Appeared in the Proceedings of IFIP Advanced Research Working Conference on Correct Hardware Design and Verification Methods (CHARME'97), Montréal, Québec, Canada, 16 - 18 October 1997

Proving the Correctness of the Interlock Mechanism in Processor Design

Xiaoshan Li [1], *Antonio Cau*[2], *Ben Moszkowski* [1],
Nick Coleman[1] *and Hussein Zedan*[2]
[1] *Department of Electrical and Electronic Engineering, University of Newcastle upon Tyne, Newcastle NE1 7RU, UK*
Email:{xiaoshan.li, b.c.moszkowski, j.n.coleman}@ncl.ac.uk
[2] *Software Technology Research Laboratory, Department of Computer Science, De Montfort University, Leicester LE1 9BH, UK*
Email: {cau, zedan}@dmu.ac.uk

Abstract

In this paper, Interval Temporal Logic (ITL) is used to specify and verify the event processor EP/3, which is a multi-threaded pipeline processor capable of executing parallel programs. We first give the high level specification of the EP/3 with emphasis on the interlock mechanism. The interlock mechanism is used in processor design especially for dealing with pipeline conflict problems. We prove that the specification satisfies certain safety and liveness properties. An advantage of ITL is that it has an executable part, i.e., we can simulate a specification before proving properties about it. This will help us to get the right specification.

Keywords

Interval Temporal Logic, Processor Verification, Executable Specification, Compositionality

1 INTRODUCTION

As is well known, the complexity of current VLSI has been increasing very rapidly. Traditional simulation methods cannot exhaustively test all cases so that the correctness of products cannot be guaranteed. Formal methods is therefore used to deal with this problem. Formal methods are based on mathematical methods and thus can ensure the correctness in a very rigorous way. We choose ITL as our basic formalism.

Our selection of ITL is based on a number of points. It is a flexible notation for both propositional and first-order reasoning about periods of time found in descrip-

tions of hardware and software systems. Unlike most temporal logics, ITL can handle both sequential and parallel composition and offers powerful and extensible specification and proof techniques for reasoning about properties involving safety, liveness and projected time(Moszkowski 1994). Timing constraints are expressible and furthermore most imperative programming constructs can be viewed as formulas in a slightly modified version of ITL (Cau and Zedan 1997). Tempura provides an executable framework for developing and experimenting with suitable ITL specifications. In addition, ITL and its mature executable subset Tempura (Moszkowski 1986) have been extensively used to specify the properties of real-time systems where the primitive circuits can directly be represented by a set of simple temporal formulae.

We will use ITL to specify and verify the correctness of the interlock control mechanism of an experimental CPU prototype, the Event Processor EP/3 (Coleman 1993). The EP/3 is a non-von Neumann data-flow pipeline processing element designed for high performance over a range of general computing tasks. The interesting aspect of the EP/3 processor architecture is the integration of multi-threading, pipelining and data flow mechanisms. This is reflected in the manner in which instructions are executed (cf.Section 4). Using the multi-threading technique, program parallelism is exploited by interleaving threads onto successive pipeline stages. The processor may also be used as an element in a multiprocessor system. Three different simulations to the EP/3 have been obtained independently (Coleman 1993, Cau *et al*. 1996, Li and Coleman 1996) which indicates that the general design of the EP/3 is correct. To increase the level of trustworthiness in the design, formal specification and correctness verification were sought in particular for the interlock control mechanism. The interlock mechanism is used to control the multi-thread pipeline during the execution of conditional and multi-destination instructions.

The approach we take in this paper is that we first simulate (execute) the specification before proving its correctness. The specification we get is the abstract version of (Cau *et al*. 1996). The correctness proof should be done in a compositional way adopting rules developed in (Moszkowski 1994, Moszkowski 1995, Moszkowski 1996). Some work on the formal verification of microprogrammed processors has already been done (Cohn 1988, Windley 1995, Tahar and Kumar 1995). However, they concentrated on the instruction level design and are thus on a lower level than the approach presented here. Furthermore the considered microprocessors have a different architecture from our EP/3.

To get an even higher level confidence the generated proofs are mechanically checked using the Prototype Verification System (PVS) (Rushby 1993) for which we have developed an ITL proof checking library (Cau and Moszkowski 1996, Cau *et al*. 1997).

The structure of this paper is as follows. Section 2 presents a brief overview of ITL. The general architecture of the EP/3 is described in section 3. We give the specification and the simulation of the EP/3 in section 4, the properties of the EP/3 in section 5 and the verification that the specification satisfies those properties in section 6. We give conclusions and discuss related issues in section 7.

2 INTERVAL TEMPORAL LOGIC

Interval temporal logic is a state based logic which can be used to specify and verify hardware and software systems. Especially it can describe both qualitative and quantitative requirements of systems. Here we only give a brief introduction of ITL. For more details, please refer to B. Moszkowski's papers (Moszkowski 1985, Moszkowski 1986, Moszkowski 1994).

An interval σ is considered to be a (in)finite sequence of states $\sigma_0 \ldots \sigma_i \ldots \sigma_n$, where a state σ_i is a mapping from the set of variables *Var* to the set of values *Val*. The length $|\sigma|$ of an interval $\sigma_0 \ldots \sigma_n$ is equal to n (one less than the number of states in the interval, i.e., a one state interval has length 0).

The main feature of ITL is the temporal operator ';' (*chop*). In ITL a formula $f_1 ; f_2$ holds on an interval $\sigma_0 \ldots \sigma_n$ means that there exists an i, $0 \leq i \leq n$, such that f_1 and f_2 hold on respectively the intervals $\sigma_0 \ldots \sigma_i$ and $\sigma_i \ldots \sigma_n$.

The syntax of *expressions* and *formulas* in ITL is defined in Table 1, where i denotes an integer; x is a static (global) variable which doesn't change within an interval; A is a state variable which can change within an interval; g is an n-ary function; p is an n-ary predicate.

Table 1 Syntax of ITL

Expressions	
$exp ::=$	$i \mid x \mid A \mid g(exp_1, \ldots, exp_n)$
Formulas	
$f ::=$	$p(exp_1, \ldots, exp_n) \mid \neg f \mid f_1 \wedge f_2 \mid \forall v \bullet f \mid \mathsf{skip} \mid f_1 ; f_2$

The informal semantics of the most interesting constructs are as follows:

- $\forall v \bullet f$: for all v such that f holds.
- skip: unit interval (length 1).
- $f_1 ; f_2$: holds if the interval can be decomposed ("chopped") into a prefix and suffix interval, such that f_1 holds over the prefix and f_2 over the suffix.

The *Chop* operator has some similarities with the sequence operator of program languages. Using the *chop* operator, the general temporal operators $\Box$ (read *always*) and $\Diamond$ (read *sometimes*) can be defined.

$$\Diamond f \;\widehat{=}\; \mathit{finite} ; f$$
$$\Box f \;\widehat{=}\; \neg \Diamond \neg f$$

where *finite* is defined in table 2. We use a special variable *len* to express the interval

length. The following formulae about interval length such as $len = n$, $len > n$ can be defined by the skip and the chop operator. Other useful abbreviations are defined in Table 2.

Table 2 Frequently used abbreviations

true	$\widehat{=}$	$0 = 0$	true value
$f_1 \vee f_2$	$\widehat{=}$	$\neg(\neg f_1 \wedge \neg f_2)$	f_1 or f_2
$f_1 \supset f_2$	$\widehat{=}$	$\neg f_1 \vee f_2$	f_1 implies f_2
$f_1 \equiv f_2$	$\widehat{=}$	$(f_1 \supset f_2) \wedge (f_2 \supset f_1)$	f_1 equivalent f_2
$\exists v \bullet f$	$\widehat{=}$	$\neg\forall v \bullet \neg f$	there exists a v s.t. f
$\bigcirc f$	$\widehat{=}$	skip$; f$	next f
inf	$\widehat{=}$	*true*$;$*false*	infinite interval
finite	$\widehat{=}$	$\neg$*inf*	finite interval
more	$\widehat{=}$	$\bigcirc$*true*	non-empty interval
empty	$\widehat{=}$	$\neg$*more*	empty interval
if f_0 then f_1 else f_2	$\widehat{=}$	$(f_0 \wedge f_1) \vee (\neg f_0 \wedge f_2)$	if then else
fin f	$\widehat{=}$	$\Box$(empty $\supset f$)	final state
$A := exp$	$\widehat{=}$	$\bigcirc A = exp$	assignment

2.1 Compositional proof rule

In (Moszkowski 1994, Moszkowski 1995, Moszkowski 1996) several compositional proof rules were developed. Due to lack of space, we will not give a full exposition to the compositionality theory and we thus refer the reader to published work. However, we will use the following compositional proof rule to prove the termination and liveness properties of the EP/3.

$$\mathcal{CR}\ \frac{\begin{array}{l}\vdash\ w \wedge S\ \supset\ T \wedge \mathit{fin}(w') \\ \vdash\ w' \wedge S'\ \supset\ T' \wedge \mathit{fin}(w'')\end{array}}{\vdash\ w \wedge (S;S')\ \supset\ (T;T') \wedge \mathit{fin}(w'')}$$

where w, w' and w'' are formulas in conventional first-order logic containing no temporal operators and describing properties of individual states. The turn-style $\vdash$ means that the formula to its right is provable in ITL axiom system. The first lemma states that if w is true in an interval's initial state and S is true on the interval then w' is true in the final state and T is true on the interval. The rule shows how to compose two such lemmas proved about input-output behavior of S, T, S' and T' into a corresponding lemma for $S;S'$ and $T;T'$.

3 THE EP/3 ARCHITECTURE

Here we give a brief introduction to the architecture of the EP/3. For simplification, we omit some details. The EP/3 processor consists of seven main components *Cache*, *Alu1*, *Alu2*, *Memory*, *Stack*, *Inst* (Instruction Issue) and *Memadd* (Memory Address) as shown in Fig 1. These components are connected by buses and control signals, such as the *Py* (Processor Highway) and the *Ilock* signal.

Instructions in the EP/3 flow in a circular pipeline controlled by a 150MHz clock. New instructions flow from the *Iy* (Instruction Highway) into the *Inst*, where they are decoded and issued onto the *My* (Memory Highway). All instructions consist of a command field which specifies the operation and operands, and a destination field which specifies the target instructions to which the result will be sent. An instruction is accompanied by a word of data which forms one of the operands. The other operand can specify a location in the main memory which is read from or written to.

From the *My* the instruction enters the *Memadd*, in which its effective address is calculated by adding the base operand and displacement. Then the instruction enters the *Memory* at next clock cycle. After 'write' or 'read' operations in memory, the instruction with the result will be sent to the *Sy* (Stack Highway).

The *Stack* receives the input data from the *Sy* at the beginning of each clock cycle. The interlock signal *Ilock* determines whether the output data on the *Py* is kept or the input data on the *Sy* is stored into the *Stack*.

The instruction from the *Py* enters the *Cache* and *Alu1* units at the same time. They compute different functions of the instruction concurrently. The *Cache* fetches the target instruction from the cache memory array according the destination address. And the target instruction will be sent to the *Inst* via the *Iy* at next clock cycle. At the same time *Alu1* executes part of an arithmetic or logical operation and sends the result to *Alu2* which computes the remainder of the arithmetic or logical operation.

Here we only focus on the interlock control mechanism. So certain components are ignored, such as *Alu1*, *Alu2*, *Memadd* and *Memory*. We also assume that the functional operations in each component are correctly implemented. We also ignore the cache loading mechanism, i.e., we assume that the complete instruction tree is already in the *Cache*.

We use a special symbol *bubble* to denote an empty pipeline-slot.

3.1 Component Interfaces and EP/3 Instruction Tree

For simplicity, we combine *Cache* and *Alu1* into one component *Stage0*, *Inst* and *Alu2* into component *Stage1* and *Memadd*, *Memory* and *Stack* into component *Stage2*. We will denote the data-flow bus from *Stage0* to *Stage1* again by *Iy* (Instruction Highway), the data-flow bus from *Stage1* to *Stage2* by *My*, and the data-flow bus from *Stage2* to *Stage0* by *Py*. The interlock signal *Ilock* is used to control the pipeline. When *Stage1* receives the data from the *Iy* at the beginning of each clock cycle, the

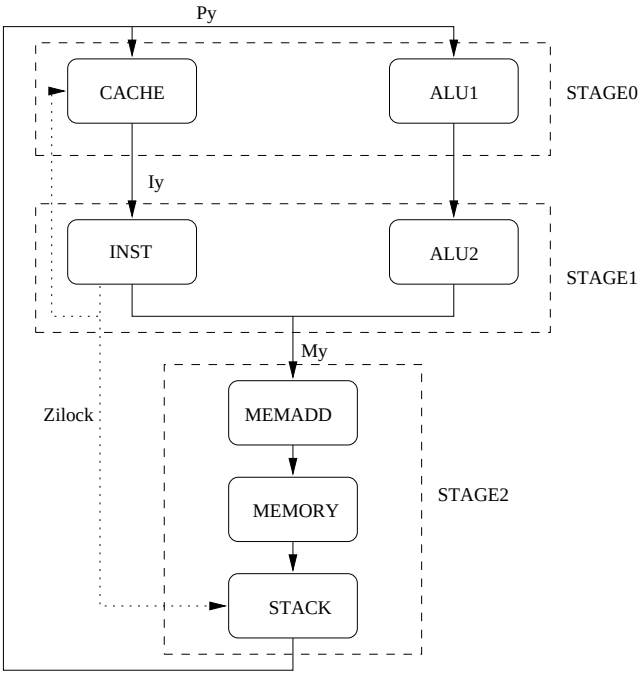

Figure 1 The EP/3 Architecture

Ilock signal will be set to 1 or 0 according to certain conditions. The *Ilock* will affect *Stage0* and *Stage2* immediately.

The input and output interface of each unit can be described as follows.

Stage0	$(in : Py, Ilock;$	$out : Iy)$
Stage1	$(in : Iy;$	$out : My, Ilock)$
Stage2	$(in : My, Ilock;$	$out : Py)$

We use an instruction tree for representing machine programs in EP/3. It is a binary tree where nodes represent the instructions, arcs represent the relations of father and son among the instructions, and leaves represent the finished instructions. The model gives the order relations among the instructions in the EP/3 program.

Figure 2 is an example. The root node of the instruction tree is instruction 0. It has two subtrees which represent two threads that start with respectively instructions 1 and 2. After instruction 0 is executed, the instructions 1 and 2 will be issued, one after the other, onto the *My*. EP/3 should execute instruction 0 before the executions of the instruction sons 1 and 2. The safety and liveness properties given in the next section will specify this order of execution. An instruction with no son will be considered terminated, for example instruction 7 is such an instruction.

Now we briefly describe the instruction structure of EP/3. An instruction consists of three parts; one is the operation part which gives the operation style such as ‘write’

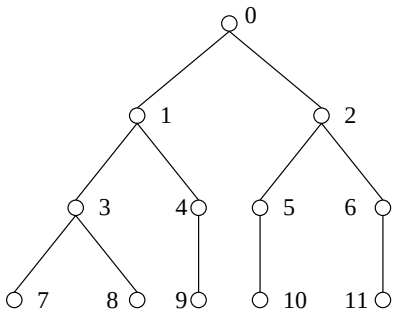

Figure 2 The Instruction Tree of EP/3.

or arithmetic operation in *Alu*, the second part is the operands of the instruction and the third is the destination addresses which are to be used to get the descending instructions. In other words, the instruction tree gives the address relations among the instructions. Here we assume that the instruction tree has only a finite number of nodes. We will use $i \prec j$ to denote that i is the ancestor of j.

4 SPECIFICATION OF THE EP/3

We will first show some simulation results (for which the Tempura code is given in the appendix1) and then proceed to give the formal specification of the EP/3.

4.1 Simulation in Tempura

In Fig 3, we present the result of executing the instruction tree of EP/3 given in Fig 2. The figure shows clearly the behavior of a stack, i.e., instruction 4 enters the stack in State 8 and leaves the stack in State 14 while instruction 6 enters the stack in State 10 and leaves the stack in State 13. The execution time for 12 instructions is 20 cycles.

If there is only a single thread in an instruction tree, the performance of the EP/3 is then at it worst, i.e., there is always at most one instruction in the pipeline. Obviously, the execution time is $3n$ cycles for a single thread of length n.

4.2 The formal specification

The specification of EP/3 is the composition of the specifications of three components, i.e.,

$$EP/3 \;\widehat{=}\; Stage0 \wedge Stage1 \wedge Stage2$$

```
State   0: Py=bubble  Iy=bubble  My=[0]     Ilock=0  I=0  L=[]
State   1: Py=[0]     Iy=bubble  My=bubble  Ilock=0  I=0  L=[]
State   2: Py=bubble  Iy=[1,2]   My=bubble  Ilock=1  I=0  L=[]
State   3: Py=bubble  Iy=[1,2]   My=[1]     Ilock=0  I=1  L=[]
State   4: Py=[1]     Iy=bubble  My=[2]     Ilock=0  I=0  L=[]
State   5: Py=[2]     Iy=[3,4]   My=bubble  Ilock=1  I=0  L=[]
State   6: Py=[2]     Iy=[3,4]   My=[3]     Ilock=0  I=1  L=[]
State   7: Py=[3]     Iy=[5,6]   My=[4]     Ilock=1  I=0  L=[]
State   8: Py=[3]     Iy=[5,6]   My=[5]     Ilock=0  I=1  L=[4]
State   9: Py=[5]     Iy=[7,8]   My=[6]     Ilock=1  I=0  L=[4]
State  10: Py=[5]     Iy=[7,8]   My=[7]     Ilock=0  I=1  L=[6,4]
State  11: Py=[7]     Iy=[10]    My=[8]     Ilock=0  I=0  L=[6,4]
State  12: Py=[8]     Iy=bubble  My=[10]    Ilock=0  I=0  L=[6,4]
State  13: Py=[10]    Iy=bubble  My=bubble  Ilock=0  I=0  L=[6,4]
State  14: Py=[6]     Iy=bubble  My=bubble  Ilock=0  I=0  L=[4]
State  15: Py=[4]     Iy=[11]    My=bubble  Ilock=0  I=0  L=[]
State  16: Py=bubble  Iy=[9]     My=[11]    Ilock=0  I=0  L=[]
State  17: Py=[11]    Iy=bubble  My=[9]     Ilock=0  I=0  L=[]
State  18: Py=[9]     Iy=bubble  My=bubble  Ilock=0  I=0  L=[]
State  19: Py=bubble  Iy=bubble  My=bubble  Ilock=0  I=0  L=[]
```

Figure 3 Simulation of executing a tree

Section 3.1 gave the input and output interfaces of each component. At the beginning of each clock cycle (present state), each component gets the input information from its input ports, then it will process the information and send the result to the output ports at the end of the clock cycle (next state), i.e., there is a unit time delay between input and output. The signal *Ilock* is an exception to this and is produced immediately after *Stage1* receives the input information. In fact, there is still a time delay between the input signals and *Ilock*, but we ignore it since it is very small relative to the clock cycle. In practical circuit design, the *Ilock* signal is produced by a combinational circuit and affects *Stage0* and *Stage2* immediately as shown in Fig 1. This is the key to the interlock mechanism in the EP/3 design. Now we will describe the specification of each component separately.

- For the *Stage0*, the input variables are *Ilock* and *Py*. The output variable is *Iy*. If $Ilock = 0$ (meaning the unit is unlocked), the value for *Iy* will be fetched from the cache memory according to the destination address of instruction *Py*. Simultaneously the result of instruction *Py* will be computed, we will omit how this is done. Here we use a function *sons* which can read the descending instruction of *Py* from the cache memory. If $Ilock = 1$, the unit is locked, then the output remains the stable. The formal ITL specification of *Stage0* can be described as follows:

$$Stage0 \mathrel{\widehat{=}} \Box(\text{if } Ilock = 0 \text{ then } Iy := sons(Py) \text{ else } Iy := Iy)$$

The definition of function *sons* is as follows

$$sons(Py) = \begin{cases} bubble & \text{if } Py \text{ is a leaf of instruction tree, or a } bubble. \\ j & \text{if } Py \text{ has one destination } j. \\ j_1 j_2 & \text{if } Py \text{ has two destinations } j_1 \text{ and } j_2. \end{cases}$$

- When the *Stage1* component receives the input instruction *Iy*, it will set or reset the interlock signal *Ilock* immediately. *Ilock* is set to 1 only when the *Stage0* receives a 2-destination instruction for the first time. Otherwise *Ilock* is set to 0. Here we use function *dest* which gives the number of destinations of *Iy*, and function *son* that will give the actual destination instructions.
  Function *dest* can be defined as follows.

$$dest(Iy) = \begin{cases} 0 & \text{if } Iy \text{ is a } bubble. \\ 1 & \text{if } Iy \text{ has one destination.} \\ 2 & \text{if } Iy \text{ has two destinations.} \end{cases}$$

  If $dest(Iy) = 1$ then $son(Iy)$ denotes the son instruction of instruction *Iy*. If $dest(Iy) = 2$ then $left(Iy)$ and $right(Iy)$ denote respectively the left and right son instruction of instruction *Iy*. The left son instruction is first sent on the *My* and the right son instruction is sent the next clock cycle.
  The formal ITL specification is as follows.

$$\begin{array}{l} Stage1 \mathrel{\widehat{=}} \\ \Box(\text{if } dest(Iy) = 0 \text{ then } Ilock = 0 \wedge My := bubble \wedge I := 0 \\ \quad \text{else if } dest(Iy) = 1 \text{ then } Ilock = 0 \wedge My := son(Iy) \wedge I := 0 \\ \quad\quad \text{else if } I = 0 \text{ then } Ilock = 1 \wedge My := left(Iy) \wedge I := 1 \\ \quad\quad\quad \text{else } Ilock = 0 \wedge My := right(Iy) \wedge I := 0 \\ ) \end{array}$$

- The *Stage2* component receives its input from *Stage1* and sends its output to *Stage0* via the *Py*. However, the instruction with two destinations will need two clock cycles to send two successive instructions onto the *My*. Therefore, *Stage2* cannot always send new instruction parcels onto *Py*. EP/3 uses the interlock signal *Ilock* to signal that *Stage2* should store the instruction from *My* at this time, and wait until some future time when *Py* is clear and a *bubble* is present on *My*. It will then pop a stored instruction which is the head of the list *L*.
  The formal ITL specification of *Stage2* is as follows, where $head(L)$ denotes the

first element of the list L, $rest(L)$ denotes the L without its first element, $\langle My\rangle^\frown L$ denotes that My is added at the front of L, and $\langle\rangle$ denotes the empty list.

$$
\begin{array}{l}
Stage2 \mathrel{\widehat{=}} \\
\Box(\ \text{if } Ilock = 1 \text{ then} \\
\qquad \text{if } My = bubble \text{ then } Py := Py \wedge L := L \\
\qquad \text{else } Py := Py \wedge L := \langle My\rangle^\frown L \\
\quad \text{else if } My \neq bubble \text{ then } Py := My \wedge L := L \\
\qquad \text{else if } L = \langle\rangle \text{ then } Py := bubble \wedge L := L \\
\qquad\qquad \text{else } Py := head(L) \wedge L := rest(L) \\
)
\end{array}
$$

5 PROPERTIES OF THE EP/3

The specification of the EP/3 should satisfy some requirements (properties), such as safety (no bad thing will happen) and liveness (a good thing will eventually happen). For example, if an instruction is pushed into the stack, then it should be popped out eventually. This is a liveness property of EP/3. If the specification of the EP/3 satisfies these properties, then we say the abstract specification is correct. Following are some safety and liveness properties. We assume that the instruction tree is finite.

Safety For any arbitrary instruction tree, EP/3 should execute the instructions conform the ancestor relation $\prec$.
The first instruction i_0 (the root node) is sent on the *My* at the beginning of execution of any instruction tree by an initial procedure of a component *I/O Processor* of EP/3 omitted in this paper.

$$Initial \mathrel{\widehat{=}} My = i_0 \wedge Py = bubble \wedge Iy = bubble \wedge\ L = \langle\rangle$$

The safety property of the EP/3 is the conjunction of following properties in which i and j denote any non-*bubble* instructions in the instruction tree.

1. Any instruction is descended and can not be lost.
   If an instruction i is sent onto the *My*, then it will pass through each component of the EP/3, i.e., it cannot be lost before it is finished. This safety property can be specified as the conjunction of following properties

$$
\begin{array}{lcl}
Sp0_a & \mathrel{\widehat{=}} & \Box(Iy := sons(Py) \vee \bigcirc Iy := sons(Py)) \\
Sp1_a & \mathrel{\widehat{=}} & \Box(My := son(Iy) \vee My := left(Iy) \vee My := right(Iy)) \\
Sp2_a & \mathrel{\widehat{=}} & \Box(My = bubble \vee Py := My \vee head(L) := My)
\end{array}
$$

2. No instruction is repeatedly executed.

Any instruction in the tree cannot appear more than once on the highways of the EP/3, i.e., *non-duplication*.

$$\begin{array}{lcl} Sp0_b & \widehat{=} & \forall i \bullet \Box(\neg(Py = i \wedge (len \geq 2 ; Py = i))) \\ Sp1_b & \widehat{=} & \forall i \bullet \Box(\neg(Iy = i \wedge (len \geq 2 ; Iy = i))) \\ Sp2_b & \widehat{=} & \forall i \bullet \Box(\neg(My = i \wedge (len \geq 1 ; My = i))) \end{array}$$

3. Execution order is consistent with $\prec$.
   An instruction can not appear on the high ways before its ancestors.

$$Sp \quad \widehat{=} \quad \forall i, j, i \prec j \bullet \Box(\neg(hwy = j \wedge (len \geq 1 ; hwy = i)))$$

where $hwy \in \{Py, Iy, My\}$.

Termination The termination property expresses that the program must terminate for any instruction tree whose number of nodes is finite. The final state of the EP/3 can be described as

$$Final \quad \widehat{=} \quad Py = bubble \wedge Iy = bubble \wedge My = bubble \wedge L = \langle\rangle$$

Suppose n is the number of nodes in the instruction tree. If there is at most one instruction in the pipeline then it is clear that the pipeline is not efficient. This could happen if the tree has no 2-destination instructions. The execution time of such a tree is $3n$. So we have the termination property

$$T \quad \widehat{=} \quad Initial \wedge len \geq 3n \supset \Diamond\Box Final$$

Liveness The following liveness properties will be considered:

1. *Stage2* must guarantee that if an instruction in the instruction tree is pushed onto the L, then it will be popped out of the L eventually.

$$L_1 \quad \widehat{=} \quad \forall i \bullet \ len \geq 3n \wedge \Box(i \in L \supset \Diamond(Py = i))$$

2. Furthermore every instruction in the instruction tree should be executed before the time bound of $3n$.

$$L_2 \quad \widehat{=} \quad \forall i \bullet Initial \wedge len = 3n \supset \Diamond Py = i$$

6 VERIFICATION OF THE EP/3

The specification of EP/3 should satisfy (imply) the requirements (properties). In this section, we will give the verifications of those properties. Here we only give the proof guidelines and omit the details.

Some lemmas are used to make proofs more understandable. They are easily derived from the specifications of each EP/3 component, initial assumption and some definitions.

- Proof of the Safety Property. We have to prove the following:

  $$\begin{array}{l} \mathit{Initial} \wedge \mathit{Stage0} \wedge \mathit{Stage1} \wedge \mathit{Stage2} \quad \supset \\ \mathit{Sp0}_a \wedge \mathit{Sp1}_a \wedge \mathit{Sp2}_a \wedge \mathit{Sp0}_b \wedge \mathit{Sp1}_b \wedge \mathit{Sp2}_b \wedge \mathit{Sp} \end{array}$$

  Since $\mathit{Sp0}_b \wedge \mathit{Sp1}_b \wedge \mathit{Sp2}_b \supset \mathit{Sp}$ and $\mathit{Sp0}_b \equiv \mathit{Sp1}_b \equiv \mathit{Sp2}_b$ the following will do.

  $$\mathit{Initial} \wedge \mathit{Stage0} \wedge \mathit{Stage1} \wedge \mathit{Stage2} \quad \supset \quad \mathit{Sp0}_a \wedge \mathit{Sp1}_a \wedge \mathit{Sp2}_a \wedge \mathit{Sp0}_b$$

  The following compositional rule

  $$\frac{\vdash f_0 \supset f_2 \qquad \vdash f_1 \supset f_3}{\vdash f_0 \wedge f_1 \supset f_2 \wedge f_3}$$

  enables us to split this into

  $$\begin{array}{lcl} \mathit{Initial} \wedge \mathit{Stage0} \wedge \mathit{Stage1} \wedge \mathit{Stage2} & \supset & \mathit{Sp0}_a \wedge \mathit{Sp0}_b \\ \mathit{Initial} \wedge \mathit{Stage1} & \supset & \mathit{Sp1}_a \\ \mathit{Initial} \wedge \mathit{Stage2} & \supset & \mathit{Sp2}_a \end{array}$$

  And these can be proven very easily.
- The proof of the Termination property requires the introduction of the following lemmas. From the specification of the *Stage1*, we can easily get

  Lemma 1 $\Box(\mathit{Iy} = \mathit{bubble} \vee \bigcirc \mathit{Iy} = \mathit{bubble} \supset \mathit{Ilock} = 0)$

  i.e., if $\mathit{Ilock} = 1$ then the current *Iy* has two destinations, so *Iy* will not be *bubble* both this state and the next state.
  The following lemma is used for the proof that once the processor is in the final state it will remain in the final state.

  Lemma 2 $\mathit{Final} \supset \bigcirc \mathit{Final}$

Proof

(1)	$Iy = bubble \supset Ilock = 0$	Lemma 1
(2)	$Py = bubble \wedge Iy = bubble \supset \bigcirc Iy = bubble$	$1, Stage0$
(3)	$\bigcirc Iy = bubble \supset Ilock = 0$	Lemma 1
(4)	$Iy = bubble \wedge Py = bubble \wedge My = bubble \supset$ $\bigcirc My = bubble$	$2,3, Stage1$
(5)	$Iy = bubble \wedge Py = bubble \wedge My = bubble \wedge L = \langle\rangle \supset$ $\bigcirc Py = bubble \wedge \bigcirc L = \langle\rangle$	$1, Stage2$
(6)	$Final \supset \bigcirc(Final)$	$2,4,5$

With the following induction proof rule of ITL

$$\frac{\vdash f \supset \bigcirc f}{\vdash f \supset \Box f}$$

we get the desired result

Lemma 3 $Final \supset \Box Final$

It is Lemma 3 that will actually be used in the proof of the termination property below.
We can prove the termination property by introducing a variable C for counting the issued instructions from *Stage0*.
We assume that at the initial state $C = 1$ and thereafter C will increase until n.

$$AS1 \quad \widehat{=} \quad (Initial \supset C = 1) \wedge \Box((\bigcirc Iy \neq bubble \supset C := C + 1) \wedge (\bigcirc Iy = bubble \supset C := C))$$

Since the number of the instructions in the tree is n, C will not increase when it reaches n. So we have the following assumption

$$AS2 \quad \widehat{=} \quad \Box(C = n \supset \bigcirc(Iy = bubble \wedge My = bubble \wedge C = n))$$

That is $Iy \neq bubble \wedge C = n \supset \bigcirc Final$
The other instructions must be *bubble* and L must be empty after the last instruction ($Iy \neq bubble \wedge C = n$). Otherwise after a few cycles Iy will not be *bubble* and

then C will become $n+1$. That is a contradiction with the assumption. If $n=1$ then the termination property holds else we have

(1)	$Initial \wedge C=1 \wedge len=2 \supset fin(C=2)$	$AS1$
(2)	$C=2 \wedge len=3 \supset fin(C \geq 3)$	$1, AS1$
(3)	$C=k \wedge k<n \wedge len=3 \supset fin(C \geq k+1)$	$2, AS1$
(4)	$Initial \wedge C=1 \wedge len=3n-1 \supset fin(C=n)$	$1,2,3, \mathcal{CR}$
(5)	$Initial \wedge C=1 \wedge len=3n \supset fin(Final)$	$4, AS2$
(6)	$Initial \wedge len \geq 3n \supset \Diamond\Box Final$	5, Lemma 3

- Proof of the liveness properties. Obviously we can prove the liveness properties L_1 and L_2 from the termination property, i.e., $T \supset L_1 \wedge L_2$.

7 DISCUSSION AND CONCLUSION

In this paper, we have specified the EP/3 at a high level of abstraction and have also given a compositional proof of correctness for its interlock mechanism. (Cau *et al*. 1996) has given a low level (at register transfer level) (executable) specification of the EP/3. (Cau and Zedan 1997) has extended ITL with refinement rules which will be used to refine the abstract specification to this low level specification. In fact, we are planning to prove the logical equivalence of a variant of the pipeline specification given here and a much more distributed, algorithmic description of the EP/3 in which each instruction is modeled as a separate process which is responsible for getting itself through the pipeline. A version of this second description has already been successfully simulated using Tempura. So the whole development process from high level specification to low level executable code can be expressed in ITL.

In the proof of the termination and liveness properties in section 6 we made the assumption that the number of instructions is finite. This assumption can be dropped if we use a queue instead of a stack in *Stage2*. Furthermore *Stage2* should be changed into (changes are underlined)

$$
\begin{array}{l}
Stage2 \mathrel{\widehat{=}} \\
\Box(\ \textsf{if } Ilock = 1 \textsf{ then} \\
\qquad \textsf{if } My = bubble \textsf{ then } Py := Py \wedge L := L \\
\qquad \textsf{else } Py := Py \wedge \underline{L := L^{\frown}\langle My \rangle} \\
\quad \textsf{else if } My \neq bubble \textsf{ then} \\
\qquad\quad \underline{\textsf{if } L = \langle\rangle \textsf{ then } Py := My \wedge L := L} \\
\qquad\quad \underline{\textsf{else } Py := head(L) \wedge L := L^{\frown}\langle My \rangle} \\
\qquad \textsf{else if } L = \langle\rangle \textsf{ then } Py := bubble \wedge L := L \\
\qquad\quad \textsf{else } Py := head(L) \wedge L := rest(L) \\
)
\end{array}
$$

Figure 4 shows a sample execution of the same instruction tree as used in Fig. 3.

```
State   0: Py=bubble  Iy=bubble  My=[0]     Ilock=0  I=0  L=[]
State   1: Py=[0]     Iy=bubble  My=bubble  Ilock=0  I=0  L=[]
State   2: Py=bubble  Iy=[1,2]   My=bubble  Ilock=1  I=0  L=[]
State   3: Py=bubble  Iy=[1,2]   My=[1]     Ilock=0  I=1  L=[]
State   4: Py=[1]     Iy=bubble  My=[2]     Ilock=0  I=0  L=[]
State   5: Py=[2]     Iy=[3,4]   My=bubble  Ilock=1  I=0  L=[]
State   6: Py=[2]     Iy=[3,4]   My=[3]     Ilock=0  I=1  L=[]
State   7: Py=[3]     Iy=[5,6]   My=[4]     Ilock=1  I=0  L=[]
State   8: Py=[3]     Iy=[5,6]   My=[5]     Ilock=0  I=1  L=[4]
State   9: Py=[4]     Iy=[7,8]   My=[6]     Ilock=1  I=0  L=[5]
State  10: Py=[4]     Iy=[7,8]   My=[7]     Ilock=0  I=1  L=[6,5]
State  11: Py=[5]     Iy=[9]     My=[8]     Ilock=0  I=0  L=[7,6]
State  12: Py=[6]     Iy=[10]    My=[9]     Ilock=0  I=0  L=[8,7]
State  13: Py=[7]     Iy=[11]    My=[10]    Ilock=0  I=0  L=[9,8]
State  14: Py=[8]     Iy=bubble  My=[11]    Ilock=0  I=0  L=[10,9]
State  15: Py=[9]     Iy=bubble  My=bubble  Ilock=0  I=0  L=[11,10]
State  16: Py=[10]    Iy=bubble  My=bubble  Ilock=0  I=0  L=[11]
State  17: Py=[11]    Iy=bubble  My=bubble  Ilock=0  I=0  L=[]
State  18: Py=bubble  Iy=bubble  My=bubble  Ilock=0  I=0  L=[]
```

Figure 4 Simulation of modified EP/3

At state 9 instruction 5 doesn't bypass the queue but enters the queue and simultaneously instruction 4 leaves the queue. This improvement ensures that, in case of an infinite tree, an instruction leaves the queue eventually, i.e., isn't overtaken by any other instruction. One can see that "the execution order" is preserved.

The termination and liveness properties should then be reformulated as follows:

$$T \;\widehat{=}\; Initial \wedge inf \supset \Diamond\Box Final$$
$$L \;\widehat{=}\; \forall i, j.i \prec j \bullet \Box(Py = i \wedge inf \supset \Diamond Py = j)$$

where i and j are not bubbles.

Following the methodology of this paper, it is easy to extend the verification of EP/3 with an *IO processor* component. The latter is responsible for cache memory loading and external communications.

In (Cau and Moszkowski 1996) we have embedded the ITL proof system within the Prototype Verification System (PVS). Some of the proofs generated in the paper are mechanically checked, see appendix2 for the ITL specification of the EP/3 encoded in PVS using the ITL library. Part of the refinement calculus of (Cau and Zedan 1997) has also been incorporated into PVS, so that refinement can also be mechanically checked (Cau *et al*. 1997). Furthermore a link between PVS and the Tempura simulator will be built which allows executable ITL specifications derived with PVS to be executed. So a general development tool is constructed in which you can verify, refine and execute ITL specifications.

The interlock mechanism of EP/3 uses both asynchronous and synchronous signals to control the components. This demonstrates that ITL is suitable for describing both synchronous circuits and asynchronous circuits. In (Cau and Zedan 1997) explicit constructs for both synchronous and asynchronous communication have been defined.

ACKNOWLEDGMENTS

We would like to thank the referees for their helpful comments. This work is part of the EPSRC funded project GR/K25922 “A compositional approach to the specification of systems using ITL and Tempura”.

REFERENCES

Cau, A. , Zedan, H. , Coleman, N. and Moszkowski, B. (1996) Using ITL and Tempura for Large Scale Specification and Simulation. In *The Fourth Euromicro Workshop on Parallel and Distributed Processing*. Braga, Portugal, 1996.

Cau, A. and Moszkowski, B. (1996) Using PVS for Interval Temporal Logic Proofs, Part 1: The Syntactic and Semantic Encoding. Technical Report, 1996.

Cau, A. , Moszkowski, B. and Zedan, H. (1997) Interval Temporal Logic Proof Checker Manual. In preparation.

Cau, A. and Zedan, H. (1997) Refining Interval Temporal Logic Specifications. In the proceedings of the Fourth AMAST Workshop on Real-Time Systems, Concurrent, and Distributed Software, LNCS 1231, Mallorca, Spain, May 21-23 1997.

Cohn, A. (1988) A Proof of the Viper Microprocessor: The First Level; In *VLSI Specification, Verification and Synthesis* , G. Birtwistle and P. Subrahmanyam (eds.), Kluwer Academic Publishers, 1988.

Coleman, N. (1993) Simulation of EP/3 in Pascal. Technical Report, University of Newcastle upon Tyne, 1993.

Coleman, N. (1993) A High Speed Data-flow Processing Element and Its Performance Compared to a Von Neumann Mainframe. In *IEEE 7th Int’l. Parallel Processing Symp*, pp24-33, Newport Beach, California, 1993.

Li, X. and Coleman, N. (1996) Simulation of EP/3 in Verilog HDL. *draft*, University of Newcastle upon Tyne, 1996.

Moszkowski, B. (1985) A Temporal Logic for Multilevel Reasoning About Hardware. IEEE Computer 1985;18:10-19.

Moszkowski, B. (1986) Executing Temporal Logic Programs. Cambridge Univ. Press, Cambridge, UK, 1986.

Moszkowski, B. (1994) Some Very Compositional Temporal Properties. In *E.R. Olderog(ed) Programming Concepts, Methods and Calculi*. IFIP Transaction, Vol. A-56, North-Holland, pp307-326, 1994.

Moszkowski, B. (1995) Compositional Reasoning About Projected and Infinite Time. In *Proceedings of the First IEEE International Conference on Engineering of Complex Computer Systems(ICECCS’95)*. IEEE Computer Society Press. Los Alamitos, California, pp238-245, 1995.

Moszkowski, B. (1996) Using temporal fixpoints to compositionally reason about liveness, in proc. of the 7th BCS FACS Refinement Workshop, He Jifeng (ed.), Bath, UK, 1996.

Rushby, J. (1993) A tutorial on Specification and Verification using PVS, in proc. of the First Intl. Symp. of Formal Methods Europe FME'93: Industrial-Strength Formal Methods, Peter Gorm Larsen (ed.), 1993, Odense, Denmark, 357–406.
Check home-page: `http://www.csl.sri.com/pvs.html`

Tahar, S. and Kumar, R. (1995) Formal Specification and Verification Techniques for RISC Pipeline Conflicts; In *Computer Journal*, Vol. 38, No. 2, 1995.

Windley, P. (1995) Formal Modeling and Verification of Microprocessor; In *IEEE Transactions on Computers*, Vol. 44, No. 1, 1995.

APPENDIX 1 TEMPURA CODE

The following is a listing of the Tempura code. The code is based on the specifications *Stage0*, *Stage1* and *Stage2*.

```
/* initial cache memory and [] denotes bubble */
define cachem=[[1,2],[3,4],[5,6],[7,8],[9],[10],[11],[],[],[],[],[]].

define sons(N)={if N=[] then [] else cachem[N[0]] }.
define dest(N)={if N=[] then 0 else |N| }.
define son(N)={ N }.
define left(N)={ [N[0]] }.
define right(N)={ [N[1]] }.

define stage0(Py,Ilock,Iy,Stop)={
  always(if Stop then {Ilock=0 and empty}
         else if Ilock=1 then Iy:=Iy  else Iy:= sons(Py) )}.

define stage1(Iy,Ilock,I,My,Stop)={
  always(if Stop then empty else
  if dest(Iy)=0      then {Ilock = 0 and My:= [] and I:=0 }
  else if dest(Iy)=1 then {Ilock=0 and My:=son(Iy) and I:=0}
  else if I = 0  then {Ilock=0 and My:=left(Iy) and I:=1}
       else { Ilock = 1 and My:=right(Iy) and I:=0 })}.

define stage2(My,Ilock,L,Py,Stop) ={always( if Stop then empty else
  if Ilock=0 then if My=[] then if L=[] then {Py:=[] and L:=L}
  else { Py:=[L[0]] and L:= L[1..|L|]} else {Py:=My and L:=L}
  else if My=[] then {Py:=Py and L:=L} else {Py:=Py and L:=My + L})}.

define ep3() = { exists L,Py,Iy,My,Ilock,I,Stop: {
  L=[] and I=0 and My=[0] and Py=[] and Iy=[] and
  always format("Py=%5t Iy=%7t My=%5t Ilock=%2t I=%2t L=%t\n",
                Py,Iy,My,Ilock,I,L) and
  always (Stop={Py=[] and Iy=[] and My=[] and L=[]}) and
  and stage0(Py,Ilock,Iy,Stop)
  and stage1(Iy,Ilock,I,My,Stop)
  and stage2(My,Ilock,L,Py,Stop) }}.
```

APPENDIX 2 PVS SPECIFICATION

The following is part of the ITL specification of the EP/3 encoded in PVS. It imports the ITL library discussed in (Cau and Moszkowski 1996, Cau *et al*. 1997). All the proofs presented here have been checked. Due to space limitations we will give no further details but refer to (Cau and Moszkowski 1996, Cau *et al*. 1997) for more details. The ITL library can be obtained from
`http://www.cms.dmu.ac.uk/~cau/project.html`.

```
interlock  : THEORY
  BEGIN
itl: LIBRARY ‘‘/export/home0/users/acau/ctempura/Pvs/itl’’
importing itl@itl1
...
initial : form =  py=bubble /\ eqa(iy,bubble2) /\ my=root /\ Llen=zero

final   : form =  py=bubble /\  eqa(iy,bubble2) /\ my=bubble /\ Llen=zero

pipe0inner : form = ife( ilock=zero, ast(iy,sons(py)), ast(iy,iy) )
pipe0    : form = inf /\ []( pipe0inner )

pipe1inner : form =
  ife(dest(iy)=zero,
          ilock=zero /\ as(my,bubble) /\ as(i,zero),
          ife(dest(iy)=one,
              ilock=zero /\ as(my,son(iy)) /\ as(i,zero),
              ife(i=zero,
                  ilock=one /\ as(my,left(iy)) /\ as(i,one),
                  ilock=zero /\ as(my,right(iy)) /\ as(i,zero)
                 )
              )
          )
pipe1    : form = inf /\ []( pipe1inner )

pipe2inner : form =
  ife(ilock=zero,
          ife(my=bubble,
              ife(Llen=zero,
                  as(py,bubble) /\ as(Llen,Llen) /\ asl(L,L),
                  as(py,head(L)) /\ as(Llen,Llen-one) /\ rest(L)
                 ),
              as(py,my) /\ as(Llen,Llen) /\ asl(L,L)
             ),
          ife(my=bubble,
              as(py,py) /\ as(Llen,Llen) /\ asl(L,L),
              as(py,py) /\ as(Llen,Llen+one) /\ concat(my,L)
             )
         )
pipe2    : form = inf /\ []( pipe2inner )

sp0a : form = inf /\ []( ast(iy, sons(py)) \/ ast2(iy, sons(py)) )
sp1a : form = inf /\ []( as(my,son(iy)) \/ as(my,left(iy)) \/ as(my,right(iy)) )
sp2a : form = inf /\ []( my=bubble \/ as(py,my) \/ as(head(L),my) )
sp0b : form = inf /\ FA(i2, zero<=i2 => [](-(py=i2 /\ (skip^skip^finite^(py=i2)))))
sp2b : form = inf /\  FA(i5, zero<=i5 => [](-(my=i5 /\ (skip^finite^(my=i5)))))

saf_1   : LEMMA  (pipe1 /\ initial) => sp1a

saf_2   : LEMMA  (pipe0 /\ pipe1 /\ pipe2 /\ initial) => (sp0a /\ sp0b)

saf_3   : LEMMA  (pipe2 /\ initial) => sp2a
...
  END interlock
```