An Object-Oriented Framework for the Formal Verification of Processors

Laurent Arditi and Hélère Collavizza

Université de Nice – Sophia Antipolis
Laboratoire I3S, CNRS-URA 1376
650, Route des Colles. B.P. 145
06903 Sophia Antipolis Cédex. France
email: arditi@unice.fr, helen@essi.fr

Abstract. We propose an object-oriented approach for the formal verification of processors. This approach has been validated on significant applications. It is based on a class hierarchy that provides the basic components to describe processors at any abstraction level, and to specify verifications to execute.
The originality of our method is to combine an object-oriented model (to ensure *genericity*) and a computer algebra verification system (to ensure *efficiency*). Computer experiments with our framework clearly shown three main advantages: processor descriptions are very easy to write down, the core of the verification system is generic so it may be applied without any modification to different processors, and last, the proof times are very short compared with previous approaches.

1 Introduction

In this section, we first provide general motivations for hardware verification, and then outline our approach in general terms.

1.1 Motivations

In order to produce correct circuits, several verification tools such as correct generators of floor-plans or logic simulators are generally applied during the design process. However, it is widely accepted that the existing CAD tools have two major drawbacks:

- they cannot perform exhaustive proofs: this is due to the combinatorial complexity of the problem,
- they are low-level oriented and therefore they cannot deal with abstract specifications.

For these reasons, attention has been paid these last ten years on general formal verification methods (for a survey and further justifications see [10]). In general, formally verifying a circuit consists in proving that its specification is logically equivalent to its implementation, in the sense of a logical equivalence. More explicitly, the verification process is decomposed in three steps:

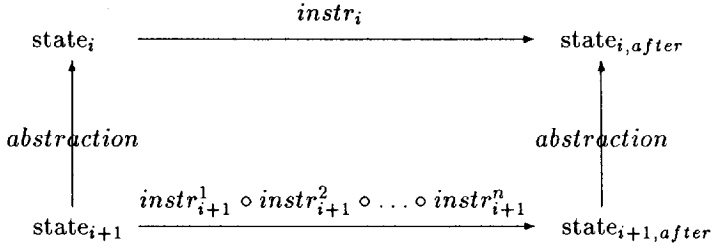
- in the first step, a formal model of the specification of the circuit is defined,
- in the second step, a formal model of the implementation of the circuit is defined, and
- in the third step, a proof that for any input and any state variable values, the two models are equal is performed.

This last point amounts to the proof of a universally quantified formula and can be realized using theorem proving techniques such as simplification or inductive reasoning.

1.2 The Problem of Processor Verification

The previous process is crucial when dealing with complex circuits such as processors. The key to the formal verification is the decomposition in successive proofs at successive abstraction levels: a more and more detailed view of the same processor is described [1]. The *specification* of the circuit is then a description of an abstract level (such as the assembly language level), while its *implementation* is a description at a more concrete level (for instance, the microinstruction level or the electronic block model). For each level, one exactly keeps the same description and verification process. The processor is specified as a set of components on which a set of actions is performed.

Let $level_i$ be an abstract specification level, and $level_{i+1}$ a more concrete one. The set of objects at $level_i$ is included in the set of objects at $level_{i+1}$, and an action at $level_i$ is realized at $level_{i+1}$ by composing a set of actions at $level_{i+1}$. So to verify the correctness between two adjacent levels one must show that the following diagram commutes:



For instance, if $level_i$ is the processor as seen by the assembly programmer, and $level_{i+1}$ is the architecture of the processor, the commutation property of the above diagram means that an assembly instruction is correctly realized by a sequence of microinstructions. Thus, to implement this abstract schema one needs:

- languages to describe states, transitions and abstractions, and
- well-suited proof systems to carry out the transitions and verify the commutation of the diagram.

1.3 Aims of the Paper and Related Works

Several interesting approaches have been proposed to realize formal verification of processors along the lines of the previous general schema, see for instance [7, 11, 18, 21, 22]. However, in our sense, these works display two kinds of shortcomings:

- they are specific to a particular processor. This means that the specification of a new processor must be built from the scratch. As a consequence, they do not provide any user-friendly interface to describe processors and verifications to execute, and
- they are based on very general logical proof tools (such as HOL or Nqthm). However these tools are too complex to support efficiently particular problems.

To overcome the above drawbacks, a new framework for processor verification is presented here. The key idea is to combine a "computer algebra system" and an "object-oriented model" in a single framework. The computer algebra system is used to efficiently simplify expressions. The object-oriented model provides three major advantages:

- the description and verification process is generic, it is the same for all processors and for all specification levels, and
- the object-oriented model is well-suited to the concrete structure of microprocessors. As consequence, the specifications are concise and easy to write down, and
- the object classification and the polymorphism of the methods simplify the core of the verification system.

Computer experiments with our framework have shown that not trivial examples can be treated, in a very efficient way (see the comparative Table 2 at the end of the paper), and with a small specification effort.

Layout of the paper: in section 2, we introduce our method on a simple example. In section 3, we present the basis of our verification framework and emphasize the advantages of our object-oriented approach and of the computer algebra system. In section 4 we deal with the implementation of our framework and in section 5 with the results we have obtained so far.

2 Overview of our Method: an Example

Before discussing the technical details of our method, let us introduce it on a simple example.

Assume we are given a simple processor with a memory and one accumulator register and suppose we wish to prove the "addition" instruction in direct memory addressing mode. Let describe this processor at the assembly language level ($level_1$) and at the microinstruction level ($level_2$) as follows:

At the assembly language level, the state of the processor consists of the memory "mem", the accumulator register "acc" and the program counter "pc". The addition instruction in direct memory addressing mode involves the two simultaneous transitions:

ADD_{DIR}:	acc := acc + mem [mem [pc](3 to 7)]	acc *is updated*
	pc := pc + 1	pc *is incremented*

"mem[pc]" contains the code of the addition instruction and, "mem[pc](3 to 7)" extracts the 5 highest bits of "mem[pc]" that contain the operand address.

In our object-oriented framework, this will be described as an instance named "proc-level$_1$" of the class **Interpreter** with attributes **state**, **transitions** and **select** (see Sect. 3.1):

- **state** contains the tuple <mem, pc, acc> where each component is built up from an object-oriented library: "mem" is an instance of the **Memory** class while "pc" and "acc" are instances of the **Register** class.
- **transitions** describes the semantics of each instruction. Here, this attribute includes the set of parallel transitions associated with ADD_{DIR} key.
- **select** selects the current instruction that is in memory at address "pc" (here it selects the addition instruction).

At the microinstruction level, a more concrete state is considered. It includes the assembly language level state plus the instruction register "ir", the operand register "rop" and the current microinstruction pointer "mpc". The addition instruction is executed by a sequence of microinstructions M_0, M_1 and M_2, that involves the following transitions:

M_0 :	ir := mem[pc]	ir *receives the instruction code*
	pc := pc + 1	pc *is incremented*
	mpc := 1	M_1 *is the next microinstruction to execute*
M_1 :	rop := ir(3 to 7)	rop *receives the operand address*
	mpc := 2	M_2 *is the next microinstruction to execute*
M_2 :	acc := acc + mem[rop]	acc *is updated*
	mpc := 0	M_0 *is the next microinstruction to execute*

In our framework, the state and transitions at the microinstruction level will be also described as an instance "proc-level$_2$" of class **Interpreter** as follows:

- **state** contains the tuple <mem, pc, acc, ir, rop, mpc>,
- **transitions** includes the definition of the semantics of M_0, M_1, M_2 as described below,

- **select** selects the microinstruction that is in the microprogram memory at address "mpc".

We will see in Sect. 3.1 that the use of same structures for different levels induces more genericity in the proof process.

In order to perform the proof, we must show that the sequence M_0, M_1, M_2 correctly realizes the transitions associated with ADD_{DIR}. The proof is also specified by an instance of the class **Proof** that has four attributes (see Sect. 3.1):

- **specification** points the object "proc-level$_1$",
- **implementation** points the object "proc-level$_2$",
- **abstraction** defines the state abstraction, that realizes here the hiding of <mem, pc, acc, ir, rop, mpc> into <mem, pc, acc>, and
- **sync** defines the temporal abstraction. Since we assume that the first microinstruction of a microinstruction sequence is always at the address 0 (which is the "fetch" sequence), the predicate **sync** is here "mpc = 0".

To perform this proof, we first take an initial formal value **state** for proc-level$_2$. We then execute the microinstructions selected by **select** of proc-level$_2$ (i.e the microinstruction pointed by "mpc" in the microprogram memory) until **sync** is true. This gives the level$_2$ final state. We also abstract the initial state to obtain the level$_1$ initial state using the **abstraction** attribute. We then execute the instruction selected by **select** of proc-level$_1$ (i.e the instruction pointed by "pc"). This gives the level$_1$ final state. Finally, we verify that level$_1$ final state is an **abstraction** of level$_2$ final state.

In this simple case, the proof does not require any simplification and the state values obtained at the two levels are exactly the same. However, for more complex processors, where addressing mechanisms are much more elaborated, this can involve reducing expressions on bit vectors.

The above specification and proof process will be the same for any processor at any abstraction level.

3 The Framework Basis

In this section, we give further justifications for the object-oriented approach and the use of a computer algebra system.

3.1 Why an Object-Oriented Approach?

The first question that naturally arises is why we choose an object-oriented approach. There are three reasons for that: first, the need for a well-suited model, second, the need for typed processor components and third, the need for genericity. We will illustrate through the next sections of this paper, that the main advantages of object-oriented programming [13, 16] answer to these requirements.

A Well-Suited Model for Processor Description. When teaching computer structure, we usually introduce a processor as a set of components (the data path) that interact according to the signals sent by the sequencer (the control part).

This naturally leads to describe a processor as an object with three attributes:

- `state`: the set of all components visible at the current level,
- `transitions`: a set of state transitions that defines the behaviour of the processor. Each transition is a set of assignments labeled by an instruction key.
- `select`: a function from the processor state that returns the key of the current transition selected by the control part.

The attribute `state` is a set of components that are built out from an object-oriented component library. Thus, its construction takes full benefit of multiple inheritance (see Sect. 4.1).

Processor Components are Strongly Typed Objects. All circuits are built out from the same basic components namely, gates, registers, buses, memories... Each component is characterized by two aspects: its data type and its temporal behaviour and has at least two associated actions: *read* and *write*. Furthermore, some components are specializations of some others. For example, a wire is an instance of a bus (bus of length one), a ROM is an instance of a memory (for which the write action is forbidden). This induces a natural classification of the components that is easily realized with an object-oriented implementation. We will also see that the polymorphism of read and write actions simplifies the core of our verification system.

Specification and Verification Process is Generic. One important point in formally verifying processors, is that the specification and verification process must be reusable for all the processors of a same class. In fact, we need two kinds of genericity:

- *horizontal* genericity: the model should be reusable to specify different processors, and
- *vertical* genericity: at each abstraction level, the same generic model and proof method should be reused.

This leads us to define any processor at any level as an object of the class `Interpreter` that has the three previous attributes `state`, `transitions` and `select`. This ensures the genericity of the description.

In order to ensure the genericity of the proof, a verification between two specification levels is an object of the class `Proof`. This class has two attributes pointing the two interpreters and two others describing the abstraction functions *from the implementation to the specification*:

- **abstraction**: it defines the state abstraction. It is a function from the implementation to the specification state. It is often a state hiding since the specification state is a subset of the implementation state.
- **sync** defines the temporal abstraction. It is a predicate which value is *true* only when the two levels are synchronized which means that the implementation state corresponds with the start of a transition at the abstract level. This predicate is usually defined as a test on the implementation program counter.

To perform the proof we have to execute a transition at $level_i$ and a sequence of transitions at $level_{i+1}$. The implementation is correct if $level_i$ final state is an abstraction of $level_{i+1}$ final state when **sync** is true.

3.2 Why a Computer Algebra Simplification System?

In order to simplify the expressions involved in state transitions, we use a computer algebra system. This system is less powerful than general provers such as HOL or the Boyer & Moore theorem prover. However, it is enough powerful for processor verification, and furthermore, it is much more efficient and easy to use. It is especially well adapted for this specific problem where expressions to be proved are simple but numerous, and for which efficiency and simplicity must surpass power.

Some other proof systems use rewriting techniques for symbolic expressions (see for example [19, 17]). We choose the computer algebra point of view because it is much more efficient and can easily deal with commutative operators. The simplification of an expression like $(op\ e_1\ \ldots\ e_N)$ is driven by the head operator *op*, so the reduction strategy associated to *op* is used. The underlying algebra includes data types such as naturals, bit vectors, Booleans,...

For example, here are the steps reducing the expression

$$\texttt{bv-nat}(\texttt{bv-concat}(V_1, \texttt{bv-concat}(V_2, V_3)))$$

which is the conversion from bit vectors to naturals of the concatenation of three bit vectors. First, using associativity of concatenation we get

$$\texttt{bv-nat}(\texttt{bv-concat}(V_1, V_2, V_3))$$

By applying a specific reduction rule for conversion we get

$$(\texttt{bv-nat}(V_1) * 2^{\texttt{length}(V_2)} + \texttt{bv-nat}(V_2)) * 2^{\texttt{length}(V_3)} + \texttt{bv-nat}(V_3)$$

Finally by using distributivity of multiplication, we obtain

$$\texttt{bv-nat}(V_1) * 2^{\texttt{length}(V_2)+\texttt{length}(V_3)} + \texttt{bv-nat}(V_2) * 2^{\texttt{length}(V_3)} + \texttt{bv-nat}(V_3)$$

4 The Framework Implementation

Having answered the reasons underlying our system, we are now ready to exhibit its components. They are related to two different aspects: the specification process and the verification process.

The first implementation of our framework (see Fig. 1) is realized using a functional and object-oriented language: STk [8], a Scheme with an object system very close to CLOS [20, 14]. STk is a good compromise in order to combine object paradigms and the ability to carry out formal operations.

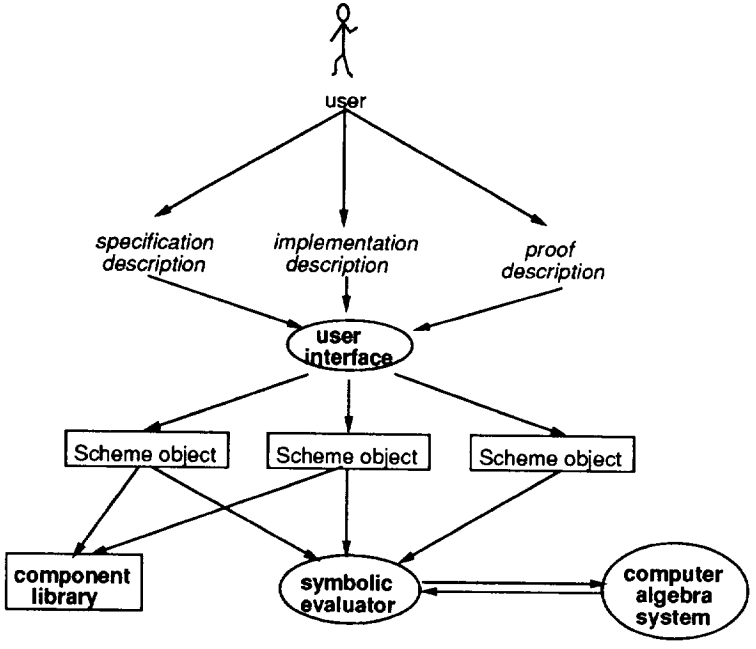


Fig. 1. Overview of the framework

4.1 The Specification Process

We first outline the library from which processor components are built up and then present a specification interface that we have designed in order to facilitate the description process.

A library of components. The attribute `state` of the processor is a set of components built up from a library that consists of a collection of classes describing the variety of structural or behavioural aspects.

Structural Classes: the main data types we use are naturals (**NAT**), bit vectors (**BV**) and arrays of bit vectors (**ARRAY**). Major generic functions are **read** (read contents), **write** (write contents) and **size** (get number of bits). Specific methods are also defined on the different structural classes.

Behavioural Classes: to represent temporal behaviour, we distinguish between what is called in [1] "stored variables" (registers, memories, ...) and "instantaneous variables" (buses or wires). So we have defined the behavioural classes **MEMO** and **NO-MEMO** which are "mixins" [5].

Using multiple inheritance, we get the major component classes to model processors. They have exactly one structural class and one behavioural class as superclasses. Figure 2 shows a simplified class hierarchy.

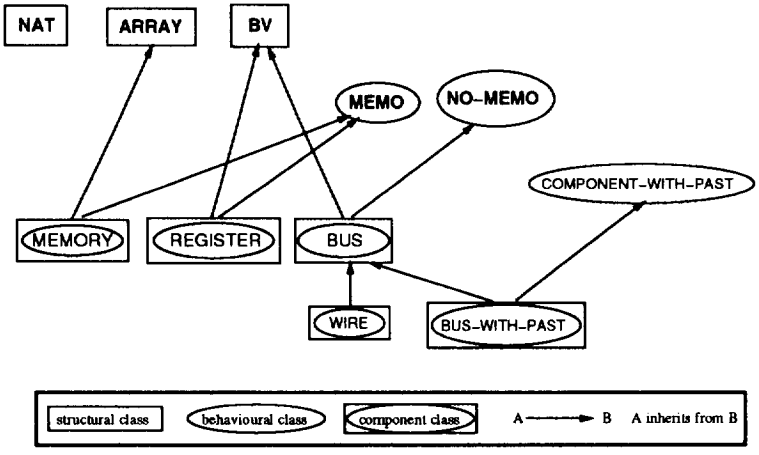


Fig. 2. Hierarchy of component classes

One have to notice that, when building a component class, the behavioural superclass must be more specific than the structural class. In CLOS, this is possible by giving the behavioural class name as the first element of the inheritance list.

Specialized Classes: more specific components such as ROM or stacks, are obtained by specialization of the main classes. Here are examples of class specializations. The first one declares wires as single-bit buses:

```
(define-class Wire (Bus)        ; The wire class inherits from the Bus class.
  ((size :initform 1 :size :getter size)))     ; a wire size is always 1.
```

We define below the **Component-With-Past** class. It allows to keep the history of the component values. This is necessary when modeling complex temporal behaviours such as the communication protocol between the processor and the external memory. The **Component-With-Past** class is also a mixin that does not have any parents. Its **write** method calls for **next-method** that writes the bus value and then memorizes the current value in a stack (**old-values** attribute). This does not produce an error because the class is never instantiated alone: it is *one of the superclasses* of a class, the other superclass is a "component class". The **Component-With-Past** class and its **write** method are defined as follows:

```
(define-class Component-With-Past ()
; The Component-With-Past class is a mixin
   ((old-values :initform '())))                    ; The stack of past values
```

```
(define-method write ((c Component-With-Past) value)
; The write method on components with past
  (next-method)                        ; first call on the normal write method
  (slot-set! c 'old-values                          ; and then push the new value
             (push value
                   (slot-ref c 'old-values)))))
```

A component whose past values are required will be built up by inheriting from the class **Component-With-Past** and a component class. For instance, buses with the capacity of getting old values must be declared as a class whose parents are **Component-With-Past** and **Bus**:

```
(define-class Bus-With-Past (Component-With-Past Bus)
; The Bus-With-Past class inherits from
; the mixin Component-With-Past and the Bus class.
  ())
```

Interface Facilities. In order to help system designers that are not familiar with Scheme programming, we have designed a textual interface to describe processors in a proof-oriented style. This interface is a simple language to enforce designers to fill all the fields of interpreter and proof objects. Its syntax is coming closer to the standard of Hardware Description Languages: VHDL. Starting from this more readable form, a translator automatically generates the Scheme internal representation (see Fig. 1).

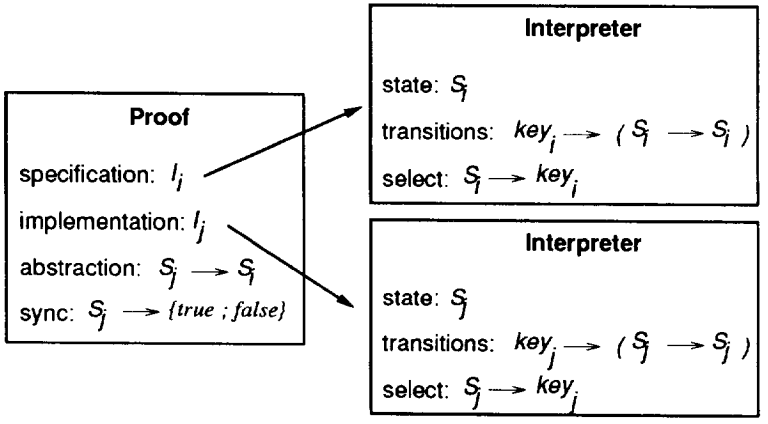


Fig. 3. One proof object and its two interpreters

4.2 The Proof Process

The Verification Steps. Assume we have defined two objects of the class `Interpreter` that specify a processor at levels $level_i$ and $level_j$, and one object of the class `Proof`, that links these two interpreters (Fig. 3). Then, the correctness proof between the two levels is performed according to the following steps:

1. compute an initial state S_j of $level_j$ such that `sync` is true:

$$S_j \ :=\ \texttt{init}(state_j)$$

2. take an abstraction S_i of this state:

$$S_i \ :=\ \texttt{abstraction}(S_j)$$

3. execute the transition selected by the $level_i$ interpreter to get the $level_i$ final state

$$\begin{aligned} trans \ &:=\ \texttt{transitions}(\texttt{select}(S_i)) \\ S_i \ &:=\ \texttt{exec-trans}(trans,\ S_i) \end{aligned}$$

4. execute transitions at $level_j$ until both interpreters are synchronized to get the $level_j$ final state:

```
repeat
   trans := transitions(select(S_j))
   S_j := exec-trans(trans, S_j)
until sync(S_j)=true
```

5. verify that $S_i \equiv \texttt{abstraction}(S_j)$

Broadly speaking, points 1 and 2 compute initial states using symbolic values.

Points 3 and 4 perform the state transitions; "**exec-trans**" carries out the transitions while simplifying expressions (its semantics will be detailed next).

Point 5 consists in a syntactic checking of the equality of the two expressions. For most of the processors and assembly instructions the two expressions (that have been first simplified by our computer algebra system) are syntactically equal. Nevertheless, we have also implemented more elaborated proof techniques, such as Binary Moment Diagrams when syntactic verification is not sufficient [2].

A Symbolic Evaluator of Transitions. Let us now describe more precisely how the transitions that define instruction semantics are performed. Assume we are given a transition *dest* := *source*. Then our symbolic evaluator executes this transition as follows:

- call on **read** method to get the value of *source* if it involves processor components,
- call on the simplification system in order to simplify the resulting value if possible,
- call on the **write** method to modify the value of *dest*.

The symbolic evaluator takes convenience of the object-oriented approach. The evaluation process is generic since **read** and **write** are methods associated to all component classes. Of course, it is not the same thing to read a memory or to read a register but these distinct behaviours are discriminated by inheritance.

A short version of the transition execution method is given below:

```
(define-method exec-trans (dest source (i Interpreter))
; This method executes the assignment of dest with the source value
; over components of the interpreter i
  (let ((rs                    ; the real source value is computed as follows:
          (if (component? source i)   ; if it contains a component of i,
              (simplify (read source))          ; read and simplify.
              (simplify source))))                ; else just simplify
    (write dest rs)))       ; call on the write method on the destination
```

The Verification Kernel. We give here the classes and methods used to implement the proof algorithm. A simplified declaration of the **Interpreter** class *is:*

```
(define-class Interpreter ()                    ; The Interpreter class:
  ((state :init-keyword :state :getter state)
   (select :init-keyword :select :getter select)
   (transitions :init-keyword :transitions :getter transitions)))
```

Some methods are defined for this class in order to reset, update state...

The **Proof** class is as follows:

```
(define-class Proof ()                                  ; The Proof class:
  ((s :init-keyword :specification :getter specification)
   (i :init-keyword :implementation :getter implementation)
   (sync :init-keyword :sync :getter sync)
   (abstraction :init-keyword :abstraction :getter abstraction)
   (prove :init-keyword :prove :getter prove)))
```

The **prove** attribute is here to facilitate the proof by cases: it is a set of *all* instructions of the specification interpreter with all their addressing modes.

The method executing the whole proof returns a Boolean showing the correctness. If the proof fails we can get the instruction being proved and the components whose values are wrong.

```
(define-method proof-ok? ((p Proof))
; Returns true if the implementation of p correctly implements its specification
  (mapand (lambda (instr)                       ; All instructions are correct?
            (instr-ok? (specification p) (implementation p)
                       instr (abstraction p) (sync p)))
          (prove p)))
```

The **instr-ok?** method discriminates on multiple arguments. It proves or disproves a single instruction: it returns a Boolean showing the equivalence of the two interpreters **s** and **i**, for instruction **instr**, and using the state and temporal abstractions **abstr** and **sync**.

```
(define-method instr-ok? ((s Interpreter) (i Interpreter)
                          instr abstr sync)
; Returns true if s and i are equivalent when executing the instruction instr
; and using abstraction functions abstr and sync.
  (update-state i (new-state i instr))              ; give i a new state
  (update-state s (abstr i))                ; give s an equivalent new state
  (update-state s (exec (member (select s) (transitions s)) s))
  ; execute the current instruction at the specification level
  (repeat-until
    (update-state i (exec (member (select i) (transitions i)) i))
    ; execute instructions at the implementation level
    (sync i))                  ; until the temporal abstraction predicate is true
  (equiv? (abstr i) s))                     ; then compare the two final states
```

5 Experimental Results

5.1 Application to Tamarack-3

Tamarack-3 is a benchmark microprocessor described in [9, 12, 19, 22]. Our specification follows the one given in [6] using HOL.

We have specified Tamarack-3 at four levels: the "macro level" which is the assembly level, the "micro level" which is the microinstruction level, the "phase level" which is the decomposition of the "micro level" for each clock phase and the "EBM" which is the structural description of the processor as functional blocks (Fig. 4). We verified the processor in three proof steps between adjacent levels.

At the macro level, components are an accumulator, a register keeping an address to return from subroutines, a program counter, a register latching an external signal of interrupt request, and a register whose value is 1 when an interrupt is being processed. The external memory is also described at this level.

Here is an extract of the specification of Tamarack-3 at the macro level:

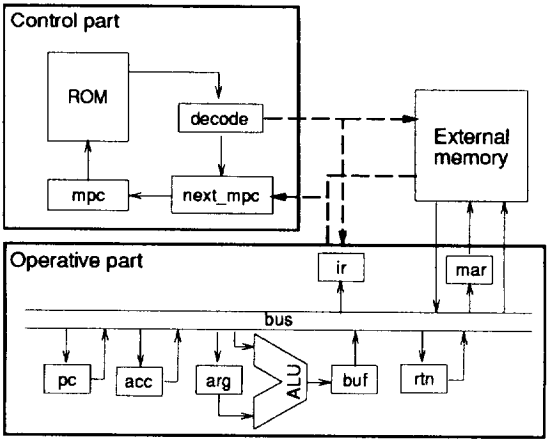


Fig. 4. Structure of the Tamarack-3 microprocessor.

```
(make Interpreter :name 'macro-tamarack
; An instance of the Interpreter class
   :state                              ; The state slot is a list of components
   (list (make Register :name 'pc :size 16)
         ; the program counter is a 16-bit register,
         (make Register :name 'acc :size 16)
         ; as the accumulator
         (make Register :name 'rtn :size 16)
          ; and the return address register,
         (make Memory :name 'mem :size 16)
          ; the external memory is a 16-bit word array,
         (make Register :name 'iack :size 1)
          ; the interrupt acknowledge and
         (make Register :name 'ireq :size 1))
         ; the interrupt request registers are single-bit.
   :select                             ; The select slot is a Lisp expression
   '(if (and ireq (not iack))          ; if an interrupt has to be processed
        ireq-one        ; then return the interrupt pseudo-instruction name
        (decw (fetch mem (address pc))))
                        ; else return the instruction in memory pointed by pc
   :transitions     ; The transitions slot is a list of instruction semantics
   '( (jmp ( (:= pc (fetch mem (address pc))) ))
      ; the jump instruction: assignment to pc
      (adda ( (:= acc (add acc (fetch mem (address pc))))
              (:= pc (inc pc)) ))
      ; the addition instruction: assignment to acc and incrementation of pc
      ...)                                   ; other instructions are here
)
```

At the micro level, the processor components are whose already described at the macro level plus the instruction register, three internal buffer registers, and two registers latching signals between the processor and the external memory. At this level, we also model the microprogram counter.

```
(make Interpreter :name 'micro-tamarack
   :state
   (list (make Register :name 'pc :size 16)
         (make Register :name 'acc :size 16)
         (make Register :name 'rtn :size 16)
         (make Memory :name 'mem :size 16)
         (make Register :name 'iack :size 1)
         (make Register :name 'ireq :size 1)
         (make Register :name 'ir :size 16)    ; the instruction register
         (make Register :name 'arg :size 16)       ; an internal buffer
         (make Register :name 'buf :size 16)             ; another one
         (make Register :name 'mar :size 16)     ; the address register
         (make Register :name 'idle :size 1)   ; for processor/memory
         (make Register :name 'dack :size 1)          ; communication
         (make Register :name 'mpc :size 4))   ; microprogram counter
   :select
   '(case (val4 mpc)                  ; the current microinstruction name is
          (0 . u0)                 ; u0 when microprogram counter value is 0,
          (1 . u1)                 ; u1 when microprogram counter value is 1,
          ... )                                          ; and so on...
   :transitions
   '( (u0                      ; First microinstruction of the fetch sequence:
       ( (:= ir (fetch mem (address pc)))          ; assignment to ir,
         (:= pc (inc pc))                        ; incrementation of pc
         (:= mpc (inc mpc)) ))                                ; and mpc
      (u1 ( ... ))                  ; specify u1 and other microinstructions
       ...)                                              ; in the same way.
)
```

The proof description between macro and micro levels is:

```
(make Proof :name 'macro-micro-tamarack
   :specification macro-tamarack                    ; points the specification
   :implementation micro-tamarack                   ; points the implementation
   :abstraction                ; the state abstraction function is a list where
    ; id is the identity function and nil suppress the corresponding component
   '(id id id id id nil nil nil nil nil nil)
   :sync                       ; the temporal abstraction predicate returns true
   '(= mpc (word4 0))          ; when the microprogram counter value is 0.
)
```

Complete proof. In order to fully prove the Tamarack-3 processor, we have also specified the phase level and the EBM. Our overall goal is to prove that the EBM correctly implements the macro level. In fact we decomposed the proof in three steps: macro versus micro, micro versus phase and phase versus EBM. Transitivity of proofs achieves the complete correctness proof.

5.2 Other Results

We applied our methodology to verify several microprocessors without any change to the `Interpreter` and `Proof` classes, or their methods, nor the proof algorithm of Sect. 4.2. The complexity of these benchmark processors is illustrated in Table 1. The comparative Table 2 shows specification code sizes and proof times for a number of processors already verified with other methods.

Processor	instr.	μinstr.	user registers	word size	addressing modes
AVM-1	30	51	35	32	1
DP-32	20	10	258	32	2
MTI	22	149	38	16	5
Anceau's proc.	8	14	4	16	4
Tamarack-3	8	16	3	16	1

Table 1. Size complexity of processors we specified and proved.

Tamarack-3 and Anceau's processors [1] (pages 181–212) are school processors. Their specifications and proofs do not pose any problem. It is interesting to compare results for Tamarack-3. Stavridou [19] used the OBJ3 theorem prover. It is based on rewriting techniques and one can see that proof time is not acceptable. Furthermore the proof is not automatic. Time proofs in [22] are satisfactory, but to the best of our knowledge, the specifications are given in HOL, which is in our opinion not well-adapted to processor specification.

Processor	#SL	SS (p.)	PT (sec.)	other SS	other PT	ref
AVM-1	4	22	1775	110	58 h	[21]
DP-32	2	9	470			
MTI	2	17	1973			
Anceau's proc.	2	4	183			
Tamarack-3	4	3	197	17	10 days	[19]
				?	360 sec.	[22]

Table 2. Processors we specified and proved. #SL is the number of different specification levels, SS is approximative specification code size in number of pages, PT is proof time in seconds on a SUN IPC workstation. Other SS and other PT are specification sizes and proof times of other works referenced in the "ref" column.

The other processors are rather complex and their proofs emphasize the usefulness of our specification and proof methodology. DP-32 is a processor described in VHDL in [4]. Its verification has raised one error in its implementation. This proof shows that our framework is able to specify, and then to prove, a processor starting from its VHDL description. A VHDL description style provable in our framework has been derived from its verification [3].

AVM-1 [21] has already been verified by Windley using HOL. Our specification is much more concise than his one, and our proof times are also much more satisfactory.

MTI is a processor designed by CNET in France [15]. It has already been studied but never fully proved. Our system discovered several errors in its design.

6 Conclusion

We have presented an object-oriented approach for processor specification and verification. Any processor at any abstraction level is described by an object of the same interpreter class. A proof between two specification levels is an object of the proof class. The main advantages of this approach is that the verification process is reusable. To verify additional processors, we do not need to modify the verification kernel. Furthermore, the specification process is very easy and systematic since it just consists to instantiate predefined classes. The specifications are very concise and the verification times are very satisfactory compared with other methods.

We are now extending our system in two major directions. First, we are developing a graphical interface for processor description. Thanks to the object framework, this will be a very simple task. Second, we will extend the class of processors we are able to prove, to more complex architectures such as DSP or pipelined architectures. We hope that this extension will only consist to an enrichment of the class hierarchy.

Acknowledgments. We are grateful to J.C. Boussard, E. Kounalis and M. Rueher who had the kindness to comment and improve a preliminary version of this paper.

References

1. F. Anceau. *The Architecture of Microprocessors.* Addison-Wesley Publishing Company, 1986.
2. L. Arditi and H. Collavizza. Binary moment diagrams for verifying loops in microprocessor instructions. Technical report, Laboratoire I3S, Université de Nice – Sophia Antipolis, Nov. 1994.
3. L. Arditi and H. Collavizza. Towards verifying VHDL descriptions of processors. Technical report, Laboratoire I3S, Université de Nice – Sophia Antipolis, Jan. 1995.
4. P. J. Ashenden. *The VHDL Cookbook.* Public Domain, Dept. Computer Science, University of Adelaide, South Australia, first edition, July 1990.
5. G. Bracha and W. Cook. Mixin-based inheritance. In *European Conference on Object Oriented Programming / Object-Oriented Programming Systems, Languages and Applications*, October 1990.
6. M. L. Coe and P. J. Windley. Microprocessor verification: A tutorial. Technical Report LAL-92-10, Laboratory for Applied Logic, Brigham Young University, Provo, Utah, 1992.
7. A. Cohn. A proof of correctness of the Viper microprocessor: the first level. In G. Birtwhistle and P. Subrahmanyam, editors, *VLSI Specification, Verification and Synthesis*, pages 27–71, Calgary, Canada, Jan. 1987. Kluwer Acad. Publishers.
8. E. Gallesio. STklos: a Scheme object oriented system dealing with the TK toolkit. In *Xhibition 94*, San Jose, Jul. 1994. ICS.
9. M. Gordon. HOL, a machine oriented formulation of higher order logic. Technical Report 68, University of Cambridge, Computer Laboratory, 1985.
10. A. Gupta. Formal hardware verification methods: a survey. *Formal Methods in System Design*, 1(2/3):151–238, Oct. 1992.
11. W. A. Hunt Jr. Microprocessor design verification. *Journal of Automated Reasoning*, 5(4):429–460, Dec. 1989.
12. J. J. Joyce. Formal verification and implementation of a microprocessor. In G. Birtwhistle and P. Subrahmanyam, editors, *VLSI Specification, Verification and Synthesis*, pages 129–157, Calgary, Canada, Jan. 1987. Kluwer Acad. Publishers.
13. B. Meyer. *Object-Oriented Software Construction.* Prentice Hall International, 1988.
14. A. Paepcke. *Object-oriented programming : the CLOS perspective.* MIT press, 1993.
15. J. Pulou, J. Rainard, and P. Thorel. Microprocesseur à test intégré MTI – description fonctionnelle et architecture. Technical Report NT/CNS/CC/59, CNET, Grenoble, Jan. 1987.
16. J. Rumbaugh, M. Blaha, W. Premerlani, F. Eddy, and W. Lorensen. *Object-oriented Modeling and Design.* Prentice Hall, 1991.
17. R. Sekar and M. Srivas. Equational techniques. In G. Birtwistle and P. Subrahmanyam, editors, *Current Trends in Hardware Verification and Automatic Theorem Proving*, pages 173–217, New-York, 1989. Springer-Verlag.
18. M. Srivas and M. Bickford. Formal verification of a pipelined microprocessor. *IEEE Software*, 7(5):52–64, Sept. 1990.

19. V. Stavridou. *Formal Methods in Circuit Design.* Number 37 in Cambridge Tracts in Theoretical Computer Science. Cambridge University Press, 1993.
20. G. L. Steele Jr. *Common Lisp the Language.* Digital Press, second edition, 1990.
21. P. J. Windley. *The Formal Verification of Generic Interpreters.* PhD thesis, University of California, Division of Computer Science, 1990.
22. Z. Zhu, J. Joyce, and C. Seger. Verification of the Tamarack-3 microprocessor in a hybrid verification environment. In J. J. Joyce and C.-J. H. Seger, editors, *Higher Order Logic Theorem Proving and Its Applications.* 6^{th} *Int. Work. HUG'93*, volume 780 of *LNCS*, pages 253–266, Vancouver, Canada, Aug. 1993. Springer-Verlag.