A Case Study in Specifying and Testing Architectural Features

Padmanabhan Krishnan

Department of Computer Science

University of Canterbury, Private Bag 4800

Christchurch, New Zealand

Email: paddy@cosc.canterbury.ac.nz

Abstract

This paper studies the specification and testing of two main architectural features. We consider restricted forms of instruction pipelining and parallel memory models present in the SPARC specification. The feasibility of using an automatic tool, the concurrency work bench, has been demonstrated.

Keywords Architecture, SPARC, specification, verification, CCS, modal logic, concurrency work bench

1 Introduction

Formal specification is the writing of the requirements of a system at a sufficiently abstract level. To gain confidence that a given specification meets the needs of an user, tests on it can be performed. If the outcomes agree with the expected behaviour, one can assume that the given specification expresses what is necessary. A particular form of testing is the construction of an implementation and verifying that the implementation satisfies the specification. However it not always possible to construct an implementation or verify that it satisfies the specification. In such cases one may construct tests using other formalisms

Formal specification and verification of computer hardware has been shown to be feasible. [GB89] describes the specification and construction of an SECD machine using HOL [Gor85], while [Coh88] describes the verification of the Viper architecture. Concurrency aspects of hardware has also been verified in HOL [LD90] where a multiprocessor cache protocol is considered.

In this paper we describe our experience in specifying certain aspects of an architecture and formal testing of the specification. The architecture we consider is the SPARC [Spa91] and the specification language we use is CCS [Mil89]. CCS belongs to a class of formalisms called process algebras, which are used to describe the observational behaviour of concurrent systems. They have been used to specify and verify many systems including communication protocols [Par87, LM87]. A prototype implementation called the Concurrency Work Bench (CWB) to help the specifier test the specifications exists [CPS89, CPS93]. The main reason for choosing the CWB over the HOL system is that CWB performs all the analysis *automatically*, while the HOL system is a proof assistant. This is not to conclude that HOL cannot be used, rather that as a *first* step in the specification and testing process the CWB is easier to use than HOL. The CWB is only one of the specification/validation environments. Implementations of LOTOS [BB89, vVD89] which is based on CCS exist and have been used to specify/verify system [vS89]. We choose the CWB mainly because it was available.

The SPARC architecture was chosen as it is relatively new architecture and addresses some of the issues in multiprocessor systems (the memory model) and supports pipelining whose definition is affects the programming model. The SPARC definition does not recommend any implementation, rather it defines a class of implementations. Hence it is crucial to design an implementation and verify that it satisfies the given specification.

In the next section we present a brief summary of CCS and the CWB while in sections 3 and 4 we describe the features and simplifications of the architecture, the specification of

the architecture in CCS and the tests performed on it.

2 Overview of CCS and CWB

In this section we present a brief summary of the concepts and notation used in this paper. The reader is referred to [Mil89] and [CPS89] for details. A set of atomic actions (Λ) with a bijection $\bar{\cdot}$ on it such that for all $a \in \Lambda$, $\bar{\bar{a}} = a$ is assumed. A special action τ which indicates synchronisation is used. The syntactic structure of processes is given by the following rules.

$$P := \mathbf{0} \mid \mu{\cdot}P \mid P \mid P \mid P + P \mid P \backslash H \mid P[\phi] \mid X \mid \text{rec } X{:}P$$

0 is a process which can exhibit no further action, $\mu{\cdot}$P can exhibit μ and then behave as P. (P |Q) is the parallel composition of P and Q, (P + Q) represents non-deterministic choice. (P \H) hides all actions specified in H, P[ϕ] relabels all actions in P by ϕ and X and (rec X:P) is used to define recursive processes. A recursive process can also be written as (X = P) which permits the specification of a system using a set of equations. An operational semantics for the processes based on labelled transition system is defined [Mil89].

The principal semantic relation is the notion of (strong) bisimulation [Par81]. Intuitively, P is bisimilar to Q means that every behaviour of P(Q) can be simulated by Q(P). Other semantic relations such as weak bisimulation which is similar to strong bisimulation except that the action τ is internalised, traces which is the automata theoretic characterisation, testing etc. can be defined [Mil89, Hen88]. [vG90] presents a comparative study of various semantics relations.

Some of these semantic relations can be described logically using the modal μ-calculus [HM85, Lar88, Sti89b, Sti89a]. We use the modal μ-calculus to verify that the specifications satisfy certain logical properties. The set of formulae includes action indexed modalities for possibility $\langle\mu\rangle\psi$, universality $[\mu]\psi$, negation $\neg\psi$ and recursion (minimal

fixed point $\mu X.\psi$ and maximal fixed point $\nu X.\psi$.) Formulae can also be combined using the propositional connectives of conjunction, disjunction etc.

Informally a process can satisfy $\langle\mu\rangle\ \psi$ if it can exhibit the action μ and evolve to a process which can satisfy ψ. $[\mu]\psi$ is defined to be $\neg(\langle\mu\rangle\neg\psi)$ and thus can be interpreted to mean that any μ move will necessarily lead to a process which can satisfy ψ. Modalities which ignore τ moves are also defined. For example, the formula $\langle\langle\mu\rangle\rangle\psi$ can be satisfied by a process which after a finite number of τ moves can exhibit μ. The minimal fixed point corresponds to infinite disjunction, while the maximal fixed point is the dual of the minimal fixed point and corresponds to infinite conjunction. Intuitively, the minimal fixed point can be interpreted as a liveness property. If, for example, the proposition (P_0 or P_1 or ...or P_n ...) were satisfied in the nth step, then we can assume that P_0 to P_{n-1} were not satisfied. Therefore, for the property to be satisfied, there should be some n such that P_n is true. By a similar argument the maximal fixed point can be interpreted to be a safety requirement as a proposition of the form (P_0 and P_1 and ...and P_n ...) to be satisfied all of P_i must be satisfied.

The CWB is an automatic tool which helps in the analysis of concurrent systems expressed in CCS. The CWB consists of three main components. The first component handles the user interface where user can define processes and formulae. The user can also issue various other commands to study the behaviour of the specified system. The command `bi` binds an identifier to a process (or an agent) and can be used to define recursion. For example, `bi X P` represents the CCS process 'recX:P' or (X = P). The command `bsi` binds an identifier to a set of actions and is useful when defining restrictions while the command `bpi` binds an identifier to a proposition. The CWB uses `t` as the ascii translation of τ, while `'a` is used instead of $\overline{a}$. The second layer performs certain semantic transformations. While this layer performs a crucial task, the user is completely shielded from it. This makes the tool easier to use than more complex systems. The third layer

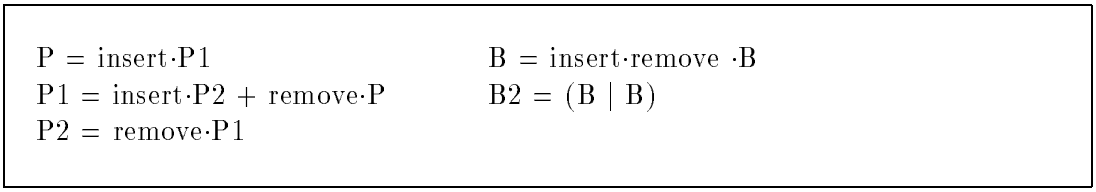

Figure 1: CCS Example

provides the commands for analysing the specification. Having defined agents and modal formulae, the CWB can perform automatic analysis to check if two agents are weakly bisimilar (`eq`), strongly bisimilar (`strongeq`), trace equivalence (`mayeq`), trace preorders (`maypre`). The CWB can also verify if an agent satisfies a logical specification (`cp`). There are other features including examining all behaviours, finding deadlocks etc. which are useful when developing the specifications.

In the next section we present a small example to give a flavour of CCS and the CWB. The expert reader can skip this section and proceed to section 3.

2.1 Example

Consider a buffer of size two which is initially empty. After performing two inserts, remove is the only possible operation on the buffer. If the buffer is empty, only an insertion can be performed. There are two ways to specify the system. The first is to explicitly enumerate the reachable states. As we have not imposed any ordering on insertions and deletions, one could also specify the system as a parallel composition of two one element buffers.

The CCS specification is shown in figure 1.

The CWB specification of the two systems is presented in figure 2.

One may now wish to verify that the two systems P and B2 are equivalent. This can be achieved by the command `eq` B2 P. In this case the CWB returns true. As P and B are not equivalent, the command `eq` P B returns false. However, any behaviour exhibited by a one buffer can be exhibited by a two buffer. This can be checked on the CWB using the command `maypre` B P which yields true. In other words, every trace exhibited by B

```
bi P
  insert.P1

bi P1
  insert.P2 + remove.P

bi P2
  remove.P1

bi B
   insert.remove.B

bi B2
  (B | B)
```

Figure 2: Specification Using the CWB Syntax

can be exhibited by P.

We now present two small examples of modal formulae. The first property we consider is that after two inserts a remove must be performed. This can be restated as after two inserts it is not possible to perform any action but remove. This is described in the modal μ-calculus as $< insert >< insert > [-remove]F$, i.e., it is possible to perform two inserts and then no action other than remove is possible.

The second property will use the maximal fixed point construct. If it possible to insert into a buffer, then it is always possible to perform the insertion followed by a remove. It is intuitively obvious that the property is true. To translate the above property into a formula modal μ-formula, we notice that the property is always true. Hence it indicates that we have to use the maximal fix-point. Given that, the formula can be written in the CWB syntax as follows.

$bpi \;\; Prop \;\; max(X.\,(CanInsert => CanIR)\&[-]X)$

$bpi \;\; CanInsert \;\; < insert > T$

$bpi \;\; CanIR \;\; < insert >< remove > T$

The validity of the property can be verified by the commands $cp \;\; P \;\; Prop$, $cp \;\; B \;\; Prop$

and $cp \quad B2 \quad Prop$ all of which return true. This concludes our brief introduction to the CWB. A more detailed example can be found in [CPS93][pages 58-66].

In the next two sections we describe the specification and the testing performed. The two main features of the SPARC architecture that involves parallelism are instruction pipelining and a memory model that supports multiprocessor operations. In this paper we consider both these aspects. For the sake of readability we use the CCS syntax for all elements except the minimal and maximal fixed point. All specifications in the CWB syntax are available from the author. Section 3 describes the modelling of instruction pipelining and the delayed instruction while section 4 describes the memory pipelining model. While both the models specify pipelining, the effects are different with instruction pipelining being simpler than the memory model.

3 A Simplified Instruction Pipelining in SPARC

While instruction pipelining is not very new, the design of an architecture where the instruction pipelining is visible at the program level is relatively modern. It has been made popular mainly by the RISC architectures. We only provide a brief explanation of this feature. The reader is referred to [Spa91] for more details.

In addition to the program counter (PC) the SPARC has an nPC which points to the next instruction. It is usually PC+4 except in the case of branch instructions. The SPARC provides two types of branch instructions, viz., normal branch and annulled branches. After executing the normal branch instruction, the instruction pointed to by the nPC is executed. In the case of annulled instructions the instruction pointed to by nPC is not executed. A simplified view is explained using the following table.

PC	Instruction
8	Non-branch
12	Branch to 40 (execute delay)
16	Non-branch
...	...
40	Instruction

The instruction sequence executed will be 8, 12, 16, 40 .... If the instruction at address 12 annulled the delayed instruction, the sequence will be 8, 12, 40 .... The formal specification and testing is defined in the next section.

3.1 Specification of Delayed Instructions

Towards modelling the instruction pipeline, we make the following simplifications. In this work we do not consider the complete generality of the SPARC branch instructions. We assume that a branch instruction is denoted by the action *branch*. As annulling of delayed instruction can depend on whether a transfer of control occurs, an internal choice of τ or *signal_annul* is used. As modelling value passing results in an infinite (or very large) space process we also do not model different addresses and hence branching to different locations.

We model the PC and the nPC as buffers of size 1. As PC and nPC represent a pipeline, elements are inserted into nPC (*insert*) and removed from PC ($'remove$) with *getfromnpc* used to transfer an instruction from nPC to PC. The processor (CPU) fetches an instruction from the PC and indicates to the environment that it did so via the action $fetch$, performs a *decode* and continues or treats the instruction as a *branch*. The unit handling control transfer instructions either executes the next (and hence delay) instruction or signals an annulment (*signal_annul*), removes the next instruction (does not execute it) and continues. Note that we need the actions $fetch$ and *signal_annul* to indi-

$$PC = getfromnpc \cdot \overline{remove} \cdot PC$$

$$NPC = insert \cdot \overline{getfromnpc} \cdot NPC$$

$$CPU = remove \cdot fetch \cdot decode \cdot (\tau \cdot \overline{branch} \cdot Continue + \tau \cdot CPU)$$

$$Branch = branch \cdot (\tau \cdot \overline{continue} \cdot Branch + \tau \cdot signal\_annul \cdot \overline{annul} \cdot Branch)$$

$$Continue = continue \cdot CPU + annul \cdot remove \cdot fetch \cdot CPU$$

$$Sys = (PC \mid NPC \mid CPU \mid Branch) \backslash \{annul\ continue\ remove\ branch\ getfromnpc\}$$

Figure 3: SPARC Processor-1

cate the behaviour of the system to the environment. Otherwise the system will collapse to an infinite sequence of τ moves.

The CCS specification used in the CWB is given in figure 3. Sometimes, it is useful to construct a diagrammatic representation of the a CCS specification. The finite state representation of the process CPU and $Continue$ (the states are indicated in the diagram) is given in figure 4. While the diagram may help clarify the behaviour, we do not present them for the sake of brevity.

It is also possible to model the pipelining by a process (called $Pipeline$) which is FIFO buffer of size 2 [Mil89] as shown in figure 5. This is similar to the example considered earlier.

One can check that the definitions involving the process Pipeline or the process PC in conjunction with the process NPC are equivalent. That is, Sys and New_Sys are weakly bisimilar. They are not strongly bisimilar due to the synchronisation between PC and NPC.

The intuitive property the system should satisfy is that after a $signal\_annul$ and a $fetch$, the action $decode$ cannot be exhibited. This is because the processor has to discard the instruction just fetched simulating annulling. This can be expressed in the modal-μ-calculus as shown in figure 6.

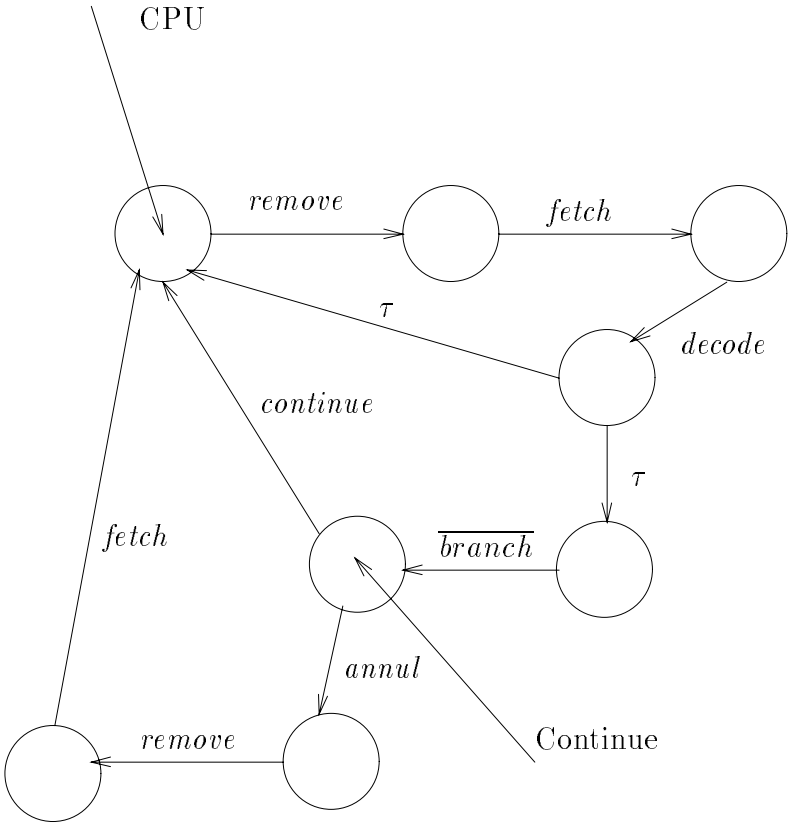

Figure 4: State Diagram

$$Pipeline = insert \cdot P_1$$

$$P_1 = insert \cdot Full + \overline{remove} \cdot Pipeline$$

$$Full = \overline{remove} \cdot P_1$$

$$NewSys = (Pipeline \mid CPU \mid Branch)\backslash \{annul\ continue\ remove\ branch\}$$

Figure 5: Pipeline as a Buffer

$$Delay = max(X.(Poss => Required)\&[-]X)$$

$$Poss =< signal\_annul >< fetch >> T$$

$$Required =< signal\_annul >< fetch >> [decode]F$$

Figure 6: Modal Formula for Delayed Instruction

The intuitive explanation of the formula is as follows. If it is possible to exhibit $signal\_annul$ followed by $fetch$ (i.e., the formula $Poss$), the required behaviour must be observed, i.e., cannot exhibit the action $decode$. The formula $Required$ specifies this by $[decode]F$ which requires that $decode$ is impossible. Note that we use the maximal fixed point operator as the specification $Delay$ is a safety property; i.e., has to be satisfied by *every* execution. The CWB verifies that the process Sys satisfies the formula $Delay$.

In this work we do not consider different types of instructions and assume that there is one action which represents an instruction. The difference between a control transfer instruction (annulling or executing the delay) is modelled as an internal choice. This concludes our discussion of instruction pipelining. In the next section we consider the two main memory models supported by the SPARC architecture.

4 A Simplified SPARC Memory Model

The definition of the SPARC memory model is applicable to both uniprocessor and shared memory multiprocessors. The memory model relates the semantics of the memory operations as issued by a processor and the semantics of the operations as executed by a memory unit. In other words, the model specifies the semantics of data load and store and the relation between the order in which a processor issues the the instructions and the order in which a central memory executes them. It also defines how instruction fetches are synchronised with memory operations.

In this work we consider a simplified model of the total store ordering (TSO) and the partial store ordering (PSO). Both these models only specify the behaviour observed by the software and hence is a good candidate to be modelled on the CWB. For the purposes of the model, a processor consists of a unit which issues loads and stores to the processor's memory port. This order is called the processor's issuing order. The memory executes the instructions of all the processors in an order called the memory order. The TSO model

guarantees that the sequence of operations executed by the memory is identical to one issued by a processor. Hence as far as the processor is concerned, the memory is a FIFO structure. In the PSO model the order in which the memory executes the operations could be different from the order in which a processor issued them. Hence the buffer is not guaranteed to be a FIFO structure. It is possible to maintain a relationship between the issuing order and the execution order using the *stbar* instruction. *stbar* instruction ensures that any memory operation issued by a processor before a *stbar* are executed before the operations issued after the *stbar*. Hence the *stbar* instruction partitions the processors issuing sequence into non-FIFO classes but the partition themselves are ordered. Consider for example a single processor issuing the instructions $i1$,$i2$,$i3$. In the TSO model, the memory will necessarily execute $i1$ followed by $i2$ followed by $i3$. However, in the the PSO model the memory could execute $i2$ followed by $i3$ followed by $i1$. If the sequence were $i1$, $i2$, *stbar* $i3$, the memory could execute $i1$ and $i2$ in any order but will execute $i3$ only after $i1$ and $i2$. Hence a limited form of FIFO behaviour is exhibited. Clearly, $i1$, *stbar*, $i2$, *stbar*, $i3$ will be executed in FIFO order. More details can be found in [Spa91][pages 59-68].

The formal specification and testing is given below.

4.1 Memory Model

In order to make the automate the process of verification, we consider a few more simplifications. The modelling of values and addresses results in a large state space (which makes automatic verification extremely time consuming) due to which we do not consider them. This restriction can be removed easily by representing addresses using nondeterminism. See [Mil89] where values are simulated by choice. Therefore, we also assume that a load does not look at the buffer to see if an appropriate store has been issued before. The the two cases of load returned without a memory operation and with a memory

operation can also be modelled as non-deterministic choice of two processes. To simply this exposition further we do not consider the $flush$ instruction. Therefore, in this paper we consider only the *store*, *load* and the *stbar* instructions.

If we were to consider a general specification of the memory model, a infinite state space process is necessary. In other words, we have to assume an unbounded memory system. As this is not practical we consider a fixed-finite buffer size. The system we model consists of a store buffer of size 3.

It appears to be very difficult to specify the buffer succinctly. The main reason seems to be the lack of a general sequencing operator as in ACP [BK88]. Furthermore, the behaviour of the buffer requires it to be history sensitive, i.e., it has to 'remember' the items inserted into it before a *stbar* instruction was executed and to distinguish the various instructions separated by *stbar*s.

Our specification is by enumeration, i.e., each possible state that the buffer could be in is explicitly listed. For example, P_{sstbl} indicates a state where a load followed by a *stbar*, followed by a store was issued. Thus it represents the minimal state machine. The behavioral specification *implicitly* removes the *stb* instructions when the last instruction before the *stbe* is removed. For example, P_{sstbl} evolves to P_s after the load instruction is removed. We use the actions $load\_insert$, $store\_insert$ and *stb* to indicate the interaction between a processor and the buffer while the actions $'load\_remove$ and $'store\_remove$ indicate the removal of the items from the buffer by the single-ported memory. The complete specification of 3 element buffer in the PSO model is given by $POBuff$ in figures 7 and 8.

Once again we enumerate each state the buffer can be in and hence is minimal. As in the PSO case the *stb* instruction is removed implicitly. The specification of the sequential buffer of size 3 is presented in figures 9 and 10.

Now we describe the tests performed on the two specifications to gain confidence

$$POBuf = load\_insert \cdot P_l + store\_insert \cdot P_s$$

$$P_l = load\_insert \cdot P_{ll} + store\_insert \cdot P_{ls} + stb \cdot P_{stbl} + \overline{load\_remove} \cdot POBuf$$

$$P_s = load\_insert \cdot P_{ls} + store\_insert \cdot P_{ss} + stb \cdot P_{stbs} + \overline{store\_remove} \cdot POBuf$$

$$P_{ll} = load\_insert \cdot P_{lll} + store\_insert \cdot P_{lls} + stb \cdot P_{stbll} + \overline{load\_remove} \cdot P_l$$

$$P_{ss} = load\_insert \cdot P_{lss} + store\_insert \cdot P_{sss} + stb \cdot P_{stbss} + \overline{store\_remove} \cdot P_s$$

$$P_{ls} = load\_insert \cdot P_{lls} + store\_insert \cdot P_{lss} + stb \cdot P_{stbls} + \overline{load\_remove} \cdot P_s + \overline{store\_remove} \cdot P_l$$

$$P_{stbl} = \overline{load\_remove} \cdot POBuf + load\_insert \cdot P_{lstbl} + store\_insert \cdot P_{sstbl}$$

$$P_{stbs} = \overline{store\_remove} \cdot POBuf + load\_insert \cdot P_{lstbs} + store\_insert \cdot P_{sstbs}$$

Figure 7: PSO-1

$$P_{lls} = \overline{load\_remove} \cdot P_{ls} + \overline{store\_remove} \cdot P_{ll}$$

$$P_{lss} = \overline{store\_remove} \cdot P_{ls} + \overline{load\_remove} \cdot P_{ss}$$

$$P_{lll} = \overline{load\_remove} \cdot P_{ll}$$

$$P_{sss} = \overline{store\_remove} \cdot P_{ss}$$

$$P_{stbss} = \overline{store\_remove} \cdot P_{stbs}$$

$$P_{stbll} = \overline{load\_remove} \cdot P_{stbl}$$

$$P_{lstbs} = \overline{store\_remove} \cdot P_l$$

$$P_{sstbs} = \overline{store\_remove} \cdot P_s$$

$$P_{lstbl} = \overline{load\_remove} \cdot P_l$$

$$P_{sstbl} = \overline{load\_remove} \cdot P_s$$

$$P_{stbls} = \overline{load\_remove} \cdot P_{stbs} + \overline{store\_remove} \cdot P_{stbl}$$

Figure 8: PSO-2

$$SeqBuff = load\_insert \cdot SeqBuff_l + store\_insert \cdot SeqBuff_s$$

$$SeqBuff_l = \overline{load\_remove} \cdot SeqBuff + store\_insert \cdot SeqBuff_{sl} + load\_insert \cdot SeqBuff_{ll} + stb \cdot SeqBuff_{stbl}$$

$$SeqBuff_s = \overline{store\_remove} \cdot SeqBuff + store\_insert \cdot SeqBuff_{ss} + load\_insert \cdot SeqBuff_{ls} + stb \cdot SeqBuff_{stbs}$$

$$SeqBuff_{sl} = \overline{load\_remove} \cdot SeqBuff_s + store\_insert \cdot SeqBuff_{ssl} + stb \cdot SeqBuff_{stbsl} + load\_insert \cdot SeqBuff_{lsl}$$

$$SeqBuff_{ls} = \overline{store\_remove} \cdot SeqBuff_l + store\_insert \cdot SeqBuff_{sls} + stb \cdot SeqBuff_{stbls} + load\_insert \cdot SeqBuff_{lls}$$

$$SeqBuff_{ll} = \overline{load\_remove} \cdot SeqBuff_l + load\_insert \cdot SeqBuff_{lll} + store\_insert \cdot SeqBuff_{sll} + stb \cdot SeqBuff_{stbll}$$

$$SeqBuff_{ss} = \overline{store\_remove} \cdot SeqBuff_s + load\_insert \cdot SeqBuff_{lss} + store\_insert \cdot SeqBuff_{sss} + stb \cdot SeqBuff_{stbss}$$

$$SeqBuff_{stbl} = \overline{load\_remove} \cdot SeqBuff + load\_insert \cdot SeqBuff_{lstbl} + store\_insert \cdot SeqBuff_{sstbl}$$

$$SeqBuff_{stbs} = \overline{store\_remove} \cdot SeqBuff + load\_insert \cdot SeqBuff_{lstbs} + store\_insert \cdot SeqBuff_{sstbs}$$

Figure 9: TSO-1

$$SeqBuff_{lll} = \overline{load\_remove} \cdot SeqBuff_{ll}$$

$$SeqBuff_{sss} = \overline{store\_remove} \cdot SeqBuff_{ss}$$

$$SeqBuff_{sll} = \overline{load\_remove} \cdot SeqBuff_{sl}$$

$$SeqBuff_{lls} = \overline{store\_remove} \cdot SeqBuff_{ll}$$

$$SeqBuff_{ssl} = \overline{load\_remove} \cdot SeqBuff_{ss}$$

$$SeqBuff_{lsl} = \overline{load\_remove} \cdot SeqBuff_{ls}$$

$$SeqBuff_{lss} = \overline{store\_remove} \cdot SeqBuff_{ls}$$

$$SeqBuff_{sls} = \overline{store\_remove} \cdot SeqBuff_{sl}$$

$$SeqBuff_{stbsl} = \overline{load\_remove} \cdot SeqBuff_{stbs}$$

$$SeqBuff_{stbll} = \overline{load\_remove} \cdot SeqBuff_{stbl}$$

$$SeqBuff_{stbls} = \overline{store\_remove} \cdot SeqBuff_{stbl}$$

$$SeqBuff_{stbss} = \overline{store\_remove} \cdot SeqBuff_{stbs}$$

$$SeqBuff_{lstbs} = \overline{store\_remove} \cdot SeqBuff_{l}$$

$$SeqBuff_{lstbl} = \overline{load\_remove} \cdot SeqBuff_{l}$$

$$SeqBuff_{sstbl} = \overline{load\_remove} \cdot SeqBuff_{s}$$

$$SeqBuff_{sstbs} = \overline{store\_remove} \cdot SeqBuff_{s}$$

Figure 10: TSO-2

$$Producer1 = store \cdot \overline{store\_insert} \cdot P11 + load \cdot \overline{load\_insert} \cdot P11$$

$$P11 = stbar \cdot \overline{stb} \cdot Producer1 + \tau \cdot Producer1$$

$$Producer2 = store \cdot \overline{store\_insert} \cdot P21 + load \cdot \overline{load\_insert} \cdot P21$$

$$P21 = \overline{stb} \cdot Producer2 + \tau \cdot Producer2$$

$$PSO = (Producer1 \mid POBuf) \setminus \{stb\ load\_insert\ store\_insert\}$$

$$TSO = (Producer1 \mid SeqBuff) \setminus \{stb\ load\_insert\ store\_insert\}$$

Figure 11: Specification for Testing the Architecture

that our definition satisfies the requirements imposed on the two models. Towards that we define processes which generate a sequence of *load*s *store*s and *stbar*'s. Again each operation is split into two actions, one for the visible part and the other for the internal synchronisation (e.g., $\overline{load\_insert}$.) Define two environments TSO and PSO are systems constructed using the TSO buffer and the PSO buffer respectively. The specification of the above is shown in figure 11. The main difference between $Producer1$ and $Producer2$ is that in $Producer2$ the issuing of *stbar* instruction is not visible.

The CWB verifies that PSO and TSO are not weakly bisimilar or even trace equivalent. This is to be expected as in the PSO model the execution of stores and loads can be different from the issuing order. However TSO is less than in the trace preorder than PSO. Thus every trace that can be exhibited by TSO can be exhibited by PSO. Therefore the specification of the sequential buffer is not inconsistent with the partial order buffer.

To ensure that the difference between TSO and PSO is indeed due to the *stbar* instruction, we verify that PSO satisfies the modal-formula $Cando$ in figure 12 which TSO cannot satisfy. The formula $Cando$ states that after a *load* and a *store*, the memory is able to execute the store operation. The dual of $Cando$ is the formula $Ordering$ which requires that after a sequence of *load* and *store* actions it is not possible to execute a $\overline{store\_remove}$.

$$Cando = << load >><< store >><< \overline{store\_remove} >> T$$

$$Ordering = << load >><< store >> [[\overline{store\_remove}]]F$$

$$STBF = max(X.\ (LoadPossible \mid [-load]X)\ \&[-]X)$$

$$LoadPossible = (<< load >><< stbar >><< store >> [[\overline{store\_remove}]]F$$

Figure 12: Modal Formulae Distinguishing TSO and PSO

PSO also satisfies the requirement that *stbar* ensures issuing order as it will satisfy the formula $STBF$ in figure 12. The intuitive meaning of $STBF$ is that if a *stbar* is issued after a *load* and before a *store*, it is not possible to execute the store operation. As this is a safety requirement, we use the maximal fixed point operator. To understand this more formally, we consider two main possibilities; viz., it is possible to perform a *load* followed by *stbar* and *store* or it is not. If it is not possible to perform the specified sequence all subsequent behaviours continue to satisfy $STBF$. Otherwise performing the sequence of actions will result in arriving at a state where it is not possible to perform $\overline{store\_remove}$. This is stated by the formula $LoadPossible$. In other words, $STBF$ is of the form $max(X.\ (P \mid Q)\ [-]X)$ where the formula corresponding to P states if a *load* and *store* are separated by *stbar* then $\overline{store\_remove}$ cannot be observed and the formula corresponding to Q states that if *load* is not possible the formula $STBF$ is satisfied in the future.

The above tests distinguished $POBuff$ and $SeqBuff$. We now identify conditions under which the two systems are equivalent. The first condition we consider is a processes which separates every load/store with a *stbar* instruction. By the definition of the effect of *stbar* on the PSO model, it is clear that the PSO model collapses to the TSO model. We also show that executing the *stbar* instruction in the TSO model has no effect. The specification of these tests are presented in figure 13.

$SeqProd$ ensures that after every *load* or *store* a *stbar* is issued. This process does not

$$SeqProd = load \cdot \overline{load\_insert} \cdot stbar \cdot \overline{stb} \cdot SeqProd + store \cdot \overline{store\_insert} \cdot stbar \cdot \overline{stb} \cdot SeqProd$$

$$Sys1 = (SeqBuff \mid SeqProd) \backslash \{stb\ load\_insert\ store\_insert\}$$

$$Sys2 = (POBuf \mid SeqProd) \backslash \{stb\ load\_insert\ store\_insert\}$$

$$Sys3 = (Producer2 \mid SeqBuff) \backslash \{stb\ load\_insert\ store\_insert\}$$

$$SProd = load \cdot \overline{load\_insert} \cdot SProd + store \cdot \overline{store\_insert} \cdot SProd$$

$$Sys4 = (SProd \mid SeqBuff) \backslash \{stbar\ stb\ load\_insert\ store\_insert\}$$

Figure 13: Equivalence Testing

use the non-ordered access of the PSO-store. Thus $Sys1$ and $Sys2$ are weakly bisimilar, i.e., the underlying memory model is of no consequence for $SeqProd$.

$Sys3$ and $Sys4$ are trace equivalent thus showing that *stbar* is a no-op in the TSO-store model. $Sys3$ and $Sys4$ are not bisimilar due to the presence of *stb* in the buffers. Note that $Producer2$ was essential as otherwise the observational actions become different.

5 Lessons Learned

In this paper we have shown the feasibility of specifying and testing concurrent aspects of an architecture. The type of analysis performed on the various specifications has been inspired by both the informal description of the various features and the formal description (using first order logic) given in the SPARC manual [Spa91]. The principal observational properties have been verified here. In verifying the system we have generated formulae which we believe were relevant.

The CWB has the capability of generating formulae which distinguish non-equivalent processes. The command `dfobs` of processes P and Q generates a formula ignoring τ which

is satisfied by P and not by Q. Similarly the command `dfstr` generates a formula where the τ actions are accounted from which is satisfied by P and not by Q while the command `dfmay` generates a trace exhibited by one but not the other.

The command `dfstr` $TSO\ PSO$ generates the formula

$$< store >< t >< t >< load >< t > [\overline{load\_remove}]F$$

while the command `dfobs` $PSO\ TSO$ generates the formula

$$<< load >>\ [[store]]\ [[load]]\ [[load]]\ << \overline{store\_remove} >>$$

These formulae capture the non-FIFO behaviour of the PSO model while requiring the FIFO behaviour of TSO model. Similarly, the command `dfmay` $PSO\ TSO$ generates the string $store, load, \overline{load\_remove}$ which can be performed by PSO but not by TSO. We have some confidence in our specifications as the CWB agrees with our observations. The CWB does not generate formulae involving fix points because a finite formula suffices to distinguish two processes.

As the SPARC architecture is specified formally, it may be possible to prove some completeness result. However such a result is beyond the scope of this paper. We hope that one will be prove that all properties specified by the first order logic specification has been covered by the modal-μ calculus specifications.

Using a completely automatic tool has its limitations. Features such as the TSO and PSO buffers had to be enumerated and hence were not elegant specifications. It also makes it difficult to generate a buffer of size $n+1$ from a buffer of size n. Consider, for example, figure 14 where a PSO buffer of size two is specified. It is easy to check that Two is less than $POBuf$ in the trace preorder-order. However, it is difficult to see how Two can be expanded to obtain $POBuf$. As $POBuf$ can be perceived to be an extension of Two one may assume that Two in parallel conjunction with another process (subject to appropriate synchronisations) can be used to obtain $POBuf$. The CWB supports a feature for equation solving which, initially, appears to be attractive. Given processes A,

$$T = si \cdot Ts + li \cdot Tl$$

$$Ts = si \cdot Tss + li \cdot Tls + stb \cdot Tsts$$

$$Tl = si \cdot Tls + li \cdot Tll + stb \cdot Tstl$$

$$Tss = \overline{sr} \cdot Ts$$

$$Tll = \overline{lr} \cdot Tl$$

$$Tstl = \overline{lr} \cdot T$$

$$Tsts = \overline{sr} \cdot T$$

$$Tls = \overline{sr} \cdot Tl + \overline{lr} \cdot Ts$$

$$Two = T[store\_insert/si, load\_insert/li, load\_remove/lr, store\_remove/sr]$$

Figure 14: PSO Buffer

B and a synchronisation set L the system finds an X such that ((A | X) \L) is bisimilar to B. This feature turn out not to be useful as $(Two \mid X \setminus L) \sim POBuf$ cannot be solved easily. Clearly the set L cannot be empty as interaction between Two and the unknown X is essential. As the equation solving system requires the user to specify L, the above equation cannot be solved.

In this paper, we have modelled a small system. As most of the algorithms to check bisimilarity, trace equivalence etc. are exponential [KS90], an automatic verifier cannot be used for large systems. But this technique is useful in studying synchronisation patterns in small systems and performing compositional verification semi-automatically.

The SPARC manual [Spa91] provides a formal definition of the memory models. As the specification is logical rather than behavioural, our specification can be considered an behavioural representation of the model. However, one needs a system where the behavioural representation can be verified against the logical specification. We believe that a system like HOL would be very useful. The technique to specify CCS in HOL has been described in [CIN91]. It remains to be seen if this technique can be adapted for our

system.

Other features such as the *flush* and *ldstub* instructions can be added to the basic specification described here. Modelling the *flush* instruction requires the specification of an instruction load and associated buffers which behave similar to the PSO model. The *dstub* blocks the processor and can be modelled by requiring a handshake (synchronisation) between the memory and the processor. All these features can be modelled individually; however a combined specification appears to be too large to run on the CWB. This indicates that a prototype implementation while satisfactory for small examples, is not sufficient for large examples. In conclusion, our work shows that the CWB is useful in studying synchronisation in the initial phases of design.

Acknowledgements

The author acknowledges the many helpful comments from a referee. This research has been partially supported by University of Canterbury Grant No 1787123.

References

[BB89] T. Bolognesi and E. Brinksma. Introduction to the ISO Specification Language LOTOS. In P. H. J. van Eijk, C. A. Vissers, and M. Diaz, editors, *The Formal Description Technique LOTOS*, pages 23–73. North Holland, 1989.

[BK88] J. A. Bergstra and J. W. Klop. Process Theory Based on Bisimulation Semantics. In *Linear Time, Branching Time and Partial Order in Logics and Models for Concurrency, LNCS 354*, pages 50–122. Springer Verlag, 1988.

[CIN91] A. Camilleri, P. Inverardi, and M. Nesi. Combining Interaction and Automation in Process Algebra Verification. In S. Abramsky and T. S. E. Maibaum, editors, *TAPSOFT-91: LNCS 494*, pages 283–296. Springer Verlag, 1991.

[Coh88] A. J. Cohn. A Proof of Correctness of the Viper Microprocessor: The First Level. In G. Birtwistle and P. A. Subrahmanyam, editors, *VLSI Specification, Verification and Synthesis*. Kluwer Academic Press, 1988.

[CPS89] R. Cleaveland, J. Parrow, and B. Steffen. The concurrency workbench. In *Proceedings of the Workshop in Automatic Verification Methods for Finite-State Systems: LNCS 407*, pages 24–37. Springer Verlag, 1989.

[CPS93] R. Cleaveland, J. Parrow, and B. Steffen. The Concurrency Work Bench: A Semantics Based Tool for the Verification of Concurrent Systems. *ACM Trans. on Programming Languages and Systems*, 15(1):36–72, January 1993.

[GB89] B. Graham and G. Birtwistle. Formalising the Design of an SECD Chip. In M. Leeser and G. Brown, editors, *Hardware Specification, Verification and Synthesis: Mathematical Aspects:LNCS 408*, pages 40–66. Springer Verlag, 1989.

[Gor85] M. Gordon. HOL: A Machine Oriented Formulation of Higher Order Logic. Technical Report 68, University of Cambridge, Computer Laboratory, 1985.

[Hen88] M. C. B. Hennessy. *Algebraic Theory of Processes*. MIT Press, 1988.

[HM85] M. Hennessy and R. Milner. Algebraic laws for nondeterminism and concurrency. *Journal of the Association of the Computing Machinery*, 32(1):137–161, January 1985.

[KS90] P. C. Kannelakis and S. A. Smolka. CCS expressions, finite state processes and three problems of equivalence. *Information and Computation*, 86(1), May 1990.

[Lar88] K. G. Larsen. Proof systems for Hennessy-Milner logic with recursion. In *13th Colloquim on Trees in Algebra and Programming*. Springer Verlag, 1988.

[LD90] P. Loewenstein and D. L. Dill. Verification of a Multiprocessor Cache Protocol Using Simulation Relations and Higher-Order Logic. In E. M. Clarke and

R. P. Krushan, editors, *Computer Aided Verification: LNCS 531*, pages 302–311. Springer Verlag, 1990.

[LM87] K. G. Larsen and R. Milner. Verifying a Protocol Using Relativized Bisimulation. In *ICALP -87, LNCS 267*. Springer Verlag, 1987.

[Mil89] R. Milner. *Communication and Concurrency*. Prentice Hall International, 1989.

[Par81] D. Park. Concurrency and Automata on Infinite Sequences. In *Proceedings of the 5th GI Conference, LNCS-104*. Springer Verlag, 1981.

[Par87] J. Parrow. Verifying a CSMA/CD-Protocol with CCS. Technical Report ECS-LFCS-87-18, Computer Science Department, University of Edinburgh, 1987.

[Spa91] Sparc International. *The SPARC Architecture Manual: Version 8*, 1991.

[Sti89a] C. Stirling. An Introduction to Modal and Temporal Logics for CCS. In *Joint UK/Japan Workshop on Concurrency:LNCS 491*, pages 2–20, 1989.

[Sti89b] C. Stirling. Temporal Logics for CCS. In *Linear Time, Branching Time and Partial Order in Logics and Models for Concurrency, LNCS 354*. Springer Verlag, 1989.

[vG90] R. van Glabbeek. The Linear Time-Branching Time Spectrum. In J. C. M. Baeten and J. W. Klop, editors, *CONCUR 90, LNCS-458*. Springer Verlag, 1990.

[vS89] J. van de Lagemaat and G. Scollo. On the Use of LOTOS for the Formal Description of a Transport Protocol. In K. J. Turner, editor, *Formal Description Techniques*. North Holland, 1989.

[vVD89] P. H. J. van Eijk, C. A. Vissers, and M. Diaz. *The Formal Description Technique LOTOS*. North Holland, 1989.