

# Correctness and Verification of Hardware Systems Using Maude

N. A. Harman\*

Department of Computer Science, University of Wales Swansea, Singleton Park,  
Swansea SA2 8PP

**Abstract.** We consider models of hardware systems, within a well-developed set of mathematical tools based on an equational, algebraic model. We implement these tools using Maude, an equational, algebraic language with strong meta-language tools and an efficient term rewriting implementation. Maude has the same mathematical basis as the existing formal tools and it is fast. We consider the process of verification, and apply it to a simple illustrative pipeline.

Microprocessors, and related systems, are modelled as *iterated maps*. *Initialisation functions* act as an invariant when applying *one-step theorems* to reduce formal verification to state exploration.

## 1 Introduction

This paper considers models of hardware systems and their correctness, represented within a well-developed set of mathematical tools, and the verification of such systems using Maude [3]. The mathematical tools have been developed by addressing case studies, and are based on an equational, algebraic model. Maude is an equational, algebraic language with strong meta-language tools and an efficient term rewriting implementation. In this paper, we consider the process of modelling and verification, and apply it to a simple illustrative example. In [13] we consider the verification of a pipelined microprocessor. Currently, work is progressing on a superscalar implementation of the same architecture.

Microprocessors, and related systems, are modelled as *iterated maps*  $F : T \times A \rightarrow A$ , where  $T$  is a clock, dividing up time and  $A$  represents the state-set of the microprocessor (registers and memories).  $F$  is equationally defined as follows

$$\begin{aligned} F(0, a) &= h(a), \\ F(t + 1, a) &= f(F(t, a)), \end{aligned}$$

where  $f$  is a *next-state function* defining state evolution, and  $h$  is an (optional) *initialisation function*, ensuring/enforcing consistency of *initial state*  $a$ . Initialisation function  $h$  also acts as an invariant when applying the *one-step theorems*

---

\* This work is supported by UK EPSRC grants GR/N15955 and GR/M82202

(Sect. 2.6, [9, 7]) to reduce formal verification to state exploration. The nature of clock  $T$  and of state set  $A$  depend on the level of abstraction of the system being modelled. In this paper, we omit discussion of input and output: however, this is easily accommodated [15, 7]. To illustrate our techniques, we formally verify a simple implementation of a pipelined system. Maude was chosen as the appropriate tool to automate the verification of examples because (a) it has the same mathematical basis as the existing formal tools; (b) it is fast (approximately 700K rewrites per second on a 700MHz Pentium III, when applied to representative examples), giving useful results with widely-available hardware; (c) the integrated meta-level tools allow proof strategies to be constructed quickly, and easily modified; and (d) it is easy to learn, enabling meaningful results to be generated quickly. However, Maude is only one of a range of software tools that could be used to implement the theoretical tools presented here: for example, initial work was undertaken with PVS [27].

This paper forms part of a series on an algebraic theory of microprocessors and other systems. In [14, 15] mathematical models of simple, *microprogrammed* examples are considered, respectively with and without *input streams*. In [9], *correctness models* for microprocessors are considered, and the formal verification process examined. In [8, 10] models of *superscalar* processors are examined by means of a substantial example. The process of verification is discussed in [11]. An extended account of some of this work is [7]. In addition, work has been undertaken on high-level languages and compilers [29], and the Java Virtual Machine [30]. The principle interest to date has been the construction of theoretical models of microprocessors, their correctness, and their verification. The basic premise of the work being that well worked-out foundational models are an important part of the engineering process. To that end, work has progressed on a set mathematical tools, intended to enable modelling of the behaviour and correctness of a variety of hardware systems, in a modular and software tool-independent way. However, in this paper and in [13] we consider the implementation of these mathematical tools within Maude.

The structure of this paper is as follows. In Sect. 1.1 we consider related work. In Sect. 2 we introduce fundamental algebraic tools. In Sect. 3 we introduce Maude. In Sect. 4 we introduce an abstract case study, and a pipelined implementation. In Sect. 5, we consider how to use Maude to verify the correctness of the implementation. Finally, in Sect. 6 we summarise our techniques and their applicability.

### 1.1 Related Work

The principal difference between the series of work of which this forms a part, and related work in the field is that the emphasis here is on building theoretical/mathematical models of system, rather than on addressing [often substantial] examples within software tools, and hence a key underlying theme of other work has been the need for efficient verification strategies. Key work on pipelined microprocessor verification includes [25, 26] on AAMP5, a processor of some complexity, and its verification in PVS [27] (recent accounts of this work are

[6, 28]: see also [17]); [32] on UINTA, a processor of moderate complexity, and its verification in HOL [12]; and [2] on a simple three-stage ALU pipeline and a fragment of the DLX architecture [16]. A refinement of this approach, more applicable to out-of-order systems and long pipelines is [19, 20]. In addition, superscalar processors have been addressed: in particular, the increased complexity of verification in the face of complex timing behaviour [31, 1, 5, 25]. [21, 4] use a variant of *Haskell* called *Hawk*, and Isabelle for proofs; and [18] additionally considers exception processing in such an environment.

The intuitive models in [25, 26, 32] are conceptually similar to our own [14, 15, 8]. However, there are differences, particularly in the approach to time, and timing abstraction. For example, in [32, 25, 26] state elements in the specification are viewed as distributed in time in the implementation, as a consequence of pipelining. We take the view that rather than being temporally shifted, such state components are fundamentally different at the levels of specification and implementation, and that all state elements of the specification are [some function of] state elements in the implementation at a single point in time (see [9, 7]). Consequently, we maintain a separation between data and temporal abstraction functions with the aim of clarity of *modelling*, possibly sacrificing some efficiency in *implementation*. In addition, and unusually, time is explicitly present in our model. Although some pains are taken to remove explicit time from the verification process (Sect. 2.6), recall that our principle interest has been in theoretical models of systems, of which microprocessors are only one example.

## 2 Algebraic Formalisms

Computer systems are modelled in an algebraic framework using primitive recursive functions. We omit detailed discussion of algebraic specification methods here, and refer the reader to (among many) [33, 24]. Functions are defined by equations, primarily using definition by cases and primitive recursion. Time is modelled using a *clock algebra* and systems are modelled with [many-sorted] *state algebras*.

A many-sorted algebra consists of *carrier sets* and *functions* ranging over the carrier sets. For example,

$$(A_1, A_2, \dots, A_l \mid f_1, f_2, \dots, f_m)$$

denotes a many-sorted algebra with carrier sets  $A_1, A_2, \dots, A_l$  and functions  $f_i$  of the form

$$f_i : A_{s_1} \times A_{s_2} \times \dots \times A_{s_n} \rightarrow A_s,$$

where  $1 \leq i \leq m$  and  $1 \leq s, s_j \leq l$  for  $1 \leq j \leq n$ , and each  $f_i$  is defined by one or more equations. The value  $n$  is called the *arity* of function  $f_i$  and if  $n = 0$  then  $f_i$  is called a *constant*. Many-sorted algebras map conveniently to Maude modules.

## 2.1 Clocks and Iterated Maps

A system starts at time zero in an *initial* state  $h(a)$  where  $h$  is called an *initialisation function*. Subsequent system states are determined by a *next-state function*  $f$  and enumerated by a clock  $T$ . The function  $h$  (which may be the identify function) constrains the number of possible state sequences, thus enabling a more flexible definition of correctness. For example, most pipelined designs will not function correctly if the pipeline is initially filled with arbitrary data: see Sect. 4.

**Definition 1.** A clock is an algebra  $(T \mid 0, +1)$ , where  $T$  is a set of clock cycles,  $0 \in T$  is the initial clock cycle and  $+1 : T \rightarrow T$  is the next (or successor) clock cycle function.

A clock cycle need not represent a constant subdivision of time, but will denote an interval between significant events. The definition of ‘significant’ will depend on the level of temporal abstraction we are considering. For example, we might use an *instruction clock* to represent the execution of instructions in a microprocessor. Each cycle of the clock would typically last different amounts of real time, because instruction execution times vary in most processor implementations.

**Definition 2.** Let  $T$  be a clock and let  $A$  be any non-empty set representing a state-space. An iterated map  $F : T \times A \rightarrow A$  is a primitive recursive function defined by

$$\begin{aligned} F(0, a) &= h(a), \\ F(t + 1, a) &= f(F(t, a)) \end{aligned}$$

where  $h : A \rightarrow A$  and  $f : A \rightarrow A$  are primitive recursive functions respectively called the initialisation and next-state functions of the state function  $F$ .

Typically, in a microprocessor,  $A$  will be a cartesian product of components representing registers and memories. By requiring  $f$  and  $h$  to be primitive recursive functions, or to have primitive recursive bounds, we eliminate all potential difficulties with partial functions, and non-termination. In practice, this condition is not restrictive.

## 2.2 Data Abstraction

Data abstraction is modelled as a surjective mapping between two state-spaces. Let  $\psi : B \rightarrow A$  be a surjective map between two non-empty sets  $A$  and  $B$ . Surjectivity ensures that all abstract states in the set  $A$  have at least one representative in the set  $B$ . Data abstraction maps are often projections between two composite state-spaces, for example, a map  $\psi$  from  $B = B_1 \times B_2 \times \dots \times B_m$  to  $A = B_{i_1} \times B_{i_2} \times \dots \times B_{i_n}$  defined as follows

$$\psi(b_1, b_2, \dots, b_m) = (b_{i_1}, b_{i_2}, \dots, b_{i_n})$$

where  $1 \leq i_j \leq m$ ,  $1 \leq j \leq n$  and  $n \leq m$ . In this case, the implementation state-space  $B$  contains, without modification, all the components of the abstract state-space  $A$  (the sets  $B_{i_j}$ ) together with components strictly unique to the implementation.

### 2.3 Temporal Abstraction: Retimings and Immersions

This section formally defines *retimings* (not related to the retimings of [22]) and *immersions*. Two clocks are related using a temporal abstraction map, or retiming. Retimings are characterised by three properties: (i) cycle zero of one clock is always mapped to cycle zero of the other; (ii) the mapping is surjective; and (iii) the mapping is monotonic. Monotonicity ensures there is never a discrepancy, after abstraction, in the temporal ordering of events because, for all  $s, s' \in S$  if  $s' \geq s$ , then  $\lambda(s') \geq \lambda(s)$  where  $\lambda$  is a retiming.

**Definition 3.** A retiming  $\lambda$  is a surjective and monotonic map between two clocks such that  $\lambda(0) = 0$ . The set of all retimings from clock  $S$  to clock  $T$  is denoted by  $Ret(S, T)$ .

The immersion  $\bar{\lambda}$  of a retiming  $\lambda \in Ret(S, T)$  is defined by

$$\bar{\lambda}(t) = \text{least } s \in S \text{ such that } \lambda(s) = t.$$

The set of all immersions of retimings in  $Ret(S, T)$  is denoted by  $Imm(S, T)$ .

Note: Although an immersion is defined by unbounded minimalisation it is total because retimings are surjective.

**Corollary 1.** If  $\lambda \in Ret(S, T)$  is a retiming then  $\lambda\bar{\lambda} = id_T$ .

Given two clocks  $S$  and  $T$  related by retiming  $\lambda \in Ret(S, T)$ , and a clock cycle  $s \in S$ , it is commonly necessary to identify the clock cycle  $s' \in S$  such that  $s'$  is the first cycle of  $S$  where  $\lambda(s') = \lambda(s)$ .

**Definition 4.** The function  $start : Ret(S, T) \rightarrow [S \rightarrow S]$  is defined by

$$start(\lambda) = (\bar{\lambda}\lambda).$$

### 2.4 State-Dependent and Uniform Retimings

In practice, retimings (and immersions) are defined relative to state transition.

**Definition 5.** A state-dependent retiming  $\lambda : A \rightarrow Ret(S, T)$  is a map from states to retimings. The set of all state-dependent retimings from state-space  $A$  to retimings in  $Ret(S, T)$  is denoted by  $Ret(A, S, T)$ .

For each state of an implementation there is an associated state-dependent retiming. *Uniform* retimings provide a strong connection between the states generated by a state function  $F$ , from an initial state  $a$ , and the one retiming associated with the state  $a$ . This connection is achieved by associating a *duration* with each state in the state-space of  $F$ . The following definition first appeared in [7].

**Definition 6.** Let  $F : S \times A \rightarrow A$  be an iterated map and  $dur : A \rightarrow S^+$  be a map from states to a positive number of clock cycles. The uniform retiming with respect to  $F$  and  $dur$  from a clock  $S$  to a slower clock  $T$ , is the state-dependent retiming  $\lambda \in Ret(A, S, T)$  such that, for all  $a \in A$  and  $t \in T$

$$\begin{aligned}\bar{\lambda}(a)(0) &= 0, \\ \bar{\lambda}(a)(t+1) &= dur(F(\bar{\lambda}(a)(t), a)) + \bar{\lambda}(a)(t)\end{aligned}$$

where  $\bar{\lambda} \in Imm(A, S, T)$  is the immersion of  $\lambda$ . The singleton set containing the uniform retiming with respect to  $F$  and  $dur$  is denoted by  $URet_F^{dur}(A, S, T)$ .

Suppose that  $F$  represents the implementation of some system over a clock  $S$ , and that  $T$  is the (slower) clock of the corresponding specification. Then specification clock cycle  $t+1 \in T$  lasts  $dur(x)$  cycles of clock  $S$ , where  $x = F(\bar{\lambda}(a)(t), a)$  is the state of  $F$  on clock cycle  $\bar{\lambda}(a)(t) \in S$ . That is, the cycle of implementation clock  $S$  corresponding with the start of the previous specification clock cycle  $t \in T$ . Note that  $dur$  is a function only of state, and consequently the number of cycles corresponding with any state is independent of the numerical value of  $t \in T$ .

**Definition 7.** A state-dependent retiming  $\lambda \in Ret(A, S, T)$  is uniform with respect to an iterated map  $F : S \times A \rightarrow A$  if, and only if, there exists a function  $dur : A \rightarrow S^+$  such that  $\lambda \in URet_F^{dur}(A, S, T)$ . The set of all uniform retimings with respect to  $F$  is denoted by  $URet_F(A, S, T)$ .

## 2.5 Implementation Correctness

This section provides a definition of correctness through the comparison of two algebras: the state function of an implementation is compared with that of an abstract specification. For this comparison, the state sequences specified by the implementation are mapped onto those of the specification by a data abstraction map  $\psi$  and a temporal abstraction map  $\lambda$ .

**Definition 8.** A state function  $G : S \times B \rightarrow B$  can be called a correct implementation of a state function  $F : T \times A \rightarrow A$  with respect to data abstraction map  $\psi : B \rightarrow A$  and a state-dependent retiming  $\lambda \in Ret(B, S, T)$  if, and only if, for all  $b \in B$  and  $s = start(\lambda(b))(s)$

$$F(\lambda(b)(s), \psi(b)) = \psi(G(s, b)).$$

**Corollary 2.** A map  $G$  is a correct implementation of  $F$  with respect to  $\lambda$  and  $\psi$  if, and only if, the following diagram commutes for all  $b \in B$  and  $s = start(\lambda(b))(s)$

$$\begin{array}{ccc} T \times A & \xrightarrow{F} & A \\ \uparrow & & \uparrow \\ & (\hat{\lambda}, \hat{\psi}) & \psi \\ S \times B & \xrightarrow{G} & B \end{array}$$

where  $\hat{\lambda} : S \times B \rightarrow T$  is a cartesian form of  $\lambda \in \text{Ret}(A, S, T)$  defined by  $\hat{\lambda}(s, b) = \lambda(b)(s)$  and  $\hat{\psi} : S \times B \rightarrow B$  is defined by  $\hat{\psi}(s, b) = \psi(b)$  [7].

Correctness is required to hold at all ‘start’ clock cycles. That is, states of the implementation corresponding with [observable] specification states. This is expressed with the equation  $s = \text{start}(\lambda(b))(s)$ . All cycles such that  $s = \text{start}(\lambda(b))(s)$  are enumerated by the immersion  $\bar{\lambda} \in \text{Imm}(B, S, T)$ .

## 2.6 Time-Consistency and the One-Step Theorems

Iterated map state functions are *time-consistent* if they possess the following property: for all times  $s \in \bar{\lambda}(B)$ , if the clock is reset to zero and the current state becomes an initial state, then is there any noticeable effect upon future state evolution? An iterated map  $F : S \times A \rightarrow A$  is time-consistent if its initialisation function  $h : A \rightarrow A$  characterises a state invariant. Formally,  $h(a) = a$  for all states  $a \in F(\bar{\lambda}(A) \times A)$  in the range of  $F$ .

**Definition 9.** An iterated map  $F : S \times A \rightarrow A$  is time-consistent with respect to a state-dependent retiming  $\lambda \in \text{Ret}(A, S, T)$  if, and only if, for all  $a \in A$  and  $t_1, t_2 \in T$

$$F(s_1 + s_2, a) = F(s_1, F(s_2, a))$$

where  $s_2 = \bar{\lambda}(a)(t_2)$  and  $s_1 = \bar{\lambda}(F(s_2, a))(t_1)$ .

**Corollary 3.** If  $F$  is an iterated map with initialisation function  $h$  and next-state function  $f$  then  $F$  is time-consistent with respect to  $\lambda$  if, and only if, the following diagram commutes for all  $a \in A$ ,  $s_2 = \bar{\lambda}(a)(t_2)$  and  $s_1 = \bar{\lambda}(F(s_2, a))(t_1)$

$$\begin{array}{ccc} a & \xrightarrow{f^{s_2} h} & F(s_2, a) \\ \downarrow f^{s_1+s_2} h & & \downarrow f^{s_1} h \\ F(s_1 + s_2, a) & \equiv & F(s_1, F(s_2, a)) \end{array}$$

That is, if:  $f^{s_1+s_2} h = f^{s_2} h f^{s_1} h$  [7].

Clearly, the initialisation function  $h$  must be carefully chosen to ensure time-consistency. Initialisation functions must be sufficiently weak to avoid changing any initial state consistent with correct future execution, while being strong enough to correct all invalid initial states. In effect, we are constructing an *invariant*. In practice,  $h$  may be complex and difficult to construct (though see [7, 13] for a systematic technique that can often, though not always, be used).

The following *one-step theorems* enable verification to be reduced to state exploration, without induction. Theorem 1 states that if  $\lambda \in \text{Ret}(B, S, T)$  is a uniform retiming then time-consistency with respect to  $\lambda$  is sufficiently verified by examining the implementation at times  $t = 0, 1$ . Theorem 2 states that retiming uniformity and implementation time-consistency are sufficient conditions to enable correctness to be wholly verified by comparing specification and implementation at times  $t = 0, 1$ .

**Theorem 1.** *If  $F : T \times A \rightarrow A$  is an iterated map with initialisation function  $h : A \rightarrow A$  and if  $\lambda \in URet_F^{dur}(A, S, T)$  is a uniform retiming then  $F$  is time-consistent with respect to  $\lambda$  if, and only if, for all  $a \in A$*

$$F(0, a) = h(F(0, a)), \text{ and } F(\bar{\lambda}(a)(1), a) = h(F(\bar{\lambda}(a)(1), a)).$$

*Proof.* See [7, 9].

**Theorem 2.** *Let  $F : T \times A \rightarrow A$  and  $G : S \times B \rightarrow B$  be iterated maps. Let  $\psi : B \rightarrow A$  be a data abstraction map and let  $\lambda \in URet_G(B, S, T)$  be a uniform retiming. If  $F$  is non-initialised, and  $G$  is time-consistent with respect to  $\lambda$  then  $G$  is a correct implementation of  $F$  if, and only if*

$$F(0, \psi(b)) = \psi(G(0, b)), \text{ and } F(1, \psi(b)) = \psi(G(\bar{\lambda}(b)(1), b)).$$

*Proof.* See [7, 9].

### 3 Introduction to Maude

Maude is an equationally-based, algebraic language with a term rewriting implementation [3]. The following simple algebra, or module, representing a memory, illustrates most of the key features.

```
fmod MEM is
  sorts Mem MAR Word .

  op _[_] : Mem MAR -> Word .          *** Memory read
  op _[_/_] : Mem Word MAR -> Mem .    *** Memory write

  var M : Mem .
  vars A B : MAR .
  var W : Word .

  eq M[W / A] [A] = W .
  ceq M[W / A] [B] = M[B] if A /= B .
endfm
```

The module MEM contains three new sorts Mem, MAR and Word, and two new operations \_[\_] and \_[\_/\_], representing memory reading and writing. The relationship between the memory read and write operations is defined by a pair of equations, one of which is conditional. Note that the memory read and write operations are defined in mixfix syntax, where arguments will replace the \_ characters in the operation name. It is also possible to use [traditional] prefix syntax by omitting the \_ characters. Other key basic features of Maude not shown above include provision to define hierarchies of algebras via a module importation mechanism, and the ability to define hierarchies of types.

## 4 An Abstract Pipeline Case Study

An abstract pipeline  $TR - PIPE$  is introduced, along with the corresponding simple, non-pipelined specification  $TR$ . The verification of  $TR - PIPE$  with respect to  $TR$  is described in Sect. 5. The intent of this example is to illustrate the process of verification of systems represented as iterated maps in Maude, rather than represent a realistic case study ( $TR - PIPE$  is straightforwardly verified by hand in [7, 11]). A more substantial pipelined microprocessor is presented in [13], and work proceeds on a superscalar example.

The device  $TR$  transfers data from a source memory to a destination memory, and in the process the data is transformed using a composite operation  $f = (f_1 \circ f_2 \circ f_3 \circ f_4)$ . The pipelined implementation  $TR - PIPE$  performs the operations  $f_i$  in time.

### 4.1 The Specification $TR$

The abstract device  $TR$  contains two memories and two memory-address registers. The memory state-space is  $M \in [MAR \rightarrow Word]$  where  $MAR$  and  $Word$  are any non-empty sets. We omit the mathematical definition of memory: the definition in Maude can be found in Sect. 3. The state-space of  $TR$  is

$$State_{TR} = M \times MAR \times MAR \times M.$$

The device  $TR$  is specified with state function  $TR : T \times State_{TR} \rightarrow State_{TR}$  and next-state function  $tr : State_{TR} \rightarrow State_{TR}$

$$\begin{aligned} TR(0, src, msr, mdr, dst) &= (src, msr, mdr, dst), \\ TR(t + 1, src, msr, mdr, dst) &= tr(TR(t, src, msr, mdr, dst)), \\ tr(src, msr, mdr, dst) &= (src, msr + 1, mdr + 1, \\ &\quad dst[f(src[msr])/mdr]) \end{aligned}$$

where  $src \in M$ ,  $msr \in MAR$ ,  $mdr \in MAR$  and  $dst \in M$ . The next-state function  $tr$  updates the destination memory  $dst$  at location  $mdr$  with  $f(src[msr])$ , and increments both memory-address registers. The primitive recursive function  $f : W \rightarrow W$  is not explicitly defined, but the pipelined implementation assumes  $f = (f_1 \circ f_2 \circ f_3 \circ f_4)$  for some

$$f_1 : W \rightarrow W_1, f_2 : W_1 \rightarrow W_2, f_3 : W_2 \rightarrow W_3, f_4 : W_3 \rightarrow W$$

where  $W_1$ ,  $W_2$  and  $W_3$  are arbitrary non-empty sets. For brevity,  $f_1 \circ f_2 : W \rightarrow W_2$  is denoted  $\vec{f}_2$ , and  $\vec{f}_2 \circ f_3 : W \rightarrow W_3$  is denoted  $\vec{f}_3$ .

The specification  $TR$  is straightforwardly represented by the following Maude code. The first module represents the state tuple; the second module the iterated map and next-state function. The `protecting` keyword specifies a module to import. The [omitted] module `INT` defines the natural numbers and integers, and associated operations.

```

***
*** Tupling and projection operations
***
fmod TRSTATE is
  protecting MEM .    *** See Sect. 3

  sort TrState .

  *** Tupling operator
  op --,--, : Mem Int IntMem -> TrState .
  *** Projection operators
  ops src dst : TrState -> Mem .
  ops msr mdr : TrState -> MAR .

  var TS : TrState .
  vars Src Dst : Mem .
  vars Msr Mdr : Int .
  eq src(Src, Msr, Mdr, Dst) = Src .
  eq dst(Src, Msr, Mdr, Dst) = Dst .
  eq msr(Src, Msr, Mdr, Dst) = Msr .
  eq mdr(Src, Msr, Mdr, Dst) = Mdr .
endfm

***
*** The specification of TR
***
fmod TR is
  protecting INT .
  protecting TRSTATE .
  op tr : Int TrState -> TrState .
  op next : TrState -> TrState .
  op f : Word -> Word .
  var T : Nat .
  var ST : TrState .
  vars SRC DST : Mem .
  vars MDR MSR : MAR .
  eq tr(0,ST) = ST .
  eq tr(T, ST) = next(tr(T - 1,ST)) if T > 0 .

  eq next(SRC, MSR, MDR, DST) = SRC, MSR + 1, MDR + 1,
    DST[f(SRC[MSR])/MDR] .

endfm

```

## 4.2 A Self-Initialising Four-Stage Pipeline

We now present a pipelined implementation  $TR - PIPE$  of  $TR$  in which the operation  $f$  is performed by a four-stage pipeline. Initially, we assume the pipeline may be full, partially-full, or empty. We fill the pipeline using a counter  $ctr \in \{1, 2, 3, 4\}$ . If  $ctr = 1$  then the pipeline is assumed to be full. If  $ctr = 4$  then the pipeline is assumed to be empty with the pipeline components  $w_1$ ,  $w_2$  and  $w_3$  containing *jun* values. The pipeline is filled by decrementing  $ctr$  while ensuring junk values are not stored in the destination memory  $dst$  until  $ctr = 1$ .

The state-space  $State_{TR-PIPE}$  is defined as follows.

$$State_{TR-PIPE} = \{1, 2, 3, 4\} \times M \times MAR \times W_1 \times W_2 \times W_3 \times MAR \times M,$$

where  $W_1$ ,  $W_2$  and  $W_3$  are state elements used to store the intermediate values in the [pipelined] computation of  $f$ . The iterated map state function  $TR - PIPE : S \times State_{TR-PIPE} \rightarrow State_{TR-PIPE}$  is defined by

$$\begin{aligned} TR - PIPE(0, \vec{\sigma}) &= init(\vec{\sigma}), \\ TR - PIPE(s + 1, \vec{\sigma}) &= trpipe(TR - PIPE(s, \vec{\sigma})) \end{aligned}$$

where  $\vec{\sigma} = (ctr, src, msr, w_1, w_2, w_3, mdr, dst)$ ,

$$init(\vec{\sigma}) = \begin{cases} (4, src, msr, w_1, w_2, w_3, mdr, dst), & \text{if } ctr > 1; \\ (1, src, msr, f_1(src[msr - 1]), \\ \quad \vec{f}_2(src[msr - 2]), \vec{f}_3(src[msr - 3]), mdr, dst), & \text{if } ctr = 1 \end{cases}$$

and

$$trpipe(\vec{\sigma}) = \begin{cases} (ctr - 1, src, msr + 1, f_1(src[msr]), \\ \quad f_2(w_1), f_3(w_2), mdr, dst), & \text{if } ctr > 1; \\ (1, src, msr + 1, f_1(src[msr]), \\ \quad f_2(w_1), f_3(w_2), mdr + 1, dst[f_4(w_3)/mdr]), & \text{if } ctr = 1. \end{cases}$$

The initialisation function  $init : State_{TR-PIPE} \rightarrow State_{TR-PIPE}$  ensures the device is either empty or full at cycle zero. In the case that it is empty, we need take no steps to ensure that the intermediate state components  $w_1, w_2, w_3$  contain values consistent with correct future execution; if it full, then all of  $w_1, w_2, w_3$  must be consistently initialised. We choose to map the intermediate pipeline stages when  $ctr = 2$  or  $ctr = 3$  to an empty pipeline by setting  $ctr = 4$ . This is a stronger initialisation function than is normally permissible: we would generally choose to define functions to consistently initialise a partly-filled pipeline. However, in this particular example, the simpler definition is adequate, and does not violate time-consistency because once the pipeline is full it remains so, and therefore these states will never re-occur. If it was possible in our example for the pipeline to become emptied during operation, it would be necessary to consider intermediate states.

If  $ctr = 1$ , then  $w_1, w_2$  and  $w_3$  should contain  $f_1(src[msr - 1]), \vec{f}_2(src[msr - 2])$  and  $\vec{f}_3(src[msr - 3])$ . Hence we initialise the pipeline to contain these values

(which should be precisely the values it contained already, if it was in a state consistent with correct execution).

The next-state function  $trpipe : State_{TR-PIPE} \rightarrow State_{TR-PIPE}$  provides for two cases: if  $ctr = 1$  then the pipeline is full and the pipeline is ready to compute and store values in  $dst$ . But if  $ctr > 1$  then the pipeline is progressively filled. This means  $ctr$  is decremented while the  $dst$  memory is unaltered to prevent storing junk values.

In the interests of brevity, we omit the Maude representation of  $TR - PIPE$ . Structurally, it is very similar to  $TR$ , with one module representing the state tuple, and another representing the iterated map, next-state function and initialisation function. Like  $TR$ , it is a straightforward translation of the mathematical representation of  $TR - PIPE$ .

## 5 Verification in Maude

To verify the correctness of  $TR - PIPE$  with respect to  $TR$ , we first construct the data abstraction map  $\psi$ , and retiming  $\lambda$ , and formulate the correctness statement.

**Proposition 1.** *The map  $TR - PIPE$  is a correct implementation of  $TR$  with respect to data abstraction map  $\psi : State_{TR-PIPE} \rightarrow State_{TR}$*

$$\psi(ctr, src, msr, w_1, w_2, w_3, mdr, dst) = \begin{cases} (src, msr, mdr, dst), & \text{if } ctr > 1; \\ (src, msr - 3, mdr, dst), & \text{if } ctr = 1, \end{cases}$$

and uniform retiming  $\lambda \in URet_{TR-PIPE}^{dur}(State_{TR-PIPE}, S, T)$  where duration function  $dur : State_{TR-PIPE} \rightarrow S^+$  is defined by

$$dur(ctr, src, msr, w_1, w_2, w_3, mdr, dst) = \begin{cases} 1, & \text{if } ctr = 1; \\ 4, & \text{if } ctr > 1. \end{cases}$$

The following Maude module (with some incidental details omitted), defines  $\psi$  and  $dur$ .

```
fmod CORRECT is
...
ops psi : TrStatePipe -> TrState .
op dur : TrStatePipe -> Nat .
op IC : Nat -> Nat .
...
vars SRC DST : Mem .
vars MSR MDR : MAR .
var CTR : Nat .
var TEMP1 : W1 .
var TEMP2 : W2 .
var TEMP3 : W3 .
```

```

var W : Word .

*** Sub-function to set CTR
ceq IC(CTR) = 1 if CTR == 1 .
ceq IC(CTR) = 4 if CTR > 0 .

*** Data abstraction map
eq psi(CTR, SRC, MSR, TEMP1, TEMP2, TEMP3, MDR, DST) =
  case(IC(CTR),
    4 : (SRC, MSR, MDR, DST)),
    1 : (SRC, MSR - 3, MDR, DST) .

*** Duration Function
eq dur(trp(CTR, SRC, MSR, TEMP1, TEMP2, TEMP3, MDR, DST)) = IC(CTR) .

*** Defn of f in terms of f1 ... f4
eq f4(f3(f2(f1(W)))) = f(W) .
...
endfm

```

Note that we define a special *case* operator, instead of using the built-in `if then else fi` operator, or conditional equations: see Sect. 5.1.

To verify the correctness of *TR – PIPE* using the one-step theorems 1 and 2, we need to prove the following.

$$trpipe(0, \vec{\sigma}) = init(trpipe(0, \vec{\sigma})) \quad (1)$$

$$trpipe(dur(\vec{\sigma}), \vec{\sigma}) = init(trpipe(dur(\vec{\sigma}), \vec{\sigma})), \quad (2)$$

$$tr(0, \psi(\vec{\sigma})) = \psi(trpipe(0, \vec{\sigma})) \quad (3)$$

$$tr(1, \psi(\vec{\sigma})) = \psi(trpipe(dur(\vec{\sigma}), \vec{\sigma})) \quad (4)$$

where  $\vec{\sigma} = (src, dst, temp1, temp2, temp3, msr, mdr, ctr)$ . Equations 1 and 2 discharge the obligations of theorem 1, and equations 3 and 4 discharge the obligations of theorem 2.

### 5.1 Verification Strategy: the Maude Meta-Level

A simple term rewriting model is not sufficient to automatically verify the examples we are interested in, because term rewriting will not proceed beyond unresolved conditionals. The simplest approach (and the most efficient in terms of number of rewrites) is to manually explore the cases, by defining a set of constants representing the state elements for each case that needs to be considered. Whilst this approach is trivial for the example in this paper, and it is relatively straightforward for the pipelined microprocessor example in [13], it is not ideal: for more complex examples it is not practical. However, Maude includes well-integrated meta-level tools, enabling Maude modules to be treated

as data types, and the rewriting process to be controlled. We can use these properties to construct a range of verification strategies, tailored to classes of example. In this paper, we employ a simple strategy, and use Maude to automatically construct and check a tree representing all subcases. We do this by defining operations to dynamically extend Maude modules by adding equations, and to direct rewriting.

The following Maude operation, defined within the Maude meta-level, extends a module (of [pre-defined] sort `FModule`) with a new equation  $T1 = T2$ .

```
op AddEq : FModule Term Term -> FModule .
var QI : Qid .
:
eq AddEq((fmod QI is IL SD SSDS ODS VDS MAS EqS endfm),T1,T2) =
      fmod QI is IL SD SSDS ODS VDS MAS
      ((eq T1 = T2 .) EqS) endfm .
```

We omit the definitions of most of the variables for brevity. (Variable `QI` is of sort `Qid` — *quoted identifier*, representing the name of the module; `ODS` is of sort `OpDeclSet` — the set of *operator declarations*, and so on.) The defining equation extends a module (`fmod QI is IL SD SSDS ODS VDS MAS EqS endfm`) with a new equation  $T1 = T2$ .

In order to identify sub-cases, we represent conditionals using special case operators. For example,

```
op case : Sort1 Sort2 Sort2 -> Sort2 .
op _:_ : Sort1 Sort2 -> Sort2 .
```

A typical application of this operator would be as follows:

```
case(S1a,S1b:X,S1c:Y),
```

where the operator evaluates to `X` if  $S1a=S1b$ , and to `Y` if  $S1a=S1c$ . In this example, the case operator decides between two alternatives. However, it is possible to define case operators that select from larger numbers of alternatives, provided the basic syntactic form remains the same. We will define Maude operators to syntactically search for such case operators. At present, all case operators must be defined explicitly using equations. However, it would be possible to do this automatically using `AddEquation`, or to extend the verification strategy to evaluate case operators (at the cost of some efficiency).

We direct the rewriting process using the following Maude code (some sub-functions have been omitted for clarity, and minor simplifications have been made).

```
ops CheckEquals CheckEqualsAux
      CheckEqualsAux2: Module Term Term -> Bool .
```

```

eq CheckEquals(MOD, T1, T2) =
  CheckEqualsAux(MOD, T1, meta-reduce(MOD, T2)) .

eq CheckEqualsAux(MOD, T1, T2) =
  if getCase(T2) == 'fail then
    meta-reduce(MOD,T1) == T2
  else CheckEqualsAux2(MOD,T1,T2) fi .

eq CheckEqualsAux2(MOD, T1, T2) =
  if getCase(T2) == 'fail then
    true
  else
    (CheckEquals(AddEq(MOD, getCase(T2),getCase2(T2)),
      T1, getCase3(T2)) and
     CheckEqualsAux2(MOD, T1, delCase(T2))) fi .

```

The operation `CheckEquals` generates and evaluates the cases, in conjunction with two auxiliary functions, within a module `MOD`. `CheckEquals` reduces `T2` as far as possible using `meta-reduce` (a built-in Maude meta-level operator), and then calls `CheckEqualsAux` to look for unresolved case operators within the resulting reduced term, using `getCase`. The `getCase` operator returns the first argument of the first case operator it finds, or `'fail` if no such operator exists. If no case operators are present, then no further conditionals remain to be evaluated, and so `CheckEqualsAux` reduces `T1` and checks equality with the [reduced] form of `T2`. `CheckEqualsAux2` tries each branch of an unresolved case operator by: adding an equation `getCase(T2) = getCase2(T2)` (asserting the first branch of the case operator); replacing the entire case operator with its first branch (`getCase3(T2)`); and recursively applying `CheckEquals`.<sup>1</sup> The first branch of the case operator is then deleted (`delCase`), and the next branch is recursively considered by `CheckEqualsAux2`. Eventually, all case operators will have been explored. In this way, the tree of all possible cases is constructed, and `CheckEquals` will return true only if `T1 = T2` in all possible cases.

This strategy requires 3628 rewrites to verify the correctness of *TR – PIPE*, compared with 373 for the manual approach (an order of magnitude increase over manual case enumeration is typical: in [13] manual case enumeration results in about 300K rewrites, compared with 2.5M for the method described in this paper). The current code is experimental, and significant efficiency gains are likely. For example, the same sequence of reductions is performed repeatedly (e.g. by `getCase`). In large examples, where the next available case operator is deeply embedded in a term, this is likely to be significant.

---

<sup>1</sup> Note that it is necessary to assert the equation as well as replacing the case operator, because the equation may allow further reduction of the specification term `T1`, as well as resolving other case operators in term `T2`.

## 6 Further Considerations

This paper presents one approach to verifying systems using Maude, by dynamically building and evaluating the tree of possible subcases. This technique is illustrated by application to a simple pipeline example, has also been used to verify a simple pipelined microprocessor [13], and is being used to address a superscalar processor (a minor variation on the example in [8, 10]). However, there are other ways to use Maude. Both the pipeline presented here and the pipelined microprocessor [13] have been verified by manually considering each possible case. This is the most efficient approach in terms of the number of rewrites required. However, manually enumerating the cases takes time, and is only realistically possible with small examples: the pipeline in this paper requires only two cases; the pipelined microprocessor [13] requires 53, which is about the realistic limit. It is estimated that the superscalar example being addressed will require about 2000 cases. In addition, we must also admit the possibility of omitting cases in error. The example from [13] was first completed without running out of physical memory on a 64Mbyte machine, suggesting that there is scope to undertake significantly larger examples on existing hardware without addressing the efficiency of the proof strategy. However, a number of approaches are being followed to increase the size of examples that can be addressed.

An alternative to the manual approach, and that considered here, is a hybrid technique, combining manual and automatic approaches. Sets of constants are used to partially specify cases: in effect, identifying groups of subcases. This would enable verifications to be broken down into more easily-manageable segments. Another approach which is being explored is to automatically generate a Maude module, representing each of the subcases to be considered, from a description of the dependencies present within the system: that is, to statically, rather than dynamically generate the subcases. The verification would thus proceed in two stages. The first to construct the Maude module representing the subcases, and the second to perform the verification. In principle, the actual verification should be equivalent to the [highly efficient] manual process. Finally, we can choose to integrate techniques found in the literature (for example [5]) to improve the efficiency of proof strategies. Note however, since our principal aim is building *theoretical models* (the techniques described here are *direct* implementations of the theoretical model), such efficiency-improving techniques must be integrated into the theory first.

The key advantages of Maude in this work have been: the ease with which examples are mapped from the theoretical model to software; the efficiency of the underlying term rewriting engine; and the ease with which proof strategies are constructed. For example, future case studies (including a memory system) will require inductive reasoning: an inductive proof strategy is currently being constructed (based on an existing example [23]).

## References

1. J Burch. Techniques for verifying superscalar microprocessors. In *Design Automation Conference*, 1996.
2. J Burch and D Dill. Automatic verification of pipelined microprocessor control. In D Dill, editor, *Proceedings of the 6th International Conference, CAV'94: Computer-Aided Verification*, pages 68 – 80. Lecture Notes in Computer Science 818, Springer-Verlag, 1994.
3. M Clavel, F Durán, S Eker, P Lincoln, N MartíOliet, J Meseguer, and J Quesada. Maude: Specification and programming in rewriting logic. Technical report, Computer Science Laboratory, SRI International, 1999.
4. B Cook, J Launchbury, and J Matthews. Specifying superscalar microprocessors in hawk. In M Sheeran, editor, *Workshop on Formal Methods for Hardware (Marstrand, Sweden)*, 1998.
5. D Cyrluk. Inverting the abstraction mapping: A methodology for hardware verification. In A Camilleri M Srivas, editor, *Formal Methods in Computer-Aided Design*, pages 172 – 186. Lecture Notes in Computer Science 1166, Springer-Verlag, 1996.
6. D Cyrluk, J Rushby, and M Srivas. Systematic formal verification of interpreters. In *IEEE International Conference on Formal Engineering Methods (ICFEM'97)*, pages 140 – 149, 1997.
7. A C J Fox. *Algebraic Representation of Advanced Microprocessors*. PhD thesis, Department of Computer Science, University of Wales Swansea, 1998.
8. A C J Fox and N A Harman. An algebraic model of correctness for superscalar microprocessors. In *Formal Methods in Computer-Aided Design*, pages 346 – 361. Lecture Notes in Computer Science 1166, Springer-Verlag, 1996.
9. A C J Fox and N A Harman. Algebraic models of correctness for microprocessors. Technical Report CSR 8-98 (accepted for *Formal Aspects of Computer Science*), University of Wales Swansea, 1998.
10. A C J Fox and N A Harman. Algebraic models of superscalar microprocessor implementations: A case study. In B Möller and J V Tucker, editors, *Prospects for Hardware Foundations*, pages 138 – 183. Lecture Notes in Computer Science 1546, Springer-Verlag, 1998.
11. A C J Fox and N A Harman. Algebraic models of temporal abstraction for initialised iterated state systems: An abstract pipelined case study. Technical Report CSR 21-98 (submitted to *Acta Informatica*), University of Wales Swansea, 1998.
12. M J C Gordon and T F Melham. *Introduction to HOL*. Cambridge University Press, 1993.
13. N A Harman. Verifying a simple pipelined microprocessor using maude. Technical Report Computer Science Report, University of Wales Swansea, 2000.
14. N A Harman and J V Tucker. Algebraic models of microprocessors: Architecture and organisation. *Acta Informatica*, 33:421 – 456, 1996.
15. N A Harman and J V Tucker. Algebraic models of microprocessors: the verification of a simple computer. In V Stavridou, editor, *Proceedings of the 1995 IMA Conference on Mathematics for Dependable Systems*. Oxford University Press, 1997.
16. J L Hennessy and D A Patterson. *Computer Architecture: A Quantative Approach*. Morgan Kaufman, 1996.
17. R Hosabettu, M Srivas, and G Gopalakrishnan. Decomposing the proof of correctness of pipelined microprocessors. In A J Hu and M Y Vardi, editors, *Computer Aided Verification: 10th International Conference*, pages 122 – 134. Springer-Verlag, Lecture Notes in Computer Science 1427, 1998.

18. J Sawada W A Hunt. Processor verification with precise exceptions and speculative execution. In A J Hu and M Y Vardi, editors, *Computer Aided Verification: 10th International Conference*, pages 135 – 147. Springer-Verlag, Lecture Notes in Computer Science 1427, 1998.
19. R B Jones, J U Skakkebæk, and D Dill. Reducing manual abstraction in formal verification of out-of-order execution. In G Gopalakrishnan and P Windley, editors, *Formal Methods in Computer-Aided Design, FMCAD 98*, pages 2 – 17. Springer-Verlag, Lecture Notes in Computer Science 1522, 1998.
20. J U Skakkebæk, R B Jones, and D Dill. Formal verification of out-of-order execution using incremental flushing. In A J Hu and M Y Vardi, editors, *Computer Aided Verification: 10th International Conference*, pages 98 – 109. Springer-Verlag, Lecture Notes in Computer Science 1427, 1998.
21. S Krstic, B Cook, J Launchbury, and J Matthews. A correctness proof of a speculative, superscalar, out-of-order, renaming microarchitecture. Technical report, Oregon Graduate Institute, 1998.
22. C E Leiserson, F M Rose, and J B Saxe. Optimizing synchronous circuitry by retiming. In R Bryant, editor, *Third Caltech Conference on VLSI*, volume 1983, pages 87–116. Computer Science Press, 1803 Research Boulevard, Rockville MD 20850, 1983.
23. F Dur´ an M Clavel, S Eker, and J Meseguer. Building equational proving tools by reflection in rewriting logic. In *Proceedings of the CafeOBJ Symposium '98*, 1998.
24. K Meinke and J V Tucker. Universal algebra. In T S E Maibaum S Abramsky, D Gabbay, editor, *Handbook of Logic in Computer Science*, pages 189 – 411. Oxford University Press, 1992.
25. S Miller and M Srivas. Formal verification of an avionics microprocessor. Technical report, SRI International Computer Science Laboratory CSL-95-04, 1995.
26. S Miller and M Srivas. Formal verification of the AAMP5 microprocessor: a case study in the industrial use of formal methods. In *Proceedings of WIFT 95, Boca Raton*, 1995.
27. S Owre, J Rushby, N Shankar, and M Srivas. A tutorial on using PVS. In *Proceedings of TPCD 94*, pages 258–279. Lecture Notes in Computer Science 901, Springer-Verlag, 1994.
28. M Srivas, H Rueß, and D Cyrluk. Hardware verification using PVS. In T Kropf, editor, *Formal Hardware Verification*, pages 156 – 205. Springer-Verlag, Lecture Notes in Computer Science 1287, 1997.
29. K Stephenson. *An Algebraic Approach to Syntax, Semantics and Compilation*. PhD thesis, University of Wales Swansea Computer Science Department, 1996.
30. K Stephenson. Algebraic specification of the Java virtual machine. In B Möller and J V Tucker, editors, *Prospects for Hardware Foundations*. Lecture Notes in Computer Science 1546, Springer-Verlag, 1998.
31. P Windley and J Burch. Mechanically checking a lemma used in an automatic verification tool. In A Camilleri M Srivas, editor, *Formal Methods in Computer-Aided Design*, pages 362 – 376. Lecture Notes in Computer Science 1166, Springer-Verlag, 1996.
32. P Windley and M Coe. A correctness model for pipelined microprocessors. In *Proceedings of the 2nd Conference on Theorem Provers in Circuit Design*, 1994.
33. M Wirsing. Algebraic specification. In J van Leeuwen, editor, *Handbook of Theoretical Computer Science, Volume B: Formal Models and Semantics*, pages 675 – 788. Elsevier, 1990.