79** (1991), pp. 1321–1336.
- [21] Leupers, R., *Code generation for embedded processors*, in: *System Synthesis, 2000. Proceedings. The 13th International Symposium on*, 2000, pp. 173–178.
- [22] Maraninchi, F., *The argos language: Graphical representation of automata and description of reactive systems*, in: *IEEE Workshop on Visual Languages*, Kobe, Japan, 1991.
- [23] Navabi, Z., *Elements of VHDL for description of hardware: a tutorial view*, in: *ASIC Seminar and Exhibit, 1990. Proceedings., Third Annual IEEE*, 1990, pp. T/7.1–T/7.9.
- [24] Roop, P. S., Z. Salcic and M. W. S. Dayaratne, *Towards direct execution of estereL programs on reactive processors*, in: *Fourth ACM International Conference on Embedded Software, Proceedings.* (2004), pp. 240–248.
- [25] Salcic, Z., P. S. Roop, M. Biglari-Abhari and A. Bigdeli, *REFLIX: A processor core with native support for control dominated embedded applications*, *Microprocessors and Microsystems* **28** (2004), pp. 13–25.
- [26] Shahdad, M., *An overview of VHDL language and technology*, in: *23rd ACM/IEEE conference on Design automation* (1986), pp. 320–326.
- [27] Software, H., *HI-TECH PICC Lite compiler* (2004).
URL <http://www.htsoft.com/products/PICCLite.php>
- [28] Thomas, D. E. and P. R. Moorby, “The Verilog hardware description language (4th ed.)” Kluwer Academic Publishers, 1998.
- [29] UCI, *SpecC System* (2004).
URL <http://www.ics.uci.edu/~specc/>