Demos2k as a Bridge to Formal Methods

Richard Taylor and Chris Tofts
HP Laboratories
HPL-2008-184

Keyword(s):
Concurrency, Multi Core, Parallelism, Formal, Analysis, Simulation, Visualisation

Abstract:
With multi core computing systems becoming commonplace it has been recognised that their full potential can only be exploited by using explicitly parallel programming solutions. It has long been known that this class of programs is difficult to constrcut correctly, difficult to demonstrate efficient and a major problem to maintain. Although formal approaches to these programming situations have demonstrated many point successes, their widespread adoption has been limited by the (reasonable) belief that they are too technically challenging and time consuming to deliver value to a project. In this paper we illustrate, via an extended example, how a formally based process based simulation language (DEMOS2k) provides a mid point between concept, implementation and formal analysis. Furthermore, regarded as a rapid software prototyping system it gives a rapid visualisation of how the algorithm concept will execute in a concurrent setting.

External Posting Date: October 21, 2008 [Fulltext] - Approved for External Publication

Internal Posting Date: October 21, 2008 [Fulltext]

DEMOS2K a Bridge to Formal Systems Analysis

Richard Taylor & Chris Tofts
HP Laboratories Bristol,
{richard.taylor, chris.tofts}@hp.com

October 13, 2008

1 Introduction

The prospect of large number multiple core processors and the need to exploit their computational power to the full, has re-ignited interest in the construction of explicitly parallel[?] coded solutions. It has been observed that to fully exploit the power of these processors we will need to change the way in which we write programs for them. Equally, it has long been understood that writing parallel programs is a demanding and error prone task, with the cause of the errors often being difficult to trace and remove.

On the other side there has long been a view that 'formal' analysis of programs can help reduce the number of errors and potentially improve productivity in complex programming settings. However, widespread adoption of these methods has been extremely limited. Usually they are regarded as fit only for situations solely where there is an immediate threat to life if the constructed system fails to perform adequately. The lack of widespread adoption of formal techniques can be attributed to many potential factors:

- the skill level (expense/rarity) of the staff required to deploy them;
- the time 'wasted' in constructing the formal model, could have been coding;
- the 'lack' of detail in the formal model, too abstract – 'not the system';
- the need to abstract into the formal system to get an effective result;
- the use of raw mathematical forms which do not provide the kinds of development support environment programmers are used to;
- the difficulty of debugging the models themselves[1]
- given the 'linguistic' gap between implementation and formal models, believing that the results of the formal analysis apply to the problem at hand;
- identifying what questions should (or could) be (effectively) asked within the formal analysis.

The problems identified above span the major problems in comprehending computational systems. One the one hand these are inherently formal objects at the level of presentation – almost all systems are provided (code) in a syntactically formal system. On the other hand attempting to understand the system at that level of detail is often unnecessary – we need to abstract. The presence of an ultimately formal basis leads to a belief that there is neither a need or any value in abstraction in the setting of analysing computational systems, which stands in contrast to the conventional approaches to understanding the complexity of algorithms!

To bridge this contradiction the programmer needs a solution representation system with several important properties:

1. it encourages the programmer to abstract to the 'core' of the problem;
2. whilst giving them confidence that they have captured it.

[1]One of the hardest problems is knowing that a formal model 'correctly' captures the system problem under consideration.

3. it gives them an effective method of model ‘debugging’;
4. it allows them to observe dynamic properties of their system (as a model);
5. it permits them to ‘see’ the kind of questions of the system they should ask.

To this end we developed the DEMOS2K system – the basic concept was to have a simulation frame which itself was well defined[2] and lends itself to the construction of models of the programming problem, rather than implementations. It is interesting to note that the construction of detailed simulations is itself as hard a project as constructing a full implementation, and therefore is best deferred to the point where the proposed solution is sufficiently well understood, and justified, as to support the effort.

Throughout the rest of this paper we will provide a brief introduction to the language DEMOS2K, and present a motivating example, a COMA style concurrent memory management protocol system. We illustrate how that system can be implemented within DEMOS2K, and that the conceptual gap between the DEMOS2K implementation and a formal analysis is sufficiently small to be a much smaller challenge than proceeding directly from a detailed implementation. Finally we illustrate how the COMA model can be moved into a ‘full’ formal analysis and the nature of the information which can be derived in that setting.

2 DEMOS2K

DEMOS2K is a direct descendant of Demos [?] and is a process oriented discrete event simulation language which was initially defined as an extension to the Simula [?] language. The new version [?] has little changed syntactically from the original other than in how entities slave themselves to others. Demos has essentially 3 components.

1. entities, defined as instances of classes;
2. resources;
3. bins.

These elements all have definitional forms:

1. entity(name,className,offfset) and class name=body
2. res(foo,amt);
3. bin(foo,amt);.

Entities are the active elements of the system, resources can be thought of a semaphores and bins as points of asynchrony. Entities interact with other elements of the system in the following manner:

1. getR(resN,amt) and putR(resN,amt) respectively claim and free the amt of resource resN, an entity cannot return amounts of resource it does not own;
2. getB(binN,amt) and putB(binN,amt) respectively claim and free the amt of bin binN;
3. sync(name) slaves the current entity on name;
4. getS(syn,amt) and putS(syn,amt) respectively claim and free the amt of sync syn, an entity cannot return amounts of sync it does not own.
5. Entity(name,className,offset) spawns a new instance of className called name at time offset into the future;
6. Hold(ti) advances the simulation clock ti into the future.

[2]This actually means that the system has to have a valid formal semantics, which in practice means it needs two which can be compared for consistency! In this case both operational [?, ?], denotational [?, ?, ?] with the comparison in [?]

Collectively getR(resN,amt), getB(binN,amt) and getS(syn,amt) are refered to as acquisitions. Are we permit the following compound operations on them:

1. req[acq1,...,acqN] requires that all of the requests can be granted simultaneously before the entity can proceed.
2. try [req1] then Bd1 etry [req2] then Bd2 ... etry [reqN] then BdN; will try each of the requests in turn until one can be satisfied, otherwise it blocks until one can be satisfied. Often used with ... etry [] then hold(t) as a non-blocking test.

Finally we allow loops

1. repeat body executes body forever;
2. do n body executes body n times, usually used in setup
3. while [req] body as long as the requirement can be met execute the body. Note a requirement can be a boolean.

A demos program consists of an entity (conventionally referred to as main) that sets up class and other definitions and then invokes some entities to form the running system.

3 An example - COMA protocol

A basic component of the COMA protocol is the maintenance of a list of processors which have a current valid copy of the data, a so-called *validity chain*. One of the processors will be distinguished as a *directory* node, and have the task of maintaining a pointer to the head of this validity chain, and will be the processor to which the remaining *client* processors will initially look for a valid copy of the data. Furthermore, the final element in the validity chain will have a pointer back to the directory node.

In each processor the data can be in one of three states:

- *exclusive*: this processor has the *only* valid copy of the data;
- *shared*: this processor and at least one other have a valid copy of the data;
- *invalid*: this processor does not have a valid copy of the data.

The COMA protocol resides in the memory-management unit that controls the interaction of a processor with its local memory and mediates the exchange of data with the communication bus. Reading the value of some data by some processor proceeds as follows. If the local copy of the data is in the state *shared* or *exclusive* then the reading proceeds unhindered; otherwise, it stalls until a valid copy of the data is obtained.

A valid copy of the data is obtained by issuing a *read request* to the directory. The directory's response to such a request is then dependent on the state of its own data:

- If the directory has a valid copy of the data (that is, it is in the *exclusive* or *shared* state), it sends the data to the requesting processor along with a pointer to the first client in the validity chain, and sets its own status to *shared*; when it receives the data, the requesting processor moves into the *shared* state and points to this next client in the chain.
- If the directory is in the *invalid* state, it forwards the read request to the first client in the validity chain, which in turn passes the data to the requesting process along with a pointer to the *next* processor in the validity chain, marking the requesting processor as the new next element in the chain, and again sets its own status to *shared*; again when it receives the data, the requesting processor moves into the *shared* state and points to this next client in the chain.

Writing a new data value by some processor proceeds in a similar vein. If the local copy of the data is in the state *exclusive* then the writing proceeds unhindered; otherwise it must first obtain exclusive access by issuing a *write request* to the directory. The directory's response to such a request is to send an *invalidation request* along the validity chain, and when the request comes back to the directory, to give the requesting processor exclusive access; the directory sets a pointer to this process as the head of the validity chain, and the process itself points back to the directory.

When a new data value is written by some processor, the local value is assumed to be valid; this is to facilitate a partial updating of the current data. Hence if the local copy of the data is in the state *invalid*, it must request a valid copy of the data from the directory along with the granting of the write privilege. If the directory has a valid copy, invalidation proceeds as described; otherwise the directory actually issues an *invalidate with data request* along the validity chain, which results in a valid copy of the data being returned to the directory from the last processor in the validity chain.

The detailed definitions of the behaviour of the COMA protocol derived from a state-transition system generated from the C code implementing the system simulator [?] is presented in Appendix **??**.

4 COMA in DEMOS2K

Since their is considerable detail in the protocol we shall just present the implementation of one client state in full below. The full protocol in DEMOS2K can be found in Appendix **??**. The client definition is as follows:

```
class ClientS(number,next)=
{local var target=0;
 local var dest=0;
 local var gID=0;
 local var gNext=0;
  trace("%v to %v",number,next);
  try [getvb(invalid,[target],target==number)]
  then {hold(transfer);
        trace("put invalid %v",next);
        putvb(invalid,[next]);
        entity(ClientI,ClientI(#number),0);
       }
  etry [getvb(readForward,[target,dest],target==number)]
  then {hold(transfer);
        trace("ReadForward number %v next %v req %v",number,next,dest);
        putvb(grant,[dest,number]);
        entity(ClientShared,ClientS(#number,#next),0);
       }
  etry [getvb(LocalR,[dest],dest==number)] then
  {entity(ClientShared,ClientS(#number,#next),0); }
  etry [getvb(LocalW,[dest],dest==number)] then
  {putvb(write,[number]);
   getvb(invalid,[target],target==number);
   hold(transfer);
   trace("put invalid %v",next);
   putvb(invalid,[next]);
   getvb(grant,[gID,gNext],gID==number);
   entity(ClientEx,ClientEX(#number),0);
  }
}
```

The client makes a choice between:

1. becoming **invalid**, another process wants to write to the data, in which case it passes the invalidation signal on the the next sharing element, and then becomes invalid;
2. have a **readForward** in which case it grants the request and tells the destination to point at it is the holder of the data;

3. have a **local read**, which it can obviously perform as it has a copy of the data;
4. on a **local write**, it needs to have sole possession of the data, so sends a write request to the directory, it must then wait for the invalidation chain to 'pass through' and then wait for the grant, after the last sharing process has handled the invalidation

This is a fairly direct implementation of the requirements of the transition table. The rest of the definition for the invalid and exclusive states proceeds in the same manner. Similarly, we have a dual definition for the directory process. Notice that we can have as many copies of this entity as we desire, and use them for visualisations of the protocol at work, alongside exploration of its performance.

There still remains the question however, even if the simulation 'runs correctly' with many processes and for long simulated times, is the protocol correct? To answer that question we can translate the state transition definitions into a process algebra such as CCS [?].

5 COMA in CCS

In the case of 1 directory and 2 clients we can expand the DEMOS2K[?] definition to a process algebra one[?], suitable for analysis with the Edinburgh Concurrency Workbench [?], by concretising all of the values. This is a straightforward mechanical operation which could potentially be automated.

We must explicitly record which other processors the client is sharing the data with. In the following state the client is sharing data with the directory ('client 0').

```
agent CLIENT1s0  =  read1.'readg1.CLIENT1s0
                 +  write1.CLIENT1s0WP
                 +  inv1.'inv0.CLIENT1i
                 +  invd1.'invd0.CLIENT1i
                 +  oneva.CLIENT1s0
                 +  readf12.error.0
                 +  readf10.error.0 ;
```

Notice that the directory should be handling any read requests and *not* forwarding them to this client, so if they are received then they must be erroneous.

If the local processor wants to have exclusive access then we must negotiate with the directory for that. This is essentially the PendingValid state.

```
agent CLIENT1s0WP  =
                    'wr1.inv1.'inv0.wrg1.'writeg1.excl1.CLIENT1e
                 +  inv1.'inv0.'wri1.wrgd1.'writeg1.excl1.CLIENT1e
                 +  invd1.'invd0.'wri1.wrgd1.'writeg1.excl1.CLIENT1e
                 +  readf10.'rdr0.CLIENT1s0WP
                 +  oneva.CLIENT1s0WP
                 +  readf12.error.0 ;
```

In the following we describe the situation where client 1 is sharing with client 2. It cannot tell whether it is also sharing with the directory, so therefore it cannot detect forwarding errors.

```
agent CLIENT1s2  =  read1.'readg1.CLIENT1s2
                 +  write1.CLIENT1s2WP
                 +  inv1.'inv2.CLIENT1i
                 +  invd1.'invd2.CLIENT1i
                 +  readf10.'rdg0.CLIENT1s2
                 +  oneva.CLIENT1s2
                 +  readf12.error.0 ;
```

The following represents the PendingValid state when Client 1 is pointing at Client 2.

```
    agent CLIENT1s2WP  =
                  'wr1.inv1.'inv2.wrg1.'writeg1.excl1.CLIENT1e
              +  inv1.onenva.'inv2.'wri1.wrgd1.'writeg1.excl1.CLIENT1e
              +  invd1.onenva.'invd2.'wri1.wrgd1.'writeg1.excl1.CLIENT1e
              +  readf10.'rdr0.CLIENT1s2WP
              +  oneva.CLIENT1s2WP
              +  readf12.error.0 ;
```

6 Verification

Given the process algebra model further analysis requires some notions of what it is for the system to be correct. From executing the model it is straightforward to identify that one would at least require the following:

1. at least one process claims to own the data at all times;
2. if a process makes a read request eventually it gets to read the data;
3. if a process makes a write request eventually it gets to write to the data.
4. if a process has exclusive access to the data, then no other process claims to have access.

All of these statements can be readily converted into the 'temporal macros'[?] version of the modal μ [?] validation engine of the workbench. As an example we present the formulae for the final correctness condition:

```
proposition Excl0   =  Box  [excl0]( ~<<oneva>>T & ~<<twova>>T ) ;
proposition Excl1   =  Box  [excl1]( ~<<dirva>>T & ~<<twova>>T ) ;
proposition Excl2   =  Box  [excl2]( ~<<dirva>>T & ~<<oneva>>T ) ;
```

And indeed with the complete system definition this condition is valid.

7 Conclusions

DEMOS2K[3] allows (well frankly forces) the user to present concurrent systems at an appropriate level of abstraction[4]. A level of detail which lies between that of a full implementation and that required for formal analysis, as illustrated in Figure **??**. Given that it is underpinned by a formal semantics, the system as defined in DEMOS2K provide a precise unambiguous account of its functioning. The DEMOS2K execution framework permits interactions with the dynamics of the model which allow the system describer to evaluate whether their implementation has a degree of 'face' correctness before (or not) proceeding to the difficulties of formal analysis. Obviously, if face correctness cannot be achieved then the designer is likely to conclude that a different solution is required and redesign accordingly.

The 'evidence' from the DEMOS2K executions of the model help the identification of critical validation requirements, and the consequent design of the formal model so that the relevant observations can be successfully made. The combination of the execution level validation achieved by the DEMOS2K simulation and the formal analysis provides a strong evidence base that the concurrent system design under consideration does in fact function correctly.

Acknowledgements: Thanks to Faron Moller and Graham Birtwistle for far too much help in this space. Any paucity of expression is ours and ours alone.

[3]Download available from `www.demos2k.com`

[4]One advantage of this form of software prototyping is that it is **not** releasable as usable code!

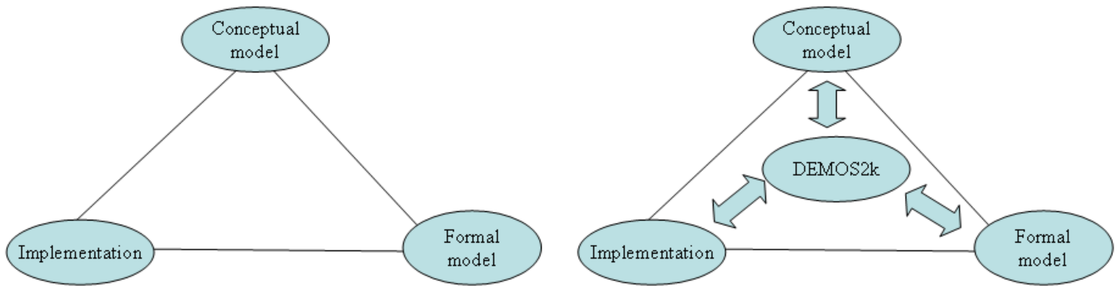

Figure 1: The conceptual distances between various modes of understanding of a concurrent system design. It is important to remember that in comprehension tasks it is the length of each step which dictates the difficulty, and two simple steps are usually **much** easier than one big one!

References

[AB86] Archibald, J. and J-L. Baer. Cache coherence protocols: evaluation using a multi-processor simulation model. *ACM Transactions on Computer Systems* **4**(4):272-298, 1986.

[Bri79] G. Birtwistle. *DEMOS — a system for discrete event modelling on Simula.* Macmillen, London, 1979.

[Bir73] G. Birtwistle, O-J. Dahl, B. Myhrhaug, and K. Nygaard. *Simula begin.* Studentlitteratur, Lund, Sweden, 1973.

[Bir85] G. Birtwistle, P.A.Luker, G.Lomow, and B.Unger. Process style packages for discrete event modelling: Experience from the transaction, activity, and event approaches. *Transactions of The Society for Computer Simulation*, 2(1):25–56, 1985.

[BPT93] G. Birtwistle, R. Pooley, and C. Tofts. Characterising the Structure of Simulation Models in CCS. *Transactions of the Society for Computer Simulation*, 10(3):205–236, 1993.

[BT93a] G. Birtwistle and C. Tofts. Operational Semantics of Process-Oriented Simulation Languages. Part 1: πDemos. *Transactions of the Society for Computer Simulation*, 10(4):299–333, 1993.

[BT93b] G. Birtwistle and C. Tofts. Operational Semantics of Process-Oriented Simulation Languages. Part 2: μDemos. *Transactions of the Society for Computer Simulation*, 11(4):303–336, 1994.

[BT96] G. Birtwistle and C. Tofts. Relating Operational and Denotational Descriptions of πDemos. *Simulation Practice and Theory*, 5(1):1–33, 1997.

[BT01a] G. Birtwistle and C. Tofts. Getting Demos Models Right - Part I: Practice. to appear *Transactions of the Society for Computer Simulation*, 2001.

[BT01b] G. Birtwistle and C. Tofts. Getting Demos Models Right - Part II: ... and Theory. to appear *Transactions of the Society for Computer Simulation*, 2001.

[BT01c] G. Birtwistle and C. Tofts. Operational Semantics of DEMOS 2000. Technical Report Number HPL-2001-263, Hewlett Packard Research Laboratories, Bristol, http://www.hpl.hp.com/techreports/2001/ 2001.

[CPS93] Cleaveland, R., J. Parrow and B. Steffen. The Concurrency Workbench: A semantics based tool for the verification of concurrent systems. *ACM Transactions on Programming Languages and Systems* **15**:36-72, 1993.

[EO94] Eriksson, L-H. and F. Orava. Analysing a π-calculus specification of a cache coherence protocol. Research Report, Swedish Institute of Computer Science, November 1994.

[GLL+90] Gharachorloo, K., D. Lenoski, J. Laudon, P. Gibbons, A. Gupta and J. Hennessy. Memory consistency and event ordering in scalable shared-memory multiprocessors. In *Proceedings of the 17th ISCA*, pp15-26, 1990.

[HLH91] Hagersten, E., A. Landin and S. Haridi. DDM: a cache-only memory architecture. Research Report R91:19, Swedish Institute of Computer Science, 1991.

[Lam79] Lamport, L. How to make a multiprocessor computer that correctly executes multiprocess programs. *IEEE Transactions on Computers* C-**28**(9):690-691, 1979.

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

[MP92] Manna, Z. and A. Pnueli. **The Temporal Logic of Reactive Systems: Specification**. Springer-Verlag, 1992.

[Mol91] Moller, F. The Edinburgh Concurrency Workbench (Version 6.0). Department of Computer Science Technical Note LFCS-TN-34, University of Edinburgh, Edinburgh, Scotland, August 1991.

[Sau95] A. Saulsbury. State transition table from COMA cache coherence simulator. Personal communication, 1995.

[SD08] Serial computing is dead; the future is parallelism SearchDataCenter.com, 30 Jun 2008 `http://searchdatacenter.techtarget.com/news/article/0,289142,sid80_gci1319113,00.html#`

[SJG92] Stenström P, T. Joe and A. Gupta. Comparative performance evaluation of cache-coherent NUMA and COMA architectures. In *Proceedings of the 19th ISCA*, pp80-91, 1992.

[Ste90] Stenström P. A survey of cache coherence schemes for multiprocessors. *Computer* **23**(6):12-24, 1990.

[Sti92] Stirling, C. Modal and temporal logics for processes. In **Logics for Concurrency: Structure vs Automata**. G. Birtwistle and F. Moller (editors), Springer-Verlag, 1995.

[TB98] C. Tofts and G. Birtwistle. A denotational semantics for a process-based simulation language. *ACM Transactions on Modelling and Simulation*, 8(3):281–305, 1998.

[Tof01] C. Tofts, HOLOS A Simulation and Multi Mathematical Modelling Tool, HPL-2001-276, `http://www.hpl.hp.co.uk/techreports/2001/HPL-2001-276.pdf`

[ZB93] Zucker, R.N. and J-L. Baer. A performance study of memory consistent models. In *Proceedings of the 19th ISCA*, pp2-12, 1993.

A COMA state transition definitions

	Shared	*Exclusive*	*Invalid*	*PendingInvalid*
cpu read	read completes	read completes	suspend cpu find global AU tag message.requester_p = NodeID(ThisNode) read request $\frown$ DirNode → PendingInvalid	—
cpu write	suspend cpu find global AU tag message.requester_p = NodeID(ThisNode) write request $\frown$ DirNode → PendingValid	write completes	suspend cpu find global AU tag message.requester_p = NodeID(ThisNode) write request $\frown$ DirNode → PendingInvalid	—
read request	—	—	—	—
bounced read request	—	—	—	read request $\frown$ DirNode

read request fwd	message.ptr = pointer
	pointer = requester
	take shared + data $\frown$ requester
		message.ptr = NULL_NODE
		pointer = requester
		take shared + data $\frown$ requester
		→ Shared
			fake msg: requester = original requester
			read request $\frown$ DirNode
				fake msg: requester = original re
				read request $\frown$ DirNode

take shared	—	—	—	fill in data pointer = message.ptr → Shared restart cpu
dir read request	dir take shared + data $\frown$ DirNode	dir take shared + data $\frown$ DirNode → Shared	—	dir rd req bounded $\frown$ DirNode
dir take shared	—	—	—	—
dir rd req bounced	—	—	—	—
write request	—	—	—	—
invalidate with	if (pointer==NULL_NODE) invalidated with + data $\frown$ DirNode else invalidate with $\frown$ pointer → Invalid	invalidated with + data $\frown$ DirNode → Invalid pointer = NUL_NODE	—	→ PendingInvalidInvalidated absorb message
invalidate without	if (pointer==NULL_NODE) invalidated without $\frown$ DirNode else invalidate without $\frown$ pointer → Invalid	—	—	→ PendingInvalidInvalidated absorb message
invalidated with	—	—	—	—

	PendingInvalid	*PendingValid*	*PendingInvalidInvalidated*	
cpu read	read completes	read completes	suspend cpu find global AU tag message.requester_p = NodeID(ThisNode) read request $\frown$ DirNode → PendingInvalid	—
cpu write	suspend cpu find global AU tag message.requester_p = NodeID(ThisNode) write request $\frown$ DirNode → PendingValid	write completes	suspend cpu find global AU tag message.requester_p = NodeID(ThisNode) write request $\frown$ DirNode → PendingInvalid	—
read request	—	—	—	—
bounced read request	—	—	—	read request $\frown$ DirNode

read request fwd	message.ptr = pointer
	pointer = requester
	take shared + data $\frown$ requester
		message.ptr = NULL_NODE
		pointer = requester
		take shared + data $\frown$ requester
		→ Shared
			fake msg: requester = original requester
			read request $\frown$ DirNode
				fake msg: requester = original re
				read request $\frown$ DirNode

take shared	—	—	—	fill in data pointer = message.ptr → Shared restart cpu
dir read request	dir take shared + data $\frown$ DirNode	dir take shared + data $\frown$ DirNode → Shared	—	dir rd req bounded $\frown$ DirNode
dir take shared	—	—	—	—
dir rd req bounced	—	—	—	—
write request	—	—	—	—

invalidate with	if (pointer==NULL_NODE)
	    invalidated with + data $\frown$ DirNode
	else
	    invalidate with $\frown$ pointer
	→ Invalid
		invalidated with + data $\frown$ DirNode
		→ Invalid
		pointer = NUL_NODE
			—	→ PendingInvalidInvalidated
				absorb message

invalidate without	if (pointer==NULL_NODE)
	    invalidated without $\frown$ DirNode
	else
	    invalidate without $\frown$ pointer
	→ Invalid
		—	—	→ PendingInvalidInvalidated
				absorb message

invalidated with	—	—	—	—

	Dir Shared	*Dir Exclusive*	*Dir Invalid*	*Dir PendingStalled*
cpu read	read completes	read completes	suspend cpu find global AU tag message.requester_p = NodeID(ThisNode) dir read request ⌒ pointer → Dir PendingStalled	—
cpu write	suspend cpu find global AU tag message.requester_p = NodeID(ThisNode) invalidate without ⌒ pointer → Dir PendingStalled	write completes	suspend cpu find global AU tag message.requester_p = NodeID(ThisNode) invalidate with ⌒ pointer → Dir PendingStalled	—

read request	message.ptr = pointer
	pointer = requester
	take shared + data ⌒ requester
		message.ptr = NULL_NODE
		pointer = requester
		take shared + data ⌒ requester
		→ Dir Shared
			read request fwd ⌒ pointer	bounced read request ⌒ re

bounced read request	—	—	—	—
read request fwd	—	—	—	—
take shared	—	—	—	—
dir read request	—	—	—	—
dir take shared	—	—	—	get data → Dir Shared restart cpu
dir rd req bounced	—	—	—	dir read request ⌒ pointer
write request	invalidate without ⌒ pointer → Dir Pending	write enable + data ⌒ requester pointer = requester → Dir Invalid	invalidate with ⌒ pointer pointer = requester → Dir Pending	bounced write request ⌒ r
invalidate with	—	—	—	—
invalidate without	—	—	—	—

invalidated with	—	—	—	if (requester == ThisNode
				  get data
				  → Dir Exclusive
				  restart cpu
				else
				  forward received data
				  pointer = requester
				  write enable + data ⌒
				  → Dir Invalid
				  also unstall directory cp

invalidated without				if (requester == ThisNode

B COMA in DEMOS2k

```
cons getRate=negexp(20);
cons ReadWrite=bin(1,0.95);
cons transfer=negexp(1);
cons clients=50;
cons Range=puni(0,clients);

bin(LocalR,0);
bin(LocalW,0);
bin(grant,0);
bin(read,0);
bin(write,0);
bin(invalid,0);
bin(readForward,0);

class ExclusiveD=
{local var dest=0;
 try [getvb(read,[dest],true)]
  then {trace("Eread %v",dest);
        hold(transfer);
        putvb(grant,[dest,0]);
        entity(SharedD,SharedD(#dest),0);
       }
 etry [getvb(write,[dest],true)]
  then {trace("Ewrite %v",dest);
        hold(transfer);
        putvb(grant,[dest,0]);
        entity(InvalidD,InvalidD(#dest),0);
        }
 etry [getvb(LocalR,[dest],dest==0.0)] then
 {entity(EX,ExclusiveD,0);}

 etry [getvb(LocalW,[dest],dest==0.0)] then
 {entity(EX,ExclusiveD,0);}
}

class SharedD(current)=
{local var dest=0;
 local var invD=0;

  trace("Scurrent %v",current);
  try [getvb(read,[dest],true)]
  then {trace("read %v",dest);
        hold(transfer);
        putvb(grant,[dest,current]);
        entity(SharedD,SharedD(#dest),0);
       }
 etry [getvb(write,[dest],true)]
  then {trace("write %v",dest);
        hold(transfer);
        trace("Start invalid %v",current);
        putvb(invalid,[current]);
        getvb(invalid,[invD],invD==0);
        trace("Granting write %v",dest);
        putvb(grant,[dest,0]);
        entity(InvalidD,InvalidD(#dest),0);
        }
 etry [getvb(LocalR,[dest],dest==0.0)] then
 {entity(SharedD,SharedD(#current),0);}
 etry [getvb(LocalW,[dest],dest==0.0)] then
 {hold(transfer);
  putvb(invalid,[current]);
  getvb(invalid,[invD],invD==0);
  entity(EX,ExclusiveD,0);
 }
}
```

```
class InvalidD(current)=
{local var dest=0;
 local var test=0;
  trace("Icurrent %v",current);
  try [getvb(read,[dest],true)]
  then {trace("Iread %v",dest);
        hold(transfer);
        putvb(readForward,[current,dest]);
        entity(InvalidD,InvalidD(#dest),0);
       }
 etry [getvb(write,[dest],true)]
  then {trace("Iwrite %v",dest);
        hold(transfer);
        trace("start invalid %v",current);
        putvb(invalid,[current]);
        getvb(invalid,[test],test==0.0);
        putvb(grant,[dest,0]);
        entity(InvalidD,InvalidD(#dest),0);
        }
 etry [getvb(LocalR,[dest],dest==0.0)] then
 {hold(transfer);
  putvb(readForward,[current,0.0]);
  getvb(grant,[dest,test],dest==0.0);
  entity(SharedD,SharedD(#current),0);
 }
 etry [getvb(LocalW,[dest],dest==0.0)] then
 {hold(transfer);
  trace("start invalid %v",current);
  putvb(invalid,[current]);
  getvb(invalid,[test],test==0);
  entity(EX,ExclusiveD,0);
 }
}

class ClientI(number)=
{local var dest=0;
 local var gID=0;
 local var gNext=0;

 try [getvb(LocalR,[dest],dest==number)] then
 {putvb(read,[number]);
  getvb(grant,[gID,gNext],gID==number);
  entity(ClientShared,ClientS(#number,#gNext),0);
 }
 etry [getvb(LocalW,[dest],dest==number)] then
 {putvb(write,[number]);
  getvb(grant,[gID,gNext],gID==number);
  entity(ClientEx,ClientEX(#number),0);
 }
}

class ClientS(number,next)=
{local var target=0;
 local var dest=0;
 local var gID=0;
 local var gNext=0;
  trace("%v to %v",number,next);
  try [getvb(invalid,[target],target==number)]
  then {hold(transfer);
        trace("put invalid %v",next);
        putvb(invalid,[next]);
        entity(ClientI,ClientI(#number),0);
       }
  etry [getvb(readForward,[target,dest],target==number)]
  then {hold(transfer);
        trace("ReadForward number %v next %v req %v",number,next,dest);
        putvb(grant,[dest,number]);
        try [dest==0.0] then {entity(ClientShared,ClientS(#number,#next),0);}
        etry [] then {entity(ClientShared,ClientS(#number,#next),0);}
       }
  etry [getvb(LocalR,[dest],dest==number)] then
  {entity(ClientShared,ClientS(#number,#next),0); }
  etry [getvb(LocalW,[dest],dest==number)] then
  {putvb(write,[number]);
   getvb(invalid,[target],target==number);
   hold(transfer);
   trace("put invalid %v",next);
   putvb(invalid,[next]);
   getvb(grant,[gID,gNext],gID==number);
   entity(ClientEx,ClientEX(#number),0);
  }
}

class ClientEX(number)=
{local var target=0;
 local var dest=0;

  trace("Got exclusive %v",number);
  try [getvb(invalid,[target],target==number)]
  then {hold(transfer);
        putvb(invalid,[0.0]);
        entity(ClientI,ClientI(#number),0);
       }
  etry [getvb(readForward,[target,dest],target==number)]
  then {hold(transfer);
        trace("ReadForward number %v req %v",number,dest);
        putvb(grant,[dest,number]);
        entity(ClientShared,ClientS(#number,#0.0),0);
       }
  etry [getvb(LocalR,[dest],dest==number)] then
  {entity(ClientEx,ClientEX(#number),0); }
  etry [getvb(LocalW,[dest],dest==number)] then
  {entity(ClientEx,ClientEX(#number),0);}
}

class work=
{repeat
 {hold(getRate);
  try [ReadWrite==1] then {putvb(LocalR,[Range]);}
  etry [] then  {putvb(LocalW,[Range]);}
 }
}

entity(EX,ExclusiveD,0);
var i=1;do clients {entity(ClientI,ClientI(#i),0);i:=i+1;}
entity(W,work,0);

hold(100000);
close;
```