Tutorial on Verification of Distributed Cache Memory Protocols

Steven M. German

IBM T.J. Watson Research Center

Formal Methods in Computer-Aided Design 2004

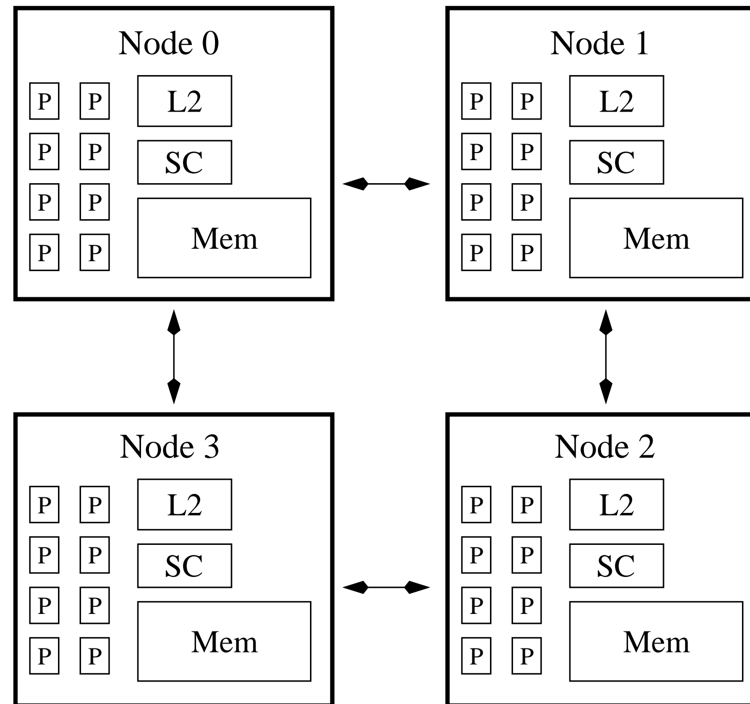
Outline

1. Modelling and verification of memory protocols at the architecture level
   - An example of an architecture-level model
2. Experiences with verifying complex protocols
   - Latest generation mainframe
   - Formal design in the memory protocol for the ASCI Blue supercomputer
3. Formal Design of hardware implementations
   - Developing hardware implementations from guarded command specifications

Part 1

Modelling and Verification of Memory Protocols

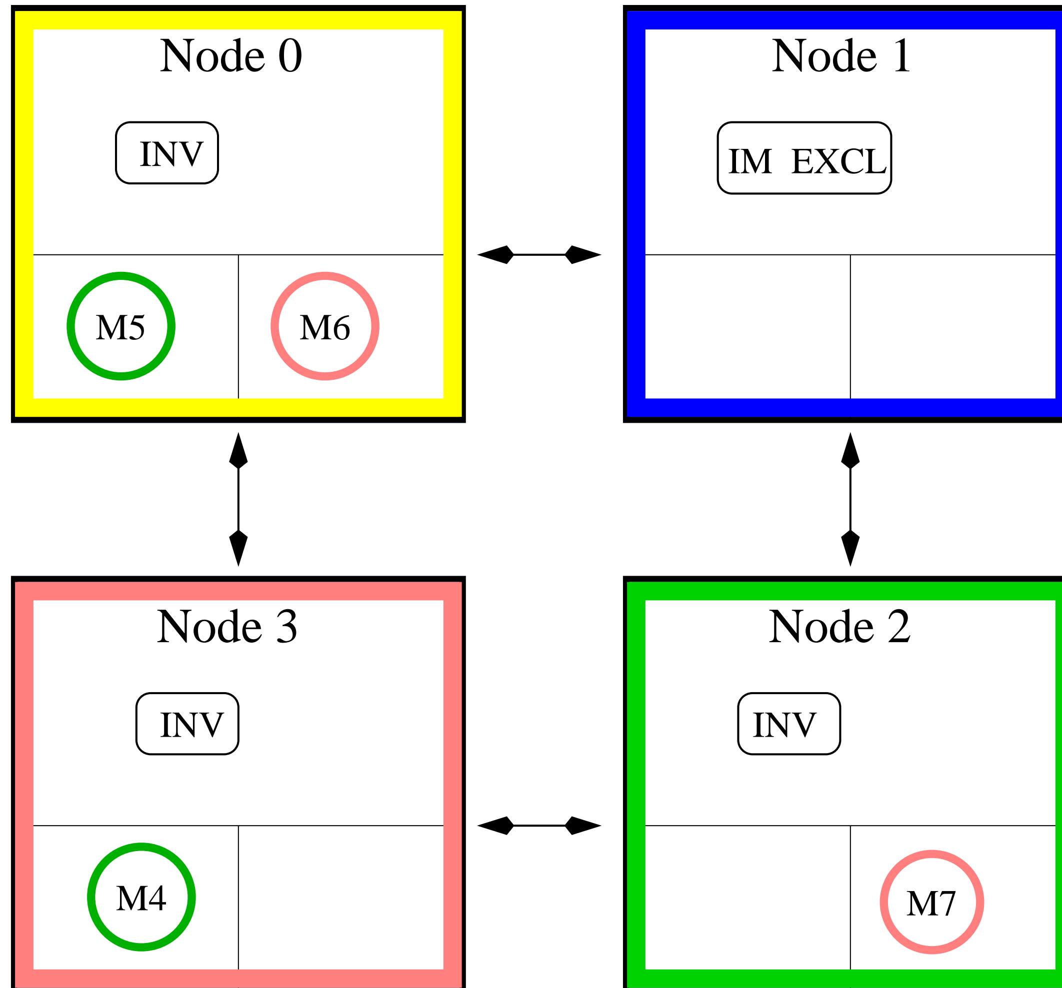
Mainframe Server System Ring



The system has four nodes connected on a bidirectional ring.

- Processors (with individual L1 caches)
- A shared L2 cache
- Memory (L3)
- System Control Element (SC): Provides coherent access to data across nodes

Ring Protocol Model



Abstract model shows states and messages for one cache line.

Transitions are larger than a clock cycle.

Themes of This Talk

- Why abstract protocol models are needed
- How to make abstract models of memory system hardware
- How to assure the hardware is consistent with the protocol model – emerging work

Traditional Approach to Verification in Industry

- Traditional view is that the RTL serves as the most precise statement of the design.
  - Design problems are worked out at RTL level
- Traditional verification is based on random simulation of completed parts of the RTL
- Simulation expert guides the simulation toward critical tests, using knowledge of the design.

Problems with Traditional Approach

- Simulation of an entire system gives poor coverage.

> Errors escape random simulation of HDL!

- The implementation is usually too large to check directly by model checking,
- Compositional model checking of the implementation is difficult because of the need for correct specifications of the components.
- Many of the errors found in testing protocol hardware are errors in distributed computing.
  - These errors should be found earlier!

Integrating Formal Methods Into the Design Process

Level 1: Formal verification of completed parts of RTL design

- Problem of scale: Global properties cannot be formally verified at RTL level!

Level 2: Architectural Verification

- An abstract model of the design is created and verified by a formal methods expert. Designers make limited use of model.

  The model is used to derive specifications for system simulation.

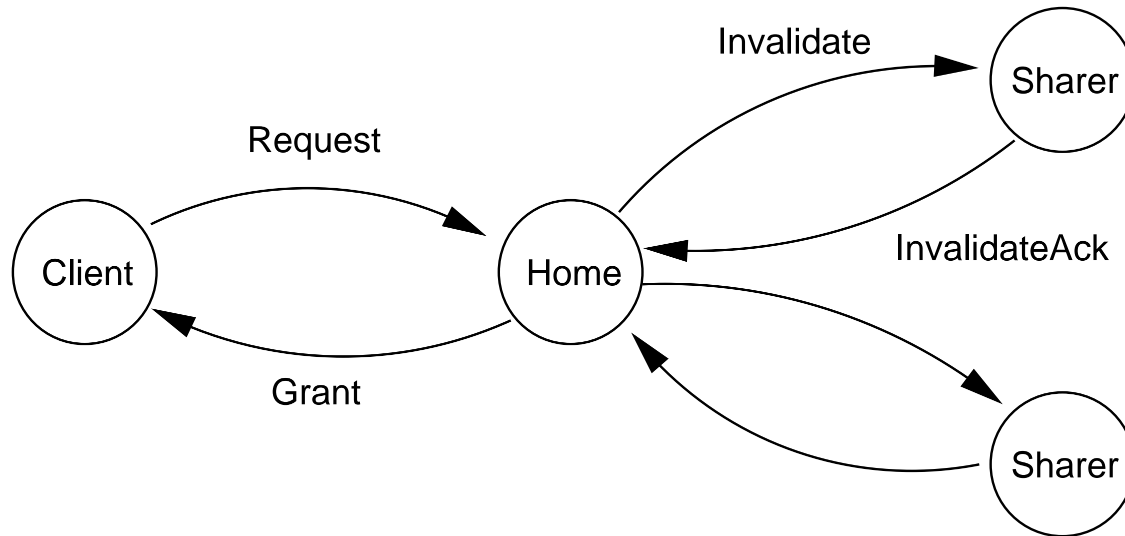
Level 3: Formal Design

- A formally verified model serves as a *guide* for RTL development.

Formal Design of a hardware protocol was first explored in IBM on the shared memory adaptor cache protocol for ASCI Blue.

The ASCI Blue RTL was completed ahead of schedule and with very few errors.

Tutorial Cache Coherence Protocol – Very High Level View



Four classes of messages

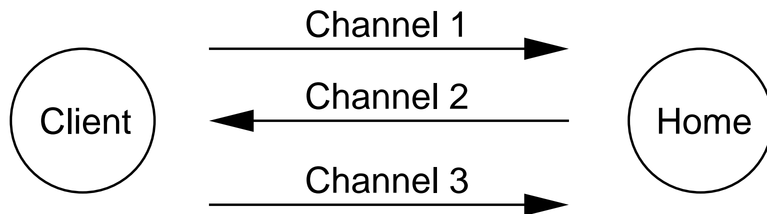
Type 1 Request to Home node

Type 2 Invalidate a remote cache

Type 3 Invalidate Acknowledgement

Type 4 Reply (Grant) from Home node

Channel Assignments



Three FIFO message channels between each pair of nodes:

Channel 1: Type 1 Request to Home node

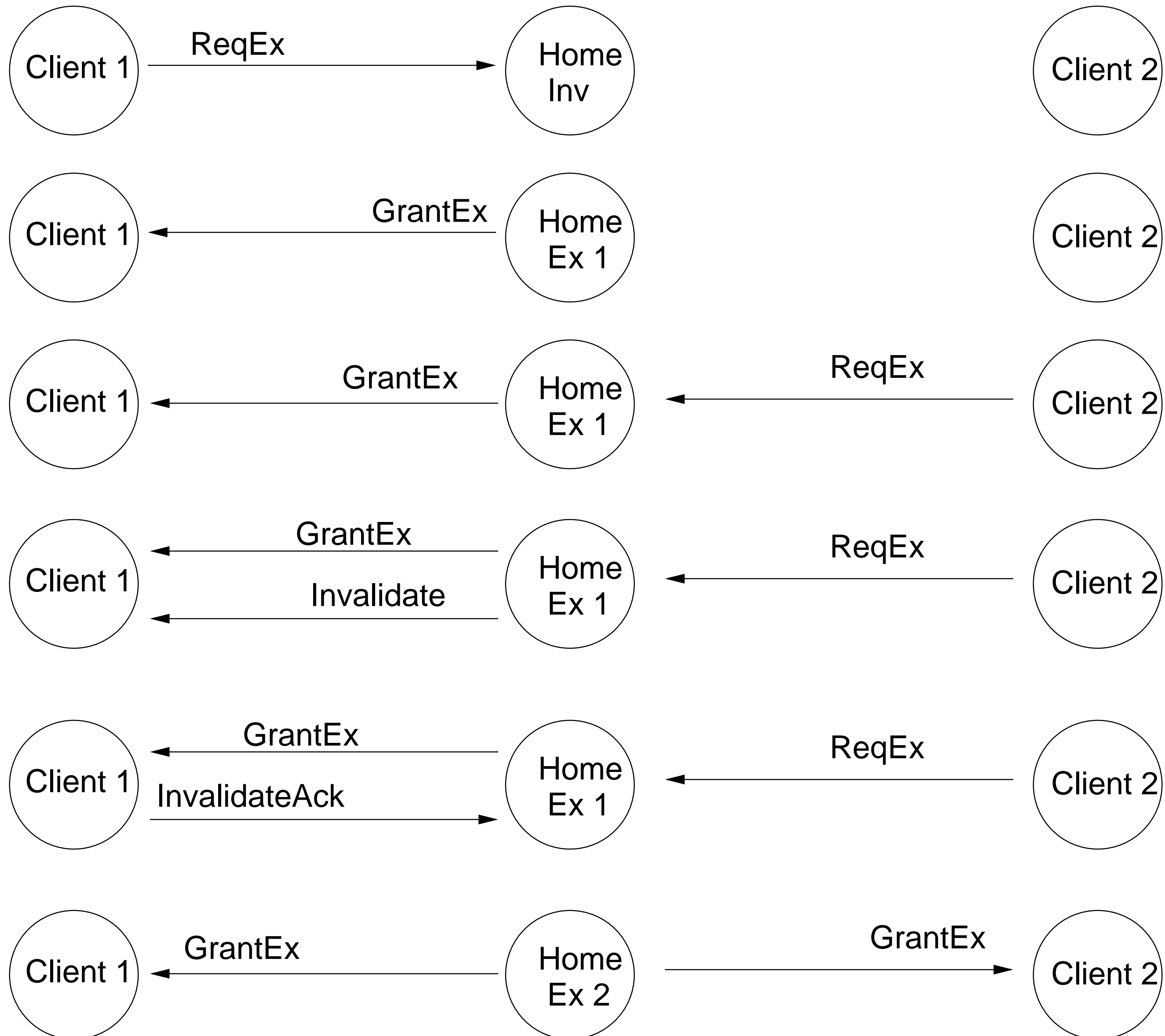
Channel 2: Type 2 Invalidate a remote cache
Type 4 Reply from Home node

Channel 3: Type 3 Invalidate Acknowledgement

Why three channels?

- Three channels is enough to avoid deadlock
- Types 2 and 4 on same channel to maintain coherence.

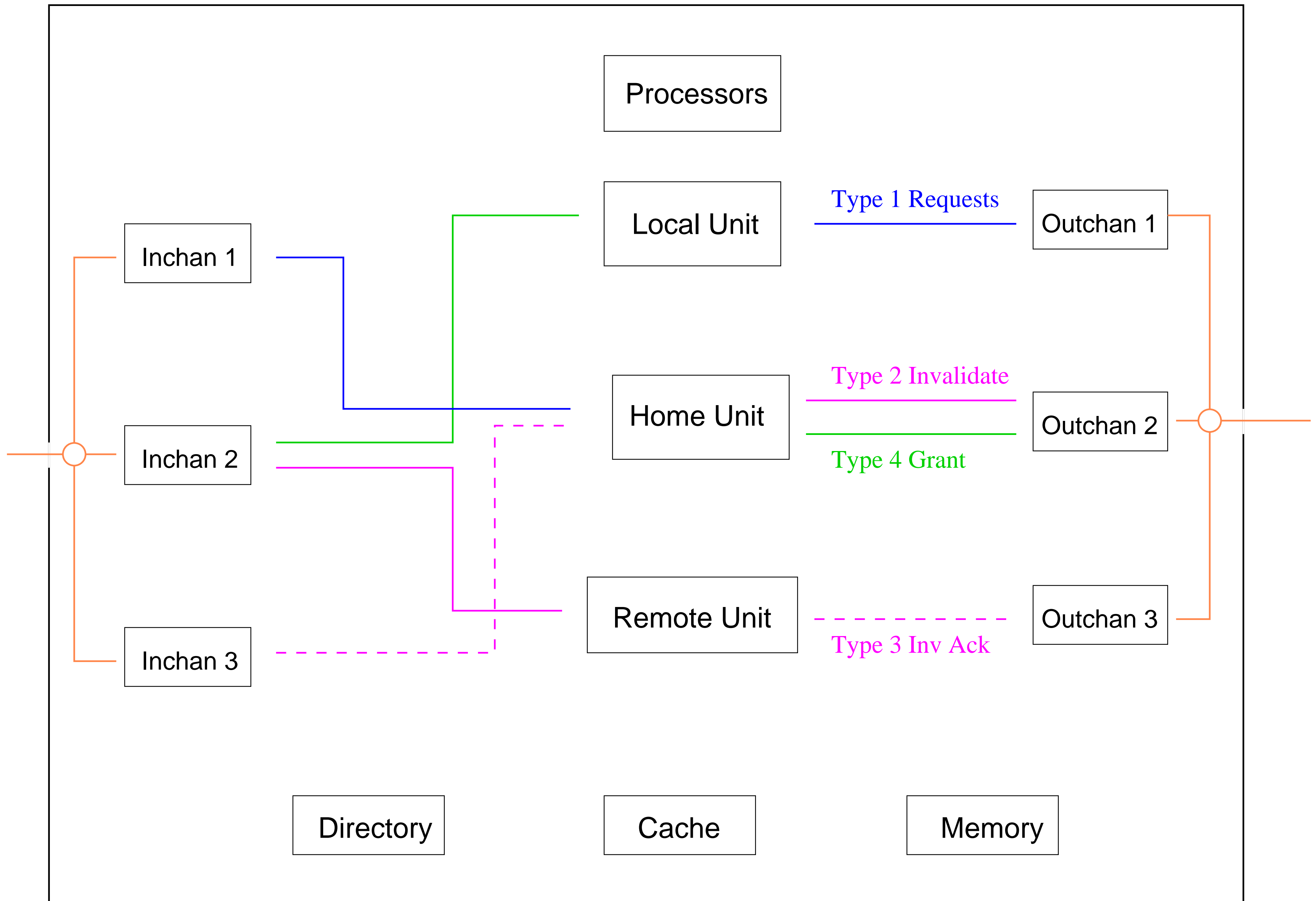
A Race Condition



Detailed Model – Represents Hardware Architecture

- Each node acts as both home and client
- Each node has Local Unit, Home Unit, Remote Unit
- Refinement of message processing into phases: Receive message, Process, Send reply
- Multiple addresses
- Multiple requests active at a node at same time
- Memory hierarchy: Main Memory and Cache
- Protocol shows movement of data
- More detail in messages
- IO channels refined
- Nodes can send messages to themselves (Request, Invalidate)

Structure of a Node



Murphi

Developed by David Dill, *et al*, Stanford University

A Murphi model consists of

- Global state variables
- Transition rules

$$condition \;\; ==> \;\; body$$

Data types: Boolean, Enumerated, Integer Subrange, Arrays, Records

A model is executed by applying one transition at each step.

A transition can be executed at a step if its enabling condition is true

Interleaving or Unity semantics

A model can specify transitions compactly using *function* and *procedure* definitions.

States of the simple model

```
type message: enum{empty, req_shared, req_exclusive,
                   invalidate, invalidate_ack,
                   grant_shared, grant_exclusive};

     cache_state: enum{invalid, shared, exclusive};

     client: 1 .. num_clients;

-- state of home node:
var home_sharer_list, home_invalidate_list:
                       array[client] of boolean;

var home_exclusive_granted: boolean;

var home_current_command: message;

var home_current_client: client;
```

```
-- state of caches in client nodes
var cache: array[client] of cache_state;


-- message channels
-- requests to home
var channel1: array[client] of message;

-- messages to client
var channel2_4: array[client] of message;

-- invalidate acks to home
var channel3: array[client] of message;
```

Detailed Model – Node

```
node_type:

 record
  local_requests: array[addr_type] of boolean;
  home_requests: array[addr_type] of home_request_type;
  remote_requests: array[addr_type] of remote_request_type;
  --  NOTE requests are indexed by address

  -- input channels
  inchan: array[channel_id] of message_buf_type;

  -- output channels
  outchan: array[channel_id] of message_buf_type;

  directory: directory_type;
  cache:  cache_type;
  memory: memory_type;
 end;
```

Detailed Model – State of Processing Units

```
status_type: enum{inactive, pending, completed};

home_request_type:
  record
   source: node_id;
   op: opcode;
   data: data_type;
   invalidate_list: node_bool_array;
   status: status_type;
  end;

 -- a type for remote unit.
remote_request_type:
  record
   home: node_id;
   op:   opcode;
   data: data_type;
   status: status_type;
  end;
```

Detailed Model – Message Types

```
message_type:
  record
    source: node_id;
    dest: node_id;
    op:   opcode;
    addr: addr_type;
    data: data_type;
  end;

message_buf_type:
  record
   msg: message_type;
   valid: boolean;
  end;
```

Complete Model

- Complete model is in Appendix A.
- Joint work with Geert Janssen, IBM

Defining Transitions

Transitions should be chosen to correspond to *atomic operations* in the hardware.

- Each operation in the hardware should only see the state before or after an atomic action.
- Atomic actions may take several clock cycles in the hardware.

How do we know the hardware implements a set of atomic operations?

- Designer intent
  - Memory system hardware often has natural atomic operations.
  - Atomic operations may be more common in memory systems than other hardware such as processor pipelines.
- We must design the hardware implementation to make the atomic operations verifiable.

Ordering of Transitions

- The model should generate all orderings of the atomic actions that are possible in the hardware.
- The orderings of actions are precedence relations:

  `Tr2` cannot preceed `Tr1`

- Introduce ordering variables in the model:

```
        var Tr1_done: boolean;

Tr1:    guard1 ==> begin  . . .   Tr1_done := true; end;

Tr2:    guard2 & Tr1_done ==>  begin  . . .  end;
```

- The model should conservatively *overapproximate* the orderings of actions in the hardware (for checking safety properties).

Points of Coherence

The tutorial example has a single *point of coherence* (home node) for each address.

Some large commercial servers use broadcasts and have a hierarchy of points of coherence:

- A cache in an exclusive state
- Caches in a shared state
- Main memory

Rules for overlapping requests are complicated in these protocols.

Phases in Verification of Cache Memory Protocol

- Message Fabric
- Coherence Protocol

Verification of Message Fabric

- Absence of deadlock
- Absence of buffer overflow
- Liveness properties

Usually need to use a multiple-address model because of resource contention.

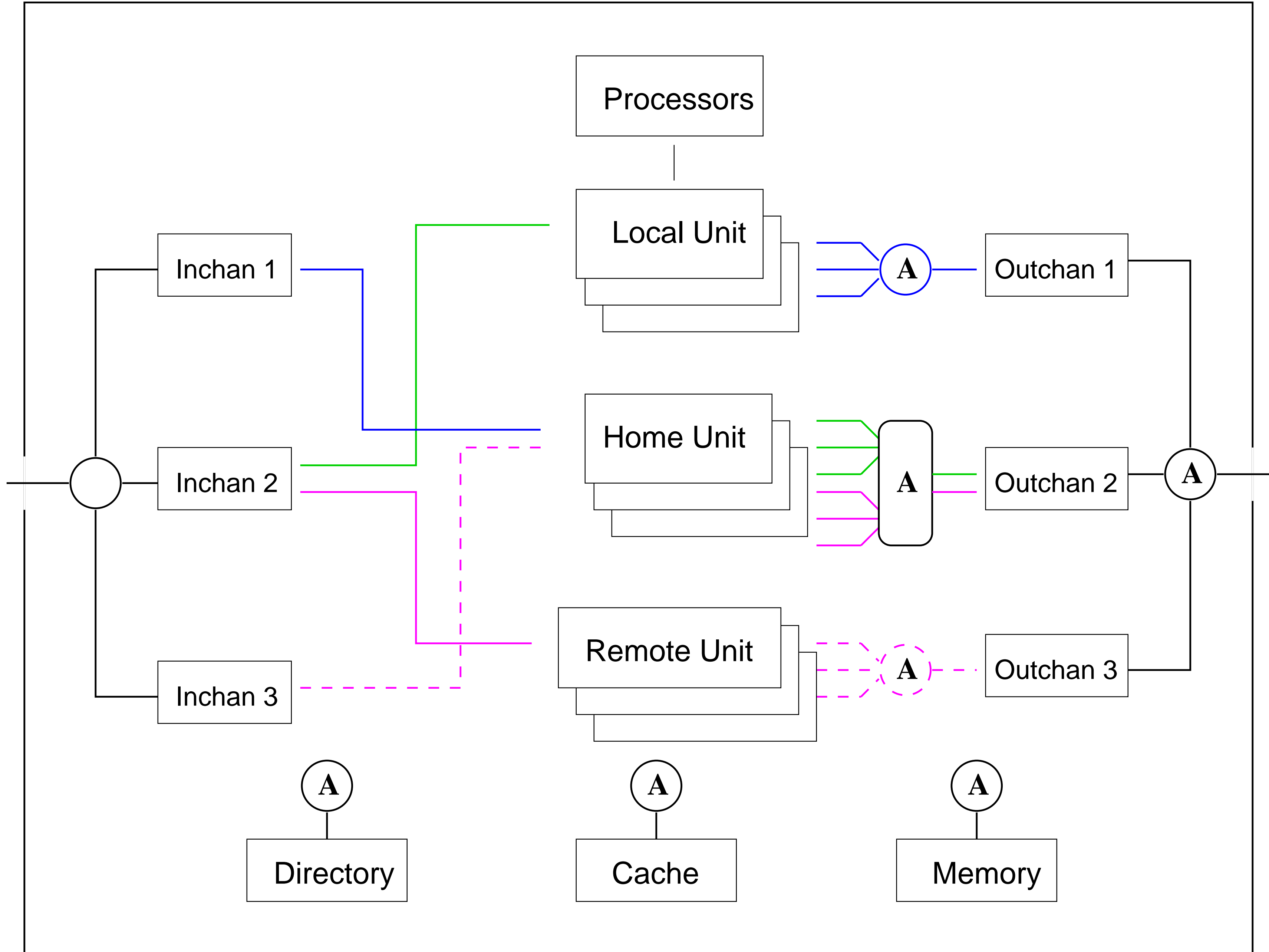
For these properties, we model the arbitration scheme of the message fabric.

Message fabrics can often be analyzed independently from the coherence protocol.

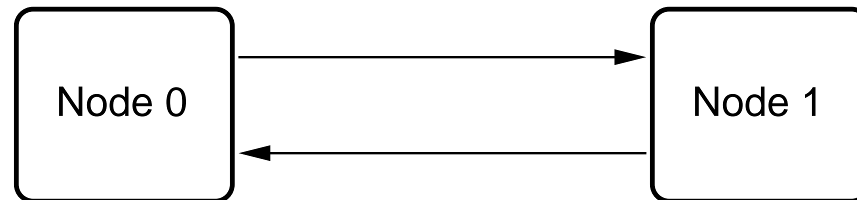
Proof methods:

- Manual proofs
- *Quantitative model checking* for buffer sizes
- Model checking on fabric model (abstract details of coherence)

Message Fabric and Arbitration



A Simple Protocol

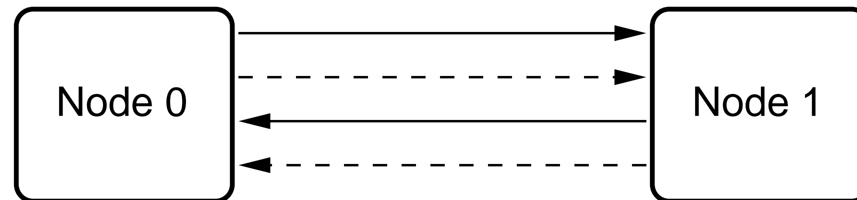


A single FIFO queue in each direction

Protocol actions

1. Send Request
2. $\ll$ Receive Request; Send Reply $\gg$
3. Receive Reply

Avoiding Deadlock



- Requests on solid lines
- Replies on dotted lines

Absence of deadlock can be shown with a simple proof using case analysis.

Quantitative Model Checking

Explicit or symbolic model checkers can be easily modified to report the maximum value of a numeric variable over the reachable states.

A typical use:

Build an abstract model to analyze buffer sizes (prevent buffer overflow).

- Define a model where the states are the number of messages in each buffer.
- Quantitative model checker reports the maximum occupancy for each buffer.

Properties of Coherence Protocol

- Coherence

  "If a cache has a copy of the data is marked *Exclusive*, then no other cache has a copy."

- Data freshness

  "All cached copies of the data have the most recently written value."

  Note: Memory may have a stale value.

- Note: Ordering properties of memory models are often implemented within processors.

Part 2

Experiences with Verification of Complex Protocols

Model Checking the Mainframe Server Protocol

- Subtle errors were found by model checking
- Errors found by model checking led to **Major Changes** in design of protocol!
  - Rules about order of message processing in nodes
  - Rules about status of memory requests
  - A planned memory transaction was dropped

Most of the Errors found by formal verification were NOT found by simulation.

Model Checking vs. Simulation of Server Protocol

Random simulation was applied constantly to the hardware implementation.

Model checking was applied to the protocol high level model.

- Simulation found a large number of hardware errors
- Most of the Errors found by model checking were not found by simulation
- In some cases, simulation continued for many months on uncorrected hardware, without detecting known protocol errors.
- Compared model checking with random simulation on a Murphi model.

  Model checking found error in 16 hours.

  Random simulation did not find error even once in 7 days ($\geq$ factor of 10)

A Shared Memory Protocol for ASCI Blue

ASCI Blue

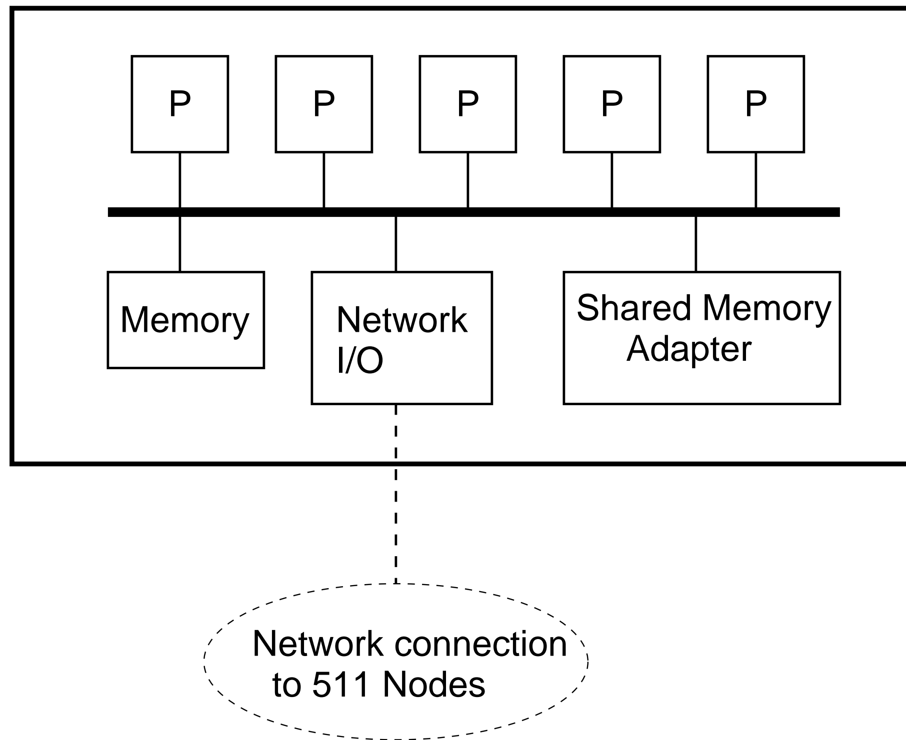
- Designed for U.S. Department of Energy
- 512 nodes
- 8 or 16 processors in each node
- High-speed communication network connecting nodes

Shared memory protocol

- Message fabric
- Coherence protocol
- Migration protocol

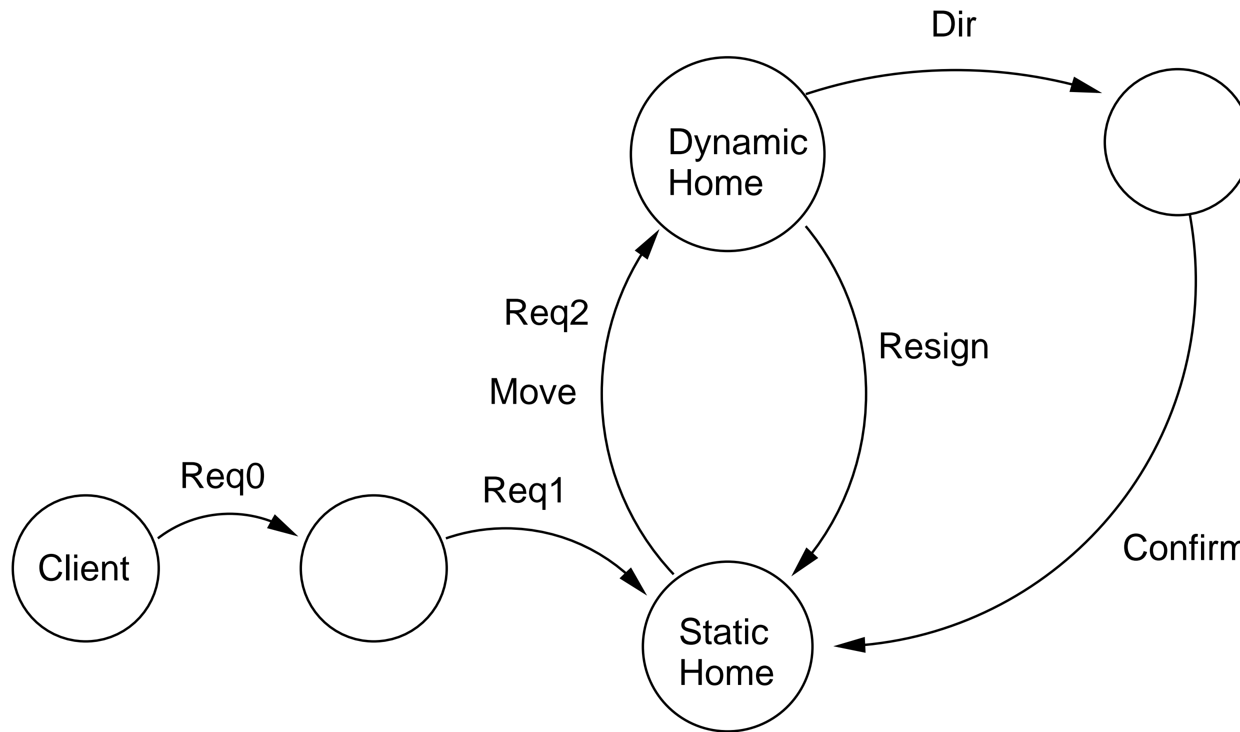
Joint work with protocol architect Kattamuri Ekanadham

ASCI Blue Node



At a high-level, the coherence protocol is similar to the German protocol.

Migration Protocol



Migration runs concurrently with the coherence protocol.

Important Properties

- Requests are eventually serviced (liveness). Requests cannot be forwarded forever.
- Order in which requests are serviced

Design Process for Shared Memory Adaptor

- Formal modelling and verification preceeded VHDL implementation.
- Our first approach to using a formal model as a guide for design of the hardware.
- The design was developed at three levels of abstraction.
  1. High level design
  2. Pipeline level design
  3. Hardware implementation

Micro-architecture Design of the Protocol Hardware

The internal structure of a node is a set of dataflow units that send and receive messages on internal communication channels.

Channel assignment and deadlock analysis was refined for this level of design.

Design was verified by hand proof and by Murphi.

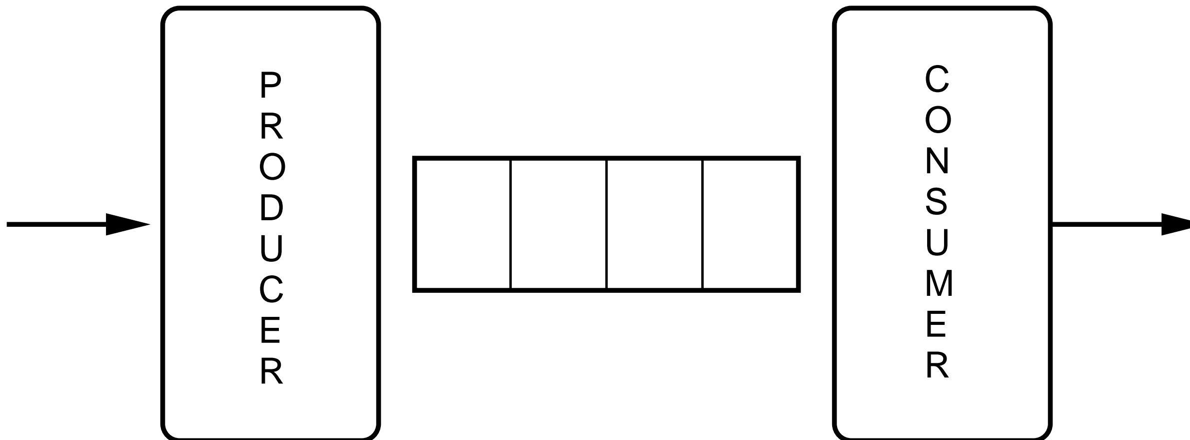
A Producer-Consumer protocol to control communication between adjacent dataflow units was designed and verified with Murphi. The VHDL code was verified with VIS.

Producer-Consumer Protocol

Producer and Consumer must communicate to keep Consumer busy as much as possible without overflowing queue.

Consumer needs a variable, unknown amount of time to process each message.

Producer needs at least one clock cycle to respond to a request to produce data or stop producing data.



Implementation of the Coherence Protocol

The hardware design team used the following approach:

- Dataflow units were implemented using a high-level notation called *vcode*.
- Vcode macro-expands to synthesizeable VHDL.
- The vcode description is much more concise than VHDL.
- Communication between dataflow units is built-in to vcode descriptions.

Overall, vcode accelerated the design of the hardware.

Very few errors were found in the portions of hardware designed with vcode.

References

Formal Design of Cache Memory Protocols in IBM,
S. German,
Formal Methods in System Design, 22, 2, March 2003,
Special Issue on Industrial Practice of Formal Hardware Verification,
G. Gopalakrishnan and W. Hunt, guest editors

Functional Verification of the z990 Superscalar, Multibook Microprocessor Complex,
D. Bair, S. German, W. Wollyung, E. Kaminski, J. Schafer, M. Mullen, W. Lewis, R. Wisniewski, J. Walter, S. Mittermaier, V. Vokhshoori, R. Adkins, M. Halas,
T. Ruane, and U. Hahn,
IBM Journal of Research and Development, 48, 3/4, May/July 2004,
Special Issue on IBM eServer z990

Part 3

Formal Design of Hardware from Guarded Commands

Project Objectives

- Develop specification methods based on *guarded commands* for specifying hardware systems.
- Develop methods for verifying that a hardware system is a correct implementation of a specification.

Related Work

- Jørgen Staunstrup

  Synchronized Transitions

  A Formal Approach to Hardware Design (1994)

- James Hoe and Arvind, MIT

  Term Rewriting Systems

  Hardware Synthesis from Term Rewriting Systems (1999)

- Bluespec, Inc. (2004)

  Hardware design toolset

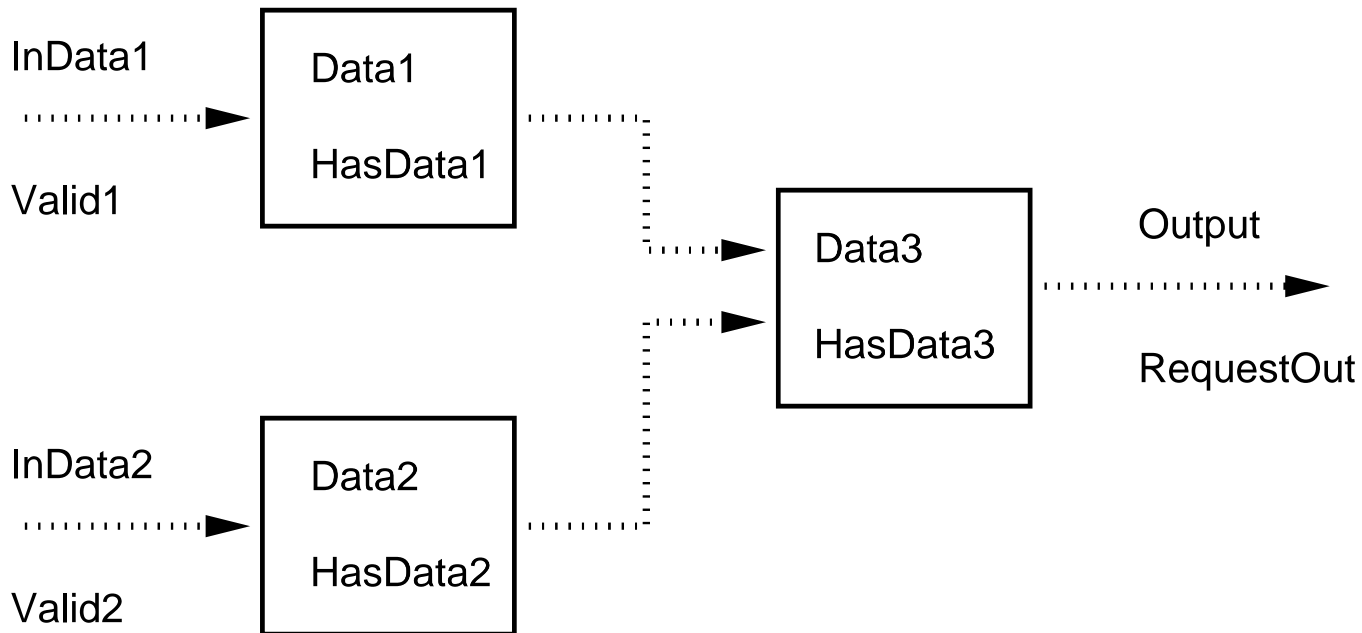
Differences from previous approaches:

- Previous work assumed complete or even synthesizeable specification

- We allow hardware designers to use expertise to implement the specification

- We verify conformance of implementation to specification

An Example



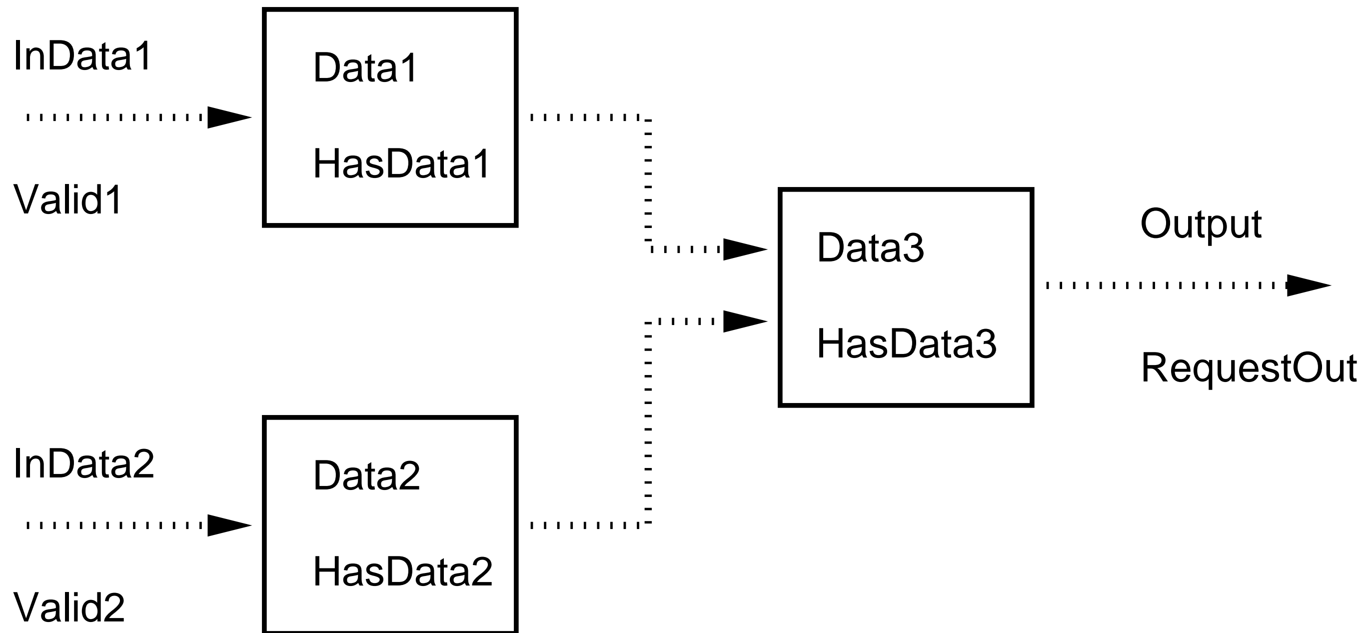

```
Variables

input InData1, InData2, Valid1, Valid2, RequestOut;

var InData1, InData2, Data1, Data2, Data3, Output: char;

var Valid1, Valid2, HasData1, HasData2, HasData3,
    RequestOut: boolean;
```

Transitions



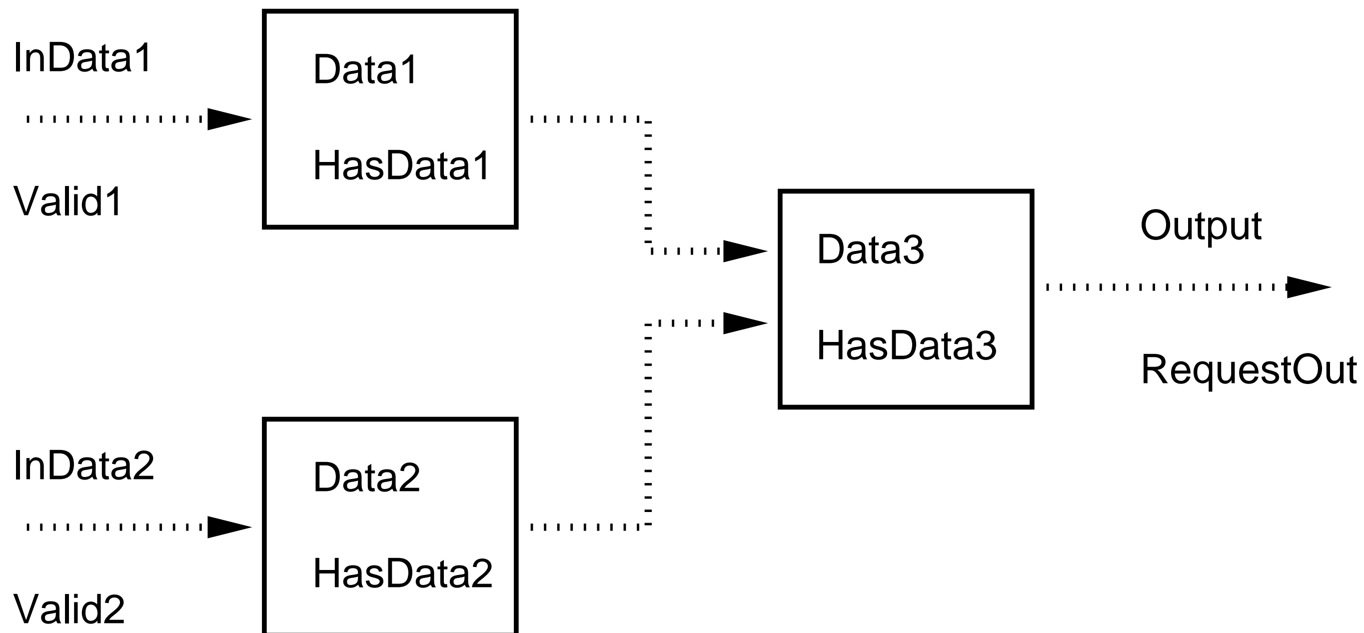

```
rule "in1" !HasData1 & Valid1 ==>
      Begin Data1 := InData1; HasData1 := true; End;

rule "in2" !HasData2 & Valid2 ==>
      Begin Data2 := InData2; HasData2 := true; End;

rule "move1" HasData1 & !HasData3 ==>  Begin
 Data3 := Data1; HasData1 := false; HasData3 := true; End;
```

```
rule "move2" HasData2 & !HasData3 ==>  Begin
 Data3 := Data2; HasData2 := false; HasData3 := true; End;

rule "out" RequestOut & HasData3 ==>
      Begin Output := Data3; HasData3 := false; End;
```



Mapping to Hardware

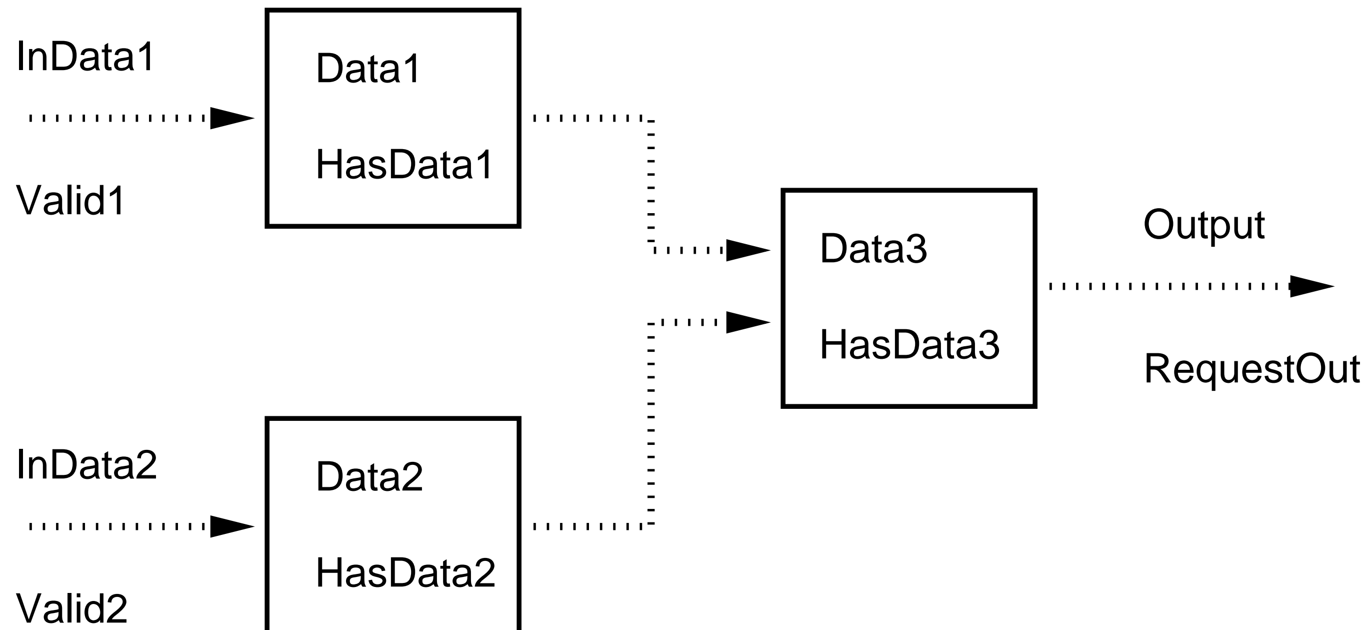
Variables $\Longleftrightarrow$ Hardware registers

Transitions $\Longleftrightarrow$ Hardware processes or state machines

A hardware process can take multiple clock cycles to execute a transition.

In hardware, *independent processes* can be active at the same time.

In the example, hardware can implement the communication with *serial signal transfers*; multiple transfers can overlap in time.



Independent Processes

For a process P that implements a transition $condition \;==>\; body$, we define:

$R_P = \{$registers read by the implementation of the guard or the body$\}$

$W_P = \{$registers updated by the implementation of the body$\}$

The implementation of the guard must not change the state of the hardware.

Processes P_1, P_2 are *independent* if

$$W_1 \cap W_2 = \emptyset$$

$$R_1 \cap W_2 = \emptyset$$
$$R_2 \cap W_1 = \emptyset$$

Several processes can overlap in time provided they are all pairwise independent.

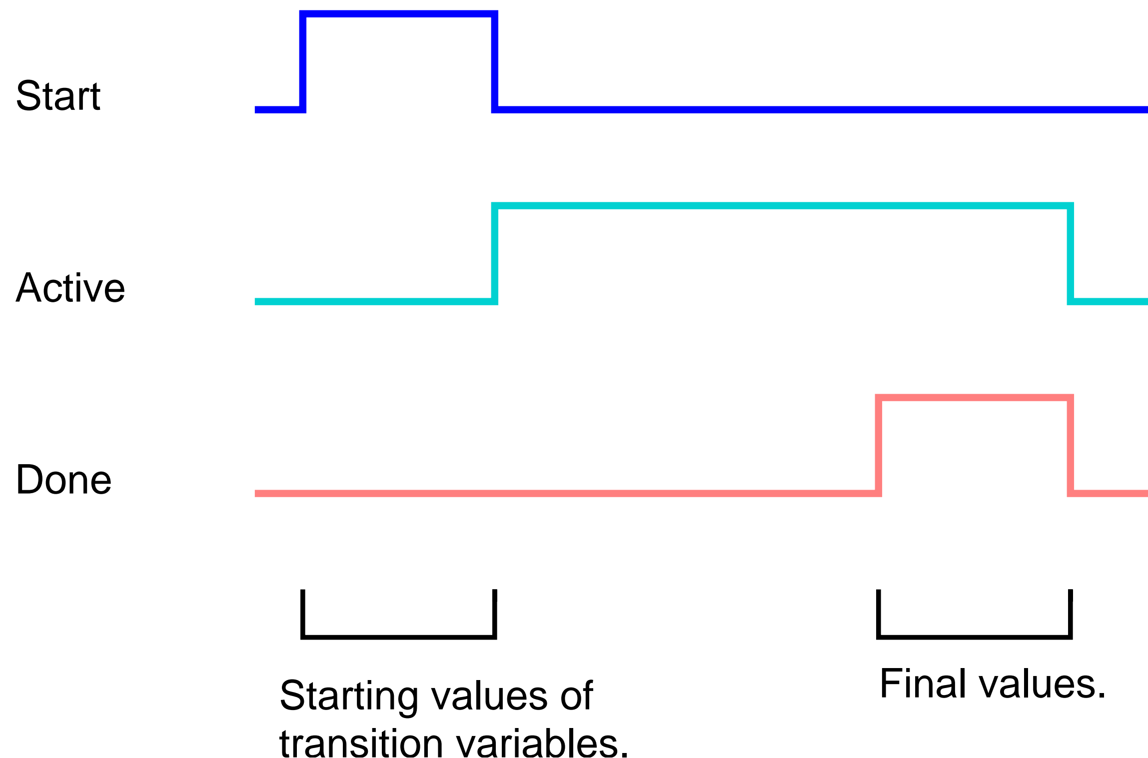
Hardware Protocol for Processes

Each process has three hardware signals that describe the state of the process.

start : Asserts that the process will start executing on the next state change.

active : Asserts that the process was active on the previous state change.

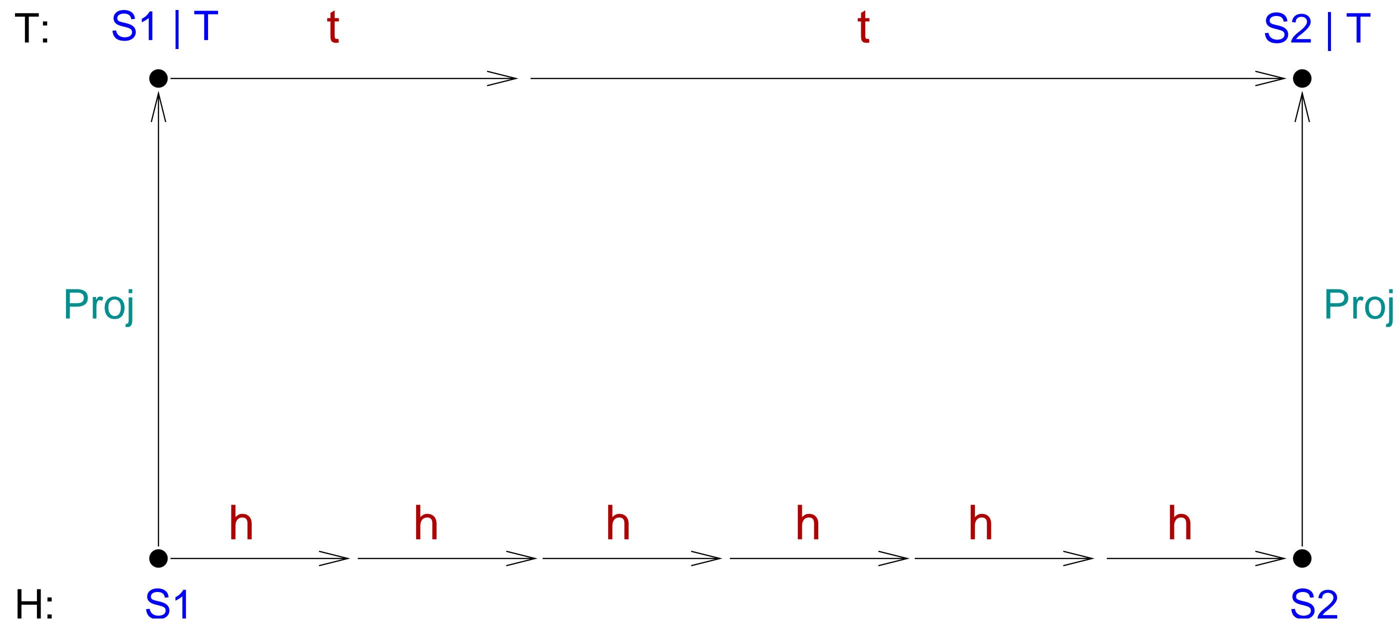
done : Asserts that the process finished executing on the previous state change.



Scheduling of Processes

1. If two processes are independent, they can be active at the same time.
2. If two processes are dependent and can be enabled in the same state, an arbiter decides which process to start.
3. If two processes are never enabled in the same state, then for this pair of processes, an arbiter is not needed to decide which process to start.

Implementation Relation



T is the specification transition system, transition function t

H is the hardware system, transition function h (a clock cycle)

var(T) $\subseteq$ var(H)

S1, S2 are hardware states where no process is active

S1|T, S2|T are specification states, projections of hardware

Proof Rules – Informal Idea

For each process, we show that the process implements a transition.

Rules to show that a process p implements a transition t:

1. Show $\mathsf{H} \models \mathsf{Start}_p \supset \mathsf{Enabled}_t$

2. Show that when process p is started, it always reaches the Done state in $\leq k$ steps, for some k.

3. Show

$$\mathsf{H} : \mathsf{S1} \stackrel{p}{\Longrightarrow} \mathsf{S2}$$

implies

$$\mathsf{T} : \mathsf{S1}|\mathsf{T} \stackrel{t}{\longrightarrow} \mathsf{S2}|\mathsf{T}$$

Summary of Proof Rules

We can use

- Hardware process structure
- Independence of concurrent processes
- Correspondence between processes and transitions

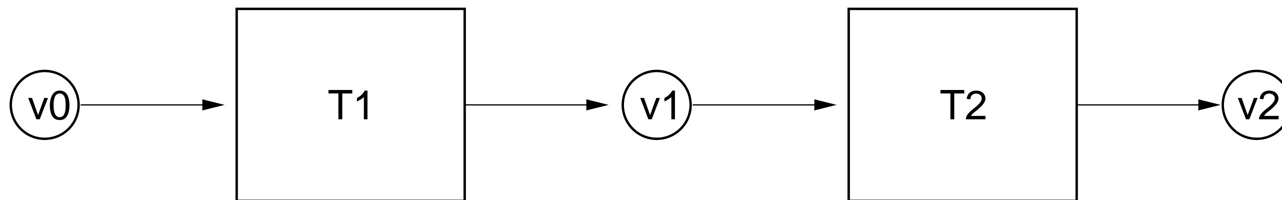
to provide an easy-to-use form of compositional reasoning.

Further development will require reasoning about refinement of actions.

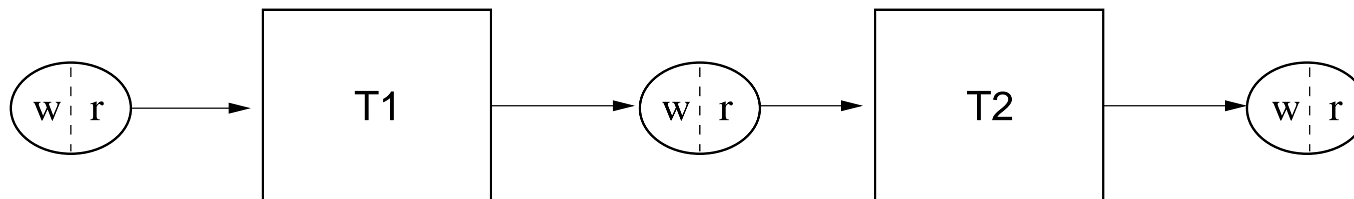
Pipelining

The initial definition of independence is too restrictive:

Two transitions cannot concurrently read and write a common variable in a pipeline.



We can write specifications for systems with pipeline concurrency,



and extend our rules to allow pipelined implementations of such specifications.

Summary of Tutorial

- Architecture-level modeling and verification can be effective in analyzing the soundness of protocol designs.
  - Model checking of architectural-level models can find problems that escape both random simulation and formal verification of small components.
- Formal models are being developed earlier in the design process.
- Formal Design approaches are emerging to guarantee the implementation conforms to a high-level model.

Appendix A. Tutorial Cache Memory Protocol Model

```
/* ---------------------------------------------------------------------- */
/* DOCUMENTATION                                                   */
/* ---------------------------------------------------------------------- */
/**@file

   Architecture-level protocol model by Steven German and Geert Janssen

   Steven German  4   June 2004 - original write-up (cachei.m)
   Geert Janssen  4 August 2004 - simplified and improved version
*/
/* ---------------------------------------------------------------------- */
/* CONSTANTS                                                              */
/* ---------------------------------------------------------------------- */

/**Constants that determine "size" of the problem. */
const num_nodes: 2; /**<number of processing nodes */
const num_addr:  1; /**<number of memory addresses  */
const num_data:  1; /**<number of data bits per address */

/* ---------------------------------------------------------------------- */
/* TYPES                                                 */
/* ---------------------------------------------------------------------- */

/**The communication channel ids. */
const channel1: 1;
const channel2: 2;
const channel3: 3;

/**Message types. */
type opcode: enum {
```

```
  op_invalid,-- clear
  read_shared,
  read_exclusive,
  req_upgrade,
  invalidate,
  invalidate_ack,
  grant_shared,
  grant_upgrade,
  grant_exclusive
};

/**Memory access message types. */
type request_opcode: enum {
  req_read_shared,-- clear
  req_read_exclusive,
  req_req_upgrade
};

/**Cache states. */
type cache_state: enum {
  cache_invalid,-- clear
  cache_shared,
  cache_exclusive
};

/**Address range. */
type addr_type: 0..num_addr-1; -- clear 0

/**Memory content at an address. */
type data_type: array[0..num_data-1] of boolean; -- clear [false,...]

/**A cache line. */
type cache_record:
  record
    state: cache_state; -- clear cache_invalid
```

```
    data:  data_type; -- clear [false,...]
  end;

/**Node id range. */
type node_id: 0..num_nodes-1; -- clear 0

/**Channel ids. */
type channel_id: 1..3; -- clear 1

/**A message. */
type message_type:
  record
    source: node_id; /**<originator, clear 0 */
    dest:   node_id; /**<destination, clear 0 */
    op:     opcode; /**<message type, clear op_invalid */
    addr:   addr_type; /**<address involved, clear 0 */
    data:   data_type; /**<data value, clear [false,...] */
  end;

/**Buffer for a message. */
type message_buf_type:
  record
    msg:   message_type; -- clear see message_type
    valid: boolean; -- clear false
  end;

/**Status of a request message. */
type status_type: enum {
  inactive,-- clear
  pending,
  completed
};

type addr_request_type:
  record
```

```
    --   addr: addr_type; NOTE requests are indexed by address
    home:   node_id; -- clear 0
    op:     opcode; -- clear op_invalid
    data:   data_type; -- clear [false,...]
    status: status_type; -- clear inactive
  end;

type line_dir_type: array[node_id] of cache_state; -- clear [cache_invalid,..]

type node_bool_array: array[node_id] of boolean; -- clear [false,...]

type home_request_type:
  record
    source: node_id; -- clear 0
    op:     opcode; -- clear op_invalid
    data:   data_type; -- clear [false,...]
    invalidate_list: node_bool_array; -- clear [false,...]
    status: status_type; -- clear inactive
  end;

type node_type:
  record
    memory:          array[addr_type] of data_type;
    cache:           array[addr_type] of cache_record;
    directory:       array[addr_type] of line_dir_type;
    local_requests:  array[addr_type] of boolean;
    home_requests:   array[addr_type] of home_request_type;
    remote_requests: array[addr_type] of addr_request_type;
    inchan:          array[channel_id] of message_buf_type;
    outchan:         array[channel_id] of message_buf_type;
  end;

/* ------------------------------------------------------------------------ */
/* VARIABLES                                                                */
/* ------------------------------------------------------------------------ */
```

```
/**Global state of the system. */
var node: array[node_id] of node_type;

/* ------------------------------------------------------------------------ */
/* FUNCTIONS                                                                */
/* ------------------------------------------------------------------------ */

/**Returns the id of the node that controls this memory address. */
function home_node(a: addr_type): node_id;
begin
  return a = 0 ? 0 : 1;
end;

/**Casts a requests code into a more general message op code. */
function req_opcode(req: request_opcode): opcode;
begin
  switch req
  case req_read_shared:    return read_shared;
  case req_read_exclusive: return read_exclusive;
  case req_req_upgrade:    return req_upgrade;
  endswitch;
end;

/**Returns smallest node id indexing a true entry in ‘a’. */
function first_node_in_vector(a: node_bool_array): node_id;
begin
  for i: node_id do
    if a[i] then
      return i;
    endif;
  endfor;
end;

/**Expresses the required invariant of the system:
```

```
   1. Can have at most 1 node having exclusive access to a certain address.
   2. If there's a node with exclusive access, no other may have shared access.
*/
function check_invariants(): boolean;
var n_exclusive, n_shared: 0..num_nodes;
begin
  for addr: addr_type do
    n_exclusive := 0;
    n_shared := 0;
    for n: node_id do
      switch node[n].cache[addr].state
      case cache_invalid:   -- do nothing
      case cache_shared:    n_shared    := n_shared + 1;
      case cache_exclusive: n_exclusive := n_exclusive + 1;
      endswitch;
    endfor;
    if n_exclusive > 1 then
      put "VIOLATION:\n\t[check_invariants]: two or more exclusive caches\n";
      return false;
    elsif n_exclusive > 0 & n_shared != 0 then
      put "VIOLATION:\n\t[check_invariants]: exclusive and shared caches\n";
      return false;
    endif;
  endfor;
  return true;
end;

/* Print opcode. Useful in debugging. */
procedure put_op_name(op:opcode);
begin
  switch op
  case op_invalid: put "invalid";
  case read_shared: put "shared";
  case read_exclusive: put "exclusive";
  case req_upgrade: put "upgrade";
```

```
  case invalidate: put "invalidate";
  case invalidate_ack: put "invalidate_ack";
  case grant_shared: put "grant_shared";
  case grant_upgrade: put "grant_upgrade";
  case grant_exclusive: put "grant_exclusive";
  endswitch;
end;

/* ------------------------------------------------------------------------ */
/* START STATE(S)                                                           */
/* ------------------------------------------------------------------------ */

/**Start state of the system; all data takes on its "smallest" value. */
startstate
  clear node;
end;

/* ------------------------------------------------------------------------ */
/* RULES      */
/* ------------------------------------------------------------------------ */

/************************************/
/* Message transfer between nodes   */
/************************************/

ruleset source: node_id; ch: channel_id do
  alias outchan: node[source].outchan[ch];
        dest:    outchan.msg.dest;
        inchan:  node[dest].inchan[ch] do
--
rule "Transfer message from ‘source’ via ‘ch’"
  -- must have valid message to transfer at source:
    outchan.valid
  -- must have empty message input buffer available at destination:
  & !inchan.valid  ==>
```

```
begin
  assert source = outchan.msg.source "message must come from source";
  assert outchan.msg.op != op_invalid "message type must be valid";
  inchan := outchan;
  -- now inchan.valid = true
  clear outchan;
  --outchan.valid := false; (implied by clear outchan, GJ04)
endrule;
endalias; endruleset;

/**********************************/
/* Node activities in "client" mode */
/**********************************/

ruleset client: node_id; req: request_opcode; addr: addr_type do
  alias cache_state: node[client].cache[addr].state;
        outchan:     node[client].outchan[channel1];
        local_request_for_addr: node[client].local_requests[addr];
        home:        home_node(addr) do
--
rule "‘client’ generates new ‘req’ for ‘addr’"
  -- can’t ask for same address if prior request hasn’t been completed:
    !local_request_for_addr
  -- can only ask for what we don’t already have:
  & ((req = req_read_shared    & cache_state = cache_invalid) |
     (req = req_read_exclusive & cache_state = cache_invalid) |
     (req = req_req_upgrade    & cache_state = cache_shared))
  -- must have an empty output channel message buffer:
  & !outchan.valid ==>
begin
  put ">> client ";
  put client;
  put " issues ";
  put_op_name(req_opcode(req));
  put " request for addr ";
```

```
  put addr;
  put "\n";

  alias msg: outchan.msg do
    -- Prepare the request message:
    msg.source := client;
    msg.dest   := home_node(addr);
    msg.op     := req_opcode(req);
    msg.addr   := addr;
    outchan.valid := true;
    -- Assure at most 1 request at a time:
    local_request_for_addr := true;
  endalias;
endrule;
endalias; endruleset;

ruleset client: node_id do
  alias inchan:  node[client].inchan[channel2];
        inmsg:   inchan.msg;
        addr:    inmsg.addr;
        request: node[client].remote_requests[addr] do
--
rule "‘client’ accepts invalidate request"
  -- must have valid data in the input message buffer:
    inchan.valid
  -- only processing invalidate type of messages:
  & inmsg.op = invalidate
  -- must be a unique request:
  & request.status = inactive ==>
begin
  assert inmsg.source = home_node(addr) "message source must be addr home";
  -- this indeed can happen for upgrade requests:
--  assert !node[client].local_requests[addr]
--  "client gets invalidation while has request pending";
  request.home   := inmsg.source;
```

```
  request.op     := inmsg.op /*= invalidate*/;
/*request.data   := don't care on input/filled in when ack;*/
  request.status := pending;
  clear inchan;
endrule;
endalias; endruleset;

ruleset client: node_id; addr: addr_type do
  alias request: node[client].remote_requests[addr] do
--
rule "`client' processes invalidate request for `addr'"
  -- must have a pending invalidation request:
    request.status = pending
  & request.op = invalidate ==>
begin
  put "<< client ";
  put client;
  put " invalidates cache for addr ";
  put addr;
  put "\n";

  alias cache_line: node[client].cache[addr] do
    -- currently cached data is saved in request message:
    request.data := cache_line.data;
    clear cache_line;
    -- Here: cache_line.state = cache_invalid
    request.status := completed;
  endalias;
endrule;
endalias; endruleset;

ruleset client: node_id; addr: addr_type do
  alias request: node[client].remote_requests[addr];
        outchan: node[client].outchan[channel3] do
--
```

```
rule "‘client’ prepares invalidate ack for ‘addr’"
  -- just completed a invalidation action:
    request.status = completed
  & request.op = invalidate
  -- have an empty output message buffer available:
  & !outchan.valid ==>
begin
  put ">> client ";
  put client;
  put " prepares invalidate ack for addr ";
  put addr;
  put "\n";

  assert request.home = home_node(addr) "dest must addr home";
  alias msg: outchan.msg do
    msg.op     := invalidate_ack;
    msg.source := client;
    msg.dest   := request.home;
    msg.data   := request.data;
    msg.addr   := addr;
    outchan.valid := true;
    clear request;
  endalias;
endrule;
endalias; endruleset;

ruleset client: node_id do
  alias inchan: node[client].inchan[channel2];
        msg:    inchan.msg;
        op:     msg.op;
        addr:   msg.addr;
        home:   home_node(addr);
        data:   msg.data do
--
rule "‘client’ receives reply from home"
```

```
  -- have valid data in message input buffer:
    inchan.valid
  -- message concerns grant:
  & (op = grant_shared | op = grant_upgrade | op = grant_exclusive) ==>
begin
  assert msg.source = home "source must match addr home";

  alias cache_line:    node[client].cache[addr];
        local_request_for_addr: node[client].local_requests[addr];
        state:         node[home].directory[addr][client] do
    -- update the cache, unless the request was handled locally by
    -- home = client

    put "<< client ";
    put client;
    put " receives ";
    put_op_name(op);
    put " for addr ";
    put addr;

    switch op
    case grant_shared:
      cache_line.data  := msg.data;
      cache_line.state := cache_shared;
    case grant_upgrade:
      -- data in cache is still valid.
      cache_line.state := cache_exclusive;
    case grant_exclusive:
      cache_line.data  := msg.data;
      cache_line.state := cache_exclusive;
    endswitch;
    assert state = cache_line.state
    "home directory record must reflect actual client state";
    -- indicate we can start issuing new requests:
    assert local_request_for_addr "must have local_request true";
```

```
    local_request_for_addr := false;
    clear inchan;
  endalias;
endrule;
endalias; endruleset;

/********************************/
/* Node activities in "home" mode */
/********************************/

ruleset home: node_id do
  alias inchan:  node[home].inchan[channel1];
        msg:     inchan.msg;
        valid:   inchan.valid;
        addr:    msg.addr;
        request: node[home].home_requests[addr] do
--
rule "‘home’ accepts a request message"
  -- have valid message input buffer:
    valid
  -- not handling another request for this address yet:
  & request.status = inactive ==>
  var b, invalidations: boolean;
  var k: node_id;
  var count: 0..num_nodes;
begin
  alias op:         msg.op;
        directory:  node[home].directory[addr];
        source:     msg.source;
        cache_line: node[home].cache[addr];
        memory:     node[home].memory[addr] do

  put "<< home ";
  put home;
  put " accepts ";
```

```
put_op_name(op);
put " request for ";
put addr;

-- In case of an upgrade request that got its cached data invalidated
-- by another request, we simply treat that upgrade as an exclusive:
if op = req_upgrade & directory[source] = cache_invalid then
  op := read_exclusive;
endif;

request.source := source;
request.op := op;

-- Now we decide how to process the request.

-- 1. If request is shared and cached shared locally,
-- just read from home cache, update directory, and send grant.
if op = read_shared & directory[home] = cache_shared then
  put "protocol rule 1: send shared from home cache\n";
  assert !exists n:node_id do directory[n] = cache_exclusive endexists
    "1. no exclusive cache";
  assert source != home "1. source cannot be home";
  -- get data from my/home cache:
  request.data   := cache_line.data;
  request.status := completed;

-- 2. If request is shared and not cached locally and directory shows
-- no exclusive copy, read from memory, send shared copy.
elsif op = read_shared & directory[home] = cache_invalid
    & !exists n:node_id do directory[n] = cache_exclusive endexists then
  put "protocol rule 2: send shared from memory\n";
  -- get data from memory:
  request.data   := memory;
  request.status := completed;
```

```
  -- 5. If request is shared and there is an exclusive copy
  -- then invalidate copy and get data from that cache.
  -- home could be exclusive!
  elsif op = read_shared &
exists n:node_id do directory[n] = cache_exclusive endexists then
    put "protocol rule 5: invalidate exclusive copy\n";
    -- determine who needs to be invalidated:
    count := 0;
    for n:node_id do
      b := directory[n] != cache_invalid;
      if b then
count := count + 1;
        k := n;
      endif;
      request.invalidate_list[n] := b;
    endfor;
    assert count = 1 "5. single non-invalid";
    assert directory[k] = cache_exclusive "5. must be exclusive";
    -- mark status of request as pending (must first do invalidation):
    put "protocol rule 5: pending...1 invalidation needed\n";
    request.status := pending;

  -- 6a. If request is upgrade and there are any other shared copies
  -- invalidate local+remote copies, send data exclusive to requestor.
  elsif (op = req_upgrade) then
    put "protocol rule 6a: upgrade request, invalidate any shared copies\n";
    assert directory[source] = node[source].cache[addr].state
      "6a. directory record must reflect actual source state";
    assert directory[source] = cache_shared
      "6a. directory must reflect source shared";
    -- see whether others (!= source) must be invalidated (no exclusive!):
    invalidations := false;
    for n: node_id do
      b := directory[n] != cache_invalid & n != source;
      invalidations := b | invalidations;
```

```
    request.invalidate_list[n] := b;
  endfor;
  -- A true upgrade: (shared) data in cache is still assumed valid.
  if invalidations then
    put "protocol rule 6a: pending...invalidations needed\n";
    request.status := pending;
  else
    request.status := completed;
  endif;

-- 6b. If request is exclusive and there are any cached copies
-- invalidate local+remote copies, send data exclusive to requestor.
elsif (op = read_exclusive) then
  put "protocol rule 6b: exclusive request, invalidate any copies\n";
  assert directory[source] = cache_invalid "6b. requestor must have invalid";
  -- determine if any and who needs to be invalidated
  -- source must be cache_invalid already
  invalidations := false;
  for n: node_id do
    b := directory[n] != cache_invalid;
    invalidations := b | invalidations;
    request.invalidate_list[n] := b;
  endfor;
  if invalidations then
    put "protocol rule 6b: pending...invalidations needed\n";
    -- data comes from a node that gets invalidated.
    request.status := pending;
  else
    put "protocol rule 6b: send exclusive from memory\n";
    request.data   := memory;
    request.status := completed;
  endif;

-- 7. If not one of the above cases, error.
else
```

```
    error "undefined case";
  endif;

  -- clear the request message
  clear inchan;

  endalias;
endrule;
endalias; endruleset;

ruleset home: node_id; addr: addr_type do
  alias request: node[home].home_requests[addr];
        out:     node[home].outchan[channel2];
        outmsg:  out.msg do
--
rule "‘home’ prepares invalidate for ‘addr’"
  -- request is still pending:
    request.status = pending
  -- we must have something to invalidate:
  & exists n:node_id do request.invalidate_list[n] endexists
  -- we have an empty message output buffer available:
  & !out.valid  ==>
  var dest: node_id;
begin
  outmsg.addr   := addr;
  outmsg.op     := invalidate;
  outmsg.source := home;
  dest := first_node_in_vector(request.invalidate_list);
  outmsg.dest   := dest;
  out.valid := true;
  request.invalidate_list[dest] := false;
endrule;
endalias; endruleset;

ruleset home: node_id do
```

```
  alias inchan:    node[home].inchan[channel3];
        inmsg:     inchan.msg;
        addr:      inmsg.addr;
        request:   node[home].home_requests[addr];
        source:    inmsg.source;
        directory: node[home].directory[addr] do
--
rule "‘home’ processes invalidate ack"
  -- we have a valid message input buffer:
    inchan.valid
  -- request is still pending:
  & request.status = pending
  -- message acknowledges an earlier invalidate request:
  & inmsg.op = invalidate_ack ==>
begin
  put "<< home ";
  put home;
  put " processes invalidate ack for ";
  put addr;

  -- if invalidating exclusive cache, save data in request and memory
  if directory[source] = cache_exclusive then
    -- write message data back to memory:
    node[home].memory[addr] := inmsg.data;
  endif;
  /* Even if just shared must get data to requestor GJ04 */
  -- same data will be send to requestor:
  request.data := inmsg.data;

  assert node[source].cache[addr].state = cache_invalid
  "source must have invalid cache";
  directory[source] := cache_invalid;

  clear inchan;
  -- try to mark request as completed:
```

```
  switch request.op
    case read_shared:
      -- just invalidated an exclusive; done.
      request.status := completed;
    case req_upgrade:
      if forall n: node_id do
        n != request.source -> directory[n] = cache_invalid endforall then
        request.status := completed;
      endif;
    case read_exclusive:
      if forall n: node_id do directory[n] = cache_invalid endforall then
        request.status := completed;
      endif;
    else error "unexpected request opcode";
  endswitch;
endrule;
endalias; endruleset;

ruleset home: node_id; addr: addr_type do
  alias request: node[home].home_requests[addr];
        outchan: node[home].outchan[channel2];
        msg:     outchan.msg do
--
rule "‘home’ sends grant for ‘addr’"
    request.status = completed
  & !outchan.valid ==>
begin
  msg.source := home;
  msg.dest   := request.source;
  switch request.op
  case read_shared:    msg.op := grant_shared;
    node[home].directory[addr][request.source] := cache_shared;
  case req_upgrade:    msg.op := grant_upgrade;
    node[home].directory[addr][request.source] := cache_exclusive;
  case read_exclusive: msg.op := grant_exclusive;
```

```
    node[home].directory[addr][request.source] := cache_exclusive;
  endswitch;
  msg.data := request.data;
  msg.addr := addr;
  clear request;
  outchan.valid := true;
endrule;
endalias; endruleset;

/* ------------------------------------------------------------------------- */
/* CHECKS      */
/* ------------------------------------------------------------------------- */

invariant check_invariants();
```