**RC23958 (W0605-092) May 17, 2006**
**Computer Science**

# IBM Research Report

## A Tutorial Example of a Cache Memory Protocol and RTL Implementation

**Steven German, Geert Janssen**
IBM Research Division
Thomas J. Watson Research Center
P.O. Box 218
Yorktown Heights, NY 10598

# A Tutorial Example of a Cache Memory Protocol and RTL Implementation

Steven German and Geert Janssen

IBM T.J. Watson Research Center
Yorktown Heights, NY 10598

May 2006

**Abstract.** This technical report is intended to facilitate research on the problem of showing that a hardware implementation is a correct implementation of a higher level model. We provide a tutorial example of a cache memory protocol and an implementation of the protocol in VHDL. Cache memory protocols are an important subject for formal verification, because these protocols are notoriously difficult to design and implement correctly. Although there is a considerable body of theory about verifying that an implementation is correct with respect to a high level model, such verifications are rare in industry. By providing realistic examples of both a high level model and its implementation, we hope to encourage the development of practical verification methods that can be used in industry.

**Keywords:** formal verification of hardware, refinement checking, cache memory protocol

## 1 Introduction

The most common approaches to formal verification of hardware are to verify either an RTL implementation or an abstract model of the implementation. Both of these approaches leave gaps in the verification. Implementations of complex hardware designs are too complex to be completely verified at the RTL. In particular, it usually not possible to verify global system properties at this level. On the other hand, abstract models make it possible to verify global properties, but there is no guarantee that an abstract model correctly represents the implementation.

A solution to the verification gap problem would be to first develop and verify an abstract model of the design, and then verify that the design conforms to the abstract model.

One notion of conformance between an abstract model and an implementation is *refinement*. An implementation refines an abstract model iff every sequence of states forming an execution of the implementation, when projected onto the states of the abstract model, is an execution of the abstract model.

Although there is a considerable body of research about proving refinement relations, in practice refinement is rarely checked in industry. Previous methods are difficult to apply in an industrial setting. Also, much of the previous work is directed at checking refinement for specific classes of designs, especially instruction processor pipelines. In contrast, we are interested in developing methods that can be applied to designs such as protocol processing engines.

In our IBM SSI Project, we have started to develop a new approach for checking refinement. Our new approach combines design principles that make the implementation easier to verify with a new method for verifying refinement [2]. Based on the work we have done so far, we believe that practical methods can be developed for routine use in industry.

To speed the development of practical refinement checking methods, it is essential to collaborate with university researchers. University researchers need to have access to realistic examples of hardware designs and specifications if they are to contribute in this area. This report aims to provide one useful example.

In this report, we provide a realistic example in the form of a cache memory protocol and its implementation in VHDL. In Section 1, we discuss the origin of the protocol model and the source material that it is based on. The operation of the protocol is outlined in Section 3, and the implementation is outlined in Section 4. The full text of the protocol model and the implementation are contained in the Appendix. The protcol model has been previously published as part of a Tutorial presented at Formal Methods in Computer-Aided Design, FMCAD 2004 [2].

## 2 Origin and Sources of the Protocol Model

The protocol model is based on an earlier tutorial example that was published in 2000, called the German 2000 protocol. The 2000 example is a highly simplified Murphi model of a directory-based cache coherence protocol. The 2000 example consists of about 130 lines in the Murphi modeling language. This model has been widely studied and used as an example in the research community.

The Murphi model in this technical report has been previously published as part of a Tutorial on Verification of Distributed Cache Memory Protocols that was presented at FMCAD 2004. The 2004 model was derived from the German 2000 model by adding some details that were missing from the 2000 model but are typical of actual protocols. Specifically, the following features were added to derive the 2004 model:

- Each node acts as both home and client
- Message processing occurs in phases: Receive message, Process, Send reply
- Protocol allows more that one memory address
- More than a single request can be active at a node at same time
- Memory hierarchy: the model includes Main Memory and Cache

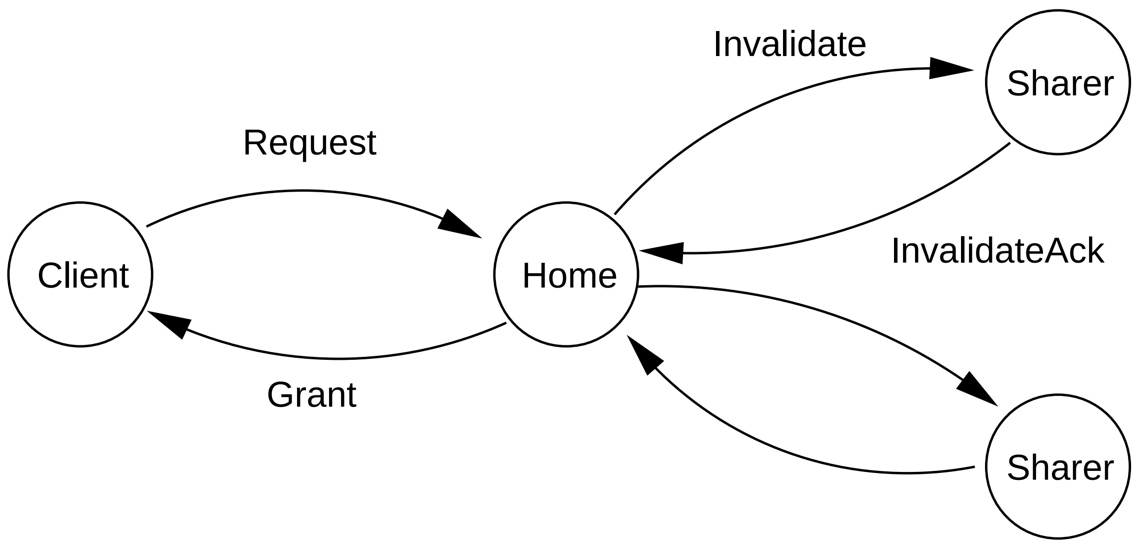


Figure 1: Message Flow

- Protocol shows movement of data
- Nodes can send messages to themselves (Request, Invalidate)

The additions made to derive the 2004 protocol are quite generic and are not based on the design of any actual commerical protocol.

The VHDL implementation presented in this technical report has not been published previously. The VHDL implementation is a quite straightforward rendering of the 2004 protocol model. The VHDL that we have developed is simplistic compared to actual implementations of commercial cache coherence protocols. Our VHDL model is not derived in any way from the design of a commercial protocol. Nor have we attempted to incorporate the characteristics of commercial designs in our VHDL. We do feel that the example presented here is a useful and challenging problem for its stated purpose, which is research on formal verification.

## 3 Abstract Model of Cache Memory Protocol

In the protocol, there are a number of nodes, each having a cache. The cache is used to satisfy memory requests for processors located at the node. The processors are not modeled as part of this protocol. A cache line in the cache can be in one of the states invalid, shared, or exclusive.

Processors can issue requests for a shared copy, an exclusive copy, or an upgrade of a copy of a cache line from shared to exclusive. The basic operation of memory requests is shown in Figure 1.

Each node controls a range of addresses and has a portion of main memory for storing the data at these addresses. The node that controls a given address is called the *home* node for the address. A

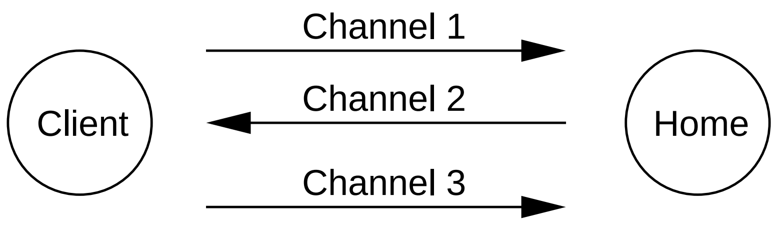


Figure 2: Channel Assignments

node making a request is called a *client*. Other nodes that have cached copies of the data are called *sharers*.

A client makes a request by sending a request message to the home node. The home node performs the necessary coherence actions to satisfy the request. In certain cases, the home node has to invalidate all of the copies of the data cached at sharer nodes. The home node has a directory that indicates which nodes have copies of the data, and in which cache state. The home node invalidates other caches by sending them invalidate messages. When a sharer node receives an invalidate message, it invalidates the line in its cache, and send an invalidate acknowledgement back to the home node. When the home node has received all of the necessary invalidate acknowledgements, it sends a grant message containing the data to the requesting client. Then the home node is ready to start processing another request.

The text of the abstract model is shown in Appendix A. The model is written in the language of the Murphi Verifier [1].

In the model, we define three communication channels between each pair of nodes (see Figure 2). Each channel is a fifo queue. Channel 1 carries requests from client nodes to the home node. Channel 2 carries invalidate requests and grants from the home node. Channel 3 carries invalidate acknowledgements to the home node. The reason for having several channels is to prevent message deadlocks. The protocol uses fact that both the invalidate requests and grants are carried on the same fifo channel to help maintain coherence.

The structure of a node is defined in the model by the type `node_type`. A node in the model contains a memory, a cache, and a directory. A node contains three tables for processing requests:

- `local_requests`: status of requests originated by this node
- `home_requests`: status of requests for which this node is the home
- `remote_requests`: status of requests for which this node is a sharer

A node also contains a set of input and output channels.

The processing steps of the protocol are defined by ten sets of rules. We briefly describe each of the rules.

1. Transfer message from source via channel.

Rule 1 transfers a message from an output buffer at one node to the input buffer at the node marked as the destination of the message.

2. Client generates new request for addr.

Rule 2 models the generation of new memory requests. The valid requests depend on the state of the cache. If the cache is invalid, a node can request either a shared or exclusive copy, while if the node has a shared copy, it can only request an upgrade.

3. Client accepts invalidate request.
4. Client processes invalidate request for addr.
5. Client prepares invalidate ack for addr.

Rules 3–5 describe the steps in processing an invalidate request. First, the client receives and stores the request, then the client invalidates its cache, and finally the client prepared an acknowledgement message.

6. Client receives reply from home.

Rule 6 describes the action of a client node on receiving a grant message from the home node.

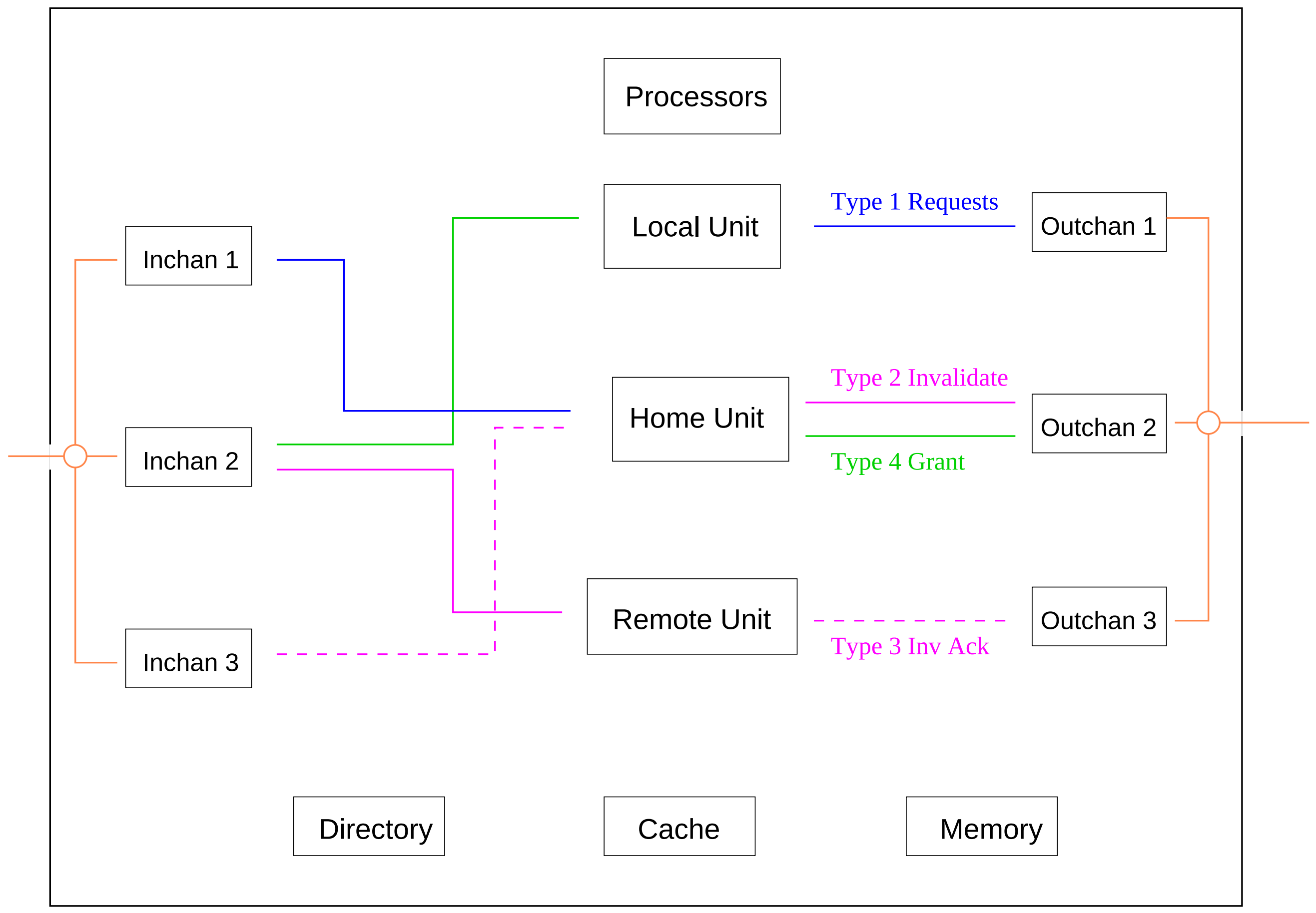
7. Home accepts a request message.
8. Home prepares invalidate for addr.
9. Home processes invalidate ack.
10. Home sends grant for addr.

Rules 7–10 describe how the home node processes requests. Rule 7 is the most complicated rule in this model. When the home receives a new request, rule 7 examines the home node's directory, and decides what actions need to be taken to process the request. The status of the request is stored in `home_requests`. The rules 8–10 perform additional actions depending on the data in `home_requests`.

## 4 Implementation in VHDL

The structure of a node and the flow of messages in shown in Figure 2. A node contains three protocol processing units: the local unit, home unit, and remote unit. A node also contains a directory, a cache, and a memory.

## Structure of a Node



The processing steps of a request are as follows. The local unit generates new coherence requests. New requests are placed in outchan 1, and transferred to the home node. At the nome node, new requests arrive on inchan 1, and are transferred to the home unit. The home unit accesses the directory, cache and memory to perform coherence actions. If the home needs to invalidate copies of the data in other nodes, it generates a series of invalidate messages, which are sent on outchan 2. An invalidate message arrives at a node on inchan 2, and is transferred to the remote unit. The remote unit invalidates the data in cache, and sends an invalidate acknowledgement on outchan 3. Invalidate acknowledgements return to the home node on inchan 3, and are read by the home unit. When the processing of a request is complete, the home unit generates a grant message, and sends it using outchan 2. Finally, the requesting client receives the grant on inchan 2, and the grant is read by the local unit. The local unit updates its cache.

The VHDL implementation of the protocol is listed in Appendix B. The implementation consists of several VHDL modules. The VHDL modules are as follows:

**Buf.vhdl** This module implements a message buffer. The message buffers are used for the input and output channels of a node.

**Cachei_pkg.vhdl** This module defines the basic data types used to implement nodes, and defines a number of utility functions.

**Cache.vhdl** This module defines a cache with several ports. Within a node, each of the processing units has an assigned port to access the cache.

Since all of the processing units can be active at the same time, a unit may have to wait several clock cycles to access the cache. The waiting implies that when a unit is executing an action corresponding to a rule in the Murphi model, the unit may be interrupted by another unit in the same node. This creates an interesting verification problem.

Note to Geert: as pointed out by Lv Yi, we need to modify the home unit to access the cache. I made this change in the Murphi model, but we need to update the VHDL.

**Home.vhdl** This module implements the home unit, corresponding to Murphi model rules 7–10.

**Local.vhdl** This module implements the local unit, Murphi rules 2 and 6.

**Node.vhdl** This module is the top level structure for a node. It instantiates all of the components of a node.

**Router.vhdl** This module transfers messages from output buffers to the destination input buffer on the same channel. Message transfer corresponds to Murphi rule 1.

**System.vhdl** This module is the top-level of the implementation. It instantiates a number of nodes and a router.

## Acknowledgement

The authors give special thanks to John Darringer for encouraging and supporting this work.

## References

[1] D.L. Dill, A.J. Drexler, A.J. Hu, and C.H. Yang, Protocol Verification as a Hardware Design Aid, in *Proceedings of IEEE International Conference on Computer Design: VLSI in Computers and Processors*, 1992. tion

[2] S.M. German, Tutorial on Verification of Distributed Cache Memory Protocols, presented at Formal Methods in Computer-Aided Design FMCAD 2004.

# Appendix A. Tutorial Cache Memory Protocol Model

```
/* Copyright (c) 2004 International Business Machines Corporation */

/* ------------------------------------------------------------------------ */
/* 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 "1. 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 "2. ‘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 "3. ‘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 "4. ‘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 "5. ‘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 "6. ‘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 "7. ‘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";

  --Modified by Yi Lv, get data from home memory, not from home cache.
  --Modified by SMG 9 Jan 06: get data from cache if cache_line.state shows
  -- data is present, otherwise get data from memory.
    If cache_line.state = cache_shared
    Then request.data := cache_line.data;
         assert cache_line.data[0] = memory[0]
          "cache is consistent with memory";
    Else  request.data   := memory;
    Endif;

    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 "8. ‘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 "9. ‘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 "10. ‘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();
```

# Appendix B. Tutorial Cache Memory Protocol Model

## B.1 Buf.vhdl

```
-- Copyright (C) 2004-2005 Geert Janssen - IBM
library ieee;
use ieee.std_logic_1164.all;

use work.cachei_pkg.all;

-- Simple message buffer with synchronous clear signal.
-- Accepts data when valid state is false and then only when accept is true.
-- reset will clear buffer and also clear valid state.
-- initial state of buffer is empty and invalid.
entity buf is
  port (default_clock: in std_ulogic;
        data_in: in message_type;
        data_out: out message_type;
        accept: in boolean;
        valid: out boolean;
        reset: in boolean
        );
end buf;

architecture behavior of buf is
  signal msg_buf: message_buf_type := chan_0;
  signal n_msg: message_type;
  signal n_valid: boolean;
  signal update: boolean;
begin
  update  <= accept and not msg_buf.valid;
  n_msg   <= msg_0 when reset else data_in when update else msg_buf.msg;
  n_valid <= false when reset else true    when update else msg_buf.valid;

  process
  begin
    wait until default_clock ='1';

    assert not reset or msg_buf.valid
      report "Spurious reset of input buf" severity warning;

    -- state updates:
    msg_buf.msg   <= n_msg;
    msg_buf.valid <= n_valid;
  end process;

  valid    <= msg_buf.valid;
  data_out <= msg_buf.msg;

end behavior;
```

## B.2 Cachei_pkg.vhdl

```
-- Copyright (C) 2004-2005 Geert Janssen - IBM

library ieee;
use ieee.std_logic_1164.all;

package cachei_pkg is

-- Constants that determine "size" of the problem.
constant num_nodes : positive := 2; -- number of processing nodes
constant num_addr  : positive := 2; -- number of addresses of the memory
constant num_data  : positive := 1; -- number of data bits per address
-- The communication channel ids.
constant channel1  : positive := 1;
constant channel2  : positive := 2;
constant channel3  : positive := 3;

-- Message types.
type opcode is
(
  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 is
(
  req_read_shared,                     -- clear
  req_read_exclusive,
  req_req_upgrade
);

-- Cache states.
type cache_state is
(
  cache_invalid,                       -- clear
  cache_shared,
  cache_exclusive
);
```

```
-- Address range.
subtype addr_type is natural range 0 to num_addr-1; -- clear 0

-- Memory content at an address.
subtype data_type is std_ulogic_vector(0 to num_data-1); -- clear data_0

-- A cache line.
type cache_record is
  record
    state: cache_state;                 -- clear cache_invalid
    data:  data_type;                   -- clear data_0
  end record;

type cache_record_array is array(natural range <>) of cache_record;
type addr_type_array is array(natural range <>) of addr_type;

-- Node id range.
subtype node_id is natural range 0 to num_nodes-1; -- clear 0

-- Channel ids.
subtype channel_id is positive range 1 to 3; -- clear 1

-- A message.
type message_type is
  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 data_0
  end record;

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

-- Status of a request message.
type status_type is
(
  inactive,                             -- clear
  pending,
  completed
);
```

```
-- Generic vector of booleans.  -- clear (others => false)
type boolean_array is array(natural range <>) of boolean;

-- Boolean vector, one entry per node.
subtype node_bool_array is boolean_array(node_id); -- clear (others => false)

-- Access request to be handled by this (home, owner of address) node.
type home_request_type is
  record
    source: node_id;                   -- clear 0
    op:     opcode;                    -- clear op_invalid
    data:   data_type;                 -- clear data_0
    invalidate_list: node_bool_array;  -- clear (others => false)
    status: status_type;               -- clear inactive
  end record;

type addr_request_type is
  record
    --   addr: addr_type; NOTE requests are indexed by address
    home:   node_id; -- clear 0
    op:     opcode; -- clear op_invalid
    data:   data_type; -- clear data_0
    status: status_type; -- clear inactive
  end record;

-- Directory of cache states of the nodes.
type line_dir_type is array(node_id) of cache_state; -- clear (others => cache_invalid)

-- The memory.
type memory_type is          array(addr_type) of data_type;
-- The cache.
type cache_type is           array(addr_type) of cache_record;
-- Directory of cache states of the nodes per address.
type directory_type is       array(addr_type) of line_dir_type;

subtype local_requests_type is boolean_array(addr_type);

type home_requests_type is   array(addr_type) of home_request_type;

type remote_requests_type is array(addr_type) of addr_request_type;
-- The message channels.
type channel_type is         array(channel_id) of message_buf_type;

type io_fabric is            array(node_id) of channel_type;

type valid_channel_type is   array(channel_id) of boolean;
type valid_node_type is      array(node_id) of valid_channel_type;

type reset_channel_type is   array(channel_id) of boolean;

type reset_node_type is      array(node_id) of reset_channel_type;

type cache_node_type is      array(node_id) of cache_type;

type node_type is
  record
    memory:          memory_type;
    cache:           cache_type;
    directory:       directory_type;
    local_requests:  local_requests_type;
    home_requests:   home_requests_type;
    remote_requests: remote_requests_type;
    inchan:          channel_type;
    outchan:         channel_type;
  end record;

-- Global state of the system.
type the_nodes is array(node_id) of node_type;

function home_node(a:addr_type) return node_id;
function req_opcode(req: request_opcode) return opcode;
function first_node_in_vector(a: node_bool_array) return node_id;
function check_invariants(caches: cache_node_type) return boolean;

constant data_0: data_type := (others => '0');
constant cache_0: cache_record := (cache_invalid, data_0);
constant msg_0: message_type := (0, 0, op_invalid, 0, data_0);
constant chan_0: message_buf_type := (msg_0, false);
constant remote_req_0: addr_request_type :=
  (0, op_invalid, data_0, inactive);
constant home_req_0: home_request_type :=
  (0, op_invalid, data_0, (others => false), inactive);

function exists(bools: boolean_array) return boolean;
function exists_exclusive(states: line_dir_type) return boolean;
function all_invalid(states: line_dir_type) return boolean;
function all_but_1_invalid(states: line_dir_type; k: node_id) return boolean;

end cachei_pkg;

package body cachei_pkg is
```

```
-- Returns the id of the node that controls this memory address.
function home_node(a:addr_type) return node_id is
begin
  if a = 0 then
    return 0;
  else
    return 1;
  end if;
end home_node;

-- Casts a requests code into a more general message op code.
function req_opcode(req: request_opcode) return opcode is
begin
  case req is
    when req_read_shared    => return read_shared;
    when req_read_exclusive => return read_exclusive;
    when req_req_upgrade    => return req_upgrade;
    when others => null;
  end case;
end req_opcode;

-- Returns smallest node id indexing a true entry in ‘a’.
function first_node_in_vector(a: node_bool_array) return node_id is
  variable i: node_id;
begin
  for i in node_id’left to node_id’right loop
    if a(i) then
      return i;
    end if;
  end loop;
end first_node_in_vector;

-- 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(caches: cache_node_type) return boolean is
  variable n_exclusive, n_shared: natural range 0 to num_nodes;
  variable addr : addr_type;
  variable n : node_id;
begin
 for addr in addr_type’left to addr_type’right loop
   n_exclusive := 0;
   n_shared := 0;
   for n in node_id’left to node_id’right loop
     case caches(n)(addr).state is
       when cache_invalid   => null;
       when cache_shared    => n_shared    := n_shared + 1;
       when cache_exclusive => n_exclusive := n_exclusive + 1;
     end case;
   end loop;
   if n_exclusive > 1 then
     --put "two exclusive caches";
     return false;
   elsif n_exclusive > 0 and n_shared /= 0 then
     --put "exclusive and shared";
     return false;
   end if;
 end loop;
 return true;
end;

function exists(bools: boolean_array) return boolean is
  variable i: natural;
begin
  for i in bools'left to bools'right loop
    if bools(i) then
      return true;
    end if;
  end loop;
  return false;
end exists;

function exists_exclusive(states: line_dir_type) return boolean is
  variable n: node_id;
begin
  for n in node_id'left to node_id'right loop
    if states(n) = cache_exclusive then
      return true;
    end if;
  end loop;
  return false;
end exists_exclusive;

function all_invalid(states: line_dir_type) return boolean is
  variable n: node_id;
begin
  for n in node_id'left to node_id'right loop
    if states(n) /= cache_invalid then
      return false;
    end if;
  end loop;
  return true;
end all_invalid;
```

```
function all_but_1_invalid(states: line_dir_type; k: node_id) return boolean is
  variable n: node_id;
begin
  for n in node_id'left to node_id'right loop
    if n /= k and states(n) /= cache_invalid then
      return false;
    end if;
  end loop;
  return true;
end all_but_1_invalid;

end package body cachei_pkg;
```

## B.3 Cache.vhdl

```
-- Copyright (C) 2004-2005 Geert Janssen - IBM

-- Inspired by Reinaldo Bergamaschi’s version

library ieee;
use ieee.std_logic_1164.all;

use work.cachei_pkg.all;

entity cache is
  generic (n: natural := 2);
  port (
        clock: in std_ulogic;
        datain: in cache_record_array(0 to n-1);
        address: in addr_type_array(0 to n-1);
        valid: in boolean_array(0 to n-1);
        ack: out boolean_array(0 to n-1);
        cacheout: out cache_type
       );
end;

architecture behavior of cache is
  signal cache_mem: cache_type := (others => cache_0);
begin
  cacheout <= cache_mem;

  process
    variable turn: natural := 0;
    variable found: boolean;
  begin  -- process
    wait until clock = ’1’;

    found := false;
    ack <= (others => false);

    for i in 0 to n-1 loop
      if valid(turn) then
--        report "cache: Writing a cache line";
        ack(turn) <= true;
        cache_mem(address(turn)) <= datain(turn);
        found := true;
      end if;

      if turn = n-1 then
        turn := 0;
      else
        turn := turn + 1;
      end if;

      if found then
        exit;
      end if;
    end loop;

  end process;

end behavior;
```

## B.4 Home.vhdl

```
-- Copyright (C) 2004-2005 Geert Janssen - IBM

library ieee;
use ieee.std_logic_1164.all;

use work.cachei_pkg.all;

-- home mode
-- reads from in channel 1 and 3
-- writes to out channel 2
entity home is
  generic (self: node_id);
  port (
    default_clock: in std_ulogic;
    cache: in  cache_type;
    in1:   in  message_buf_type;
    reset_in1: out boolean;
    in3:   in  message_buf_type;
    reset_in3: out boolean;
    out2:  out message_buf_type;
    reset_out2: in boolean
    );
end home;

architecture behavior of home is
  signal buf2: message_buf_type := chan_0;
  signal home_requests: home_requests_type := (others => home_req_0);
  signal home_requests_next: home_requests_type := (others => home_req_0);
begin
  out2 <= buf2;

  reset_in1 <= in1.valid and home_requests(in1.msg.addr).status = inactive;

  reset_in3 <= in3.valid and home_requests(in3.msg.addr).status = pending;

  process
    variable memory:          memory_type := (others => data_0);
    variable directory:       directory_type
      := (others => (others => cache_invalid));

    variable dest: node_id;
    variable b, invalidations: boolean;
    variable real_in1_op: opcode;
  begin
    wait until default_clock = '1';
    home_requests <= home_requests_next;

    if reset_out2 then
      assert buf2.valid
        report "Spurious reset of output buffer 2" severity warning;
      buf2 <= chan_0;
    elsif not buf2.valid then
      -- rule "‘home’ prepares invalidate for ‘addr’"
      for addr in addr_type loop
        if home_requests(addr).status = pending then
          if exists(home_requests(addr).invalidate_list) then
            buf2.msg.addr   <= addr;
            buf2.msg.op     <= invalidate;
            buf2.msg.source <= self;
            dest := first_node_in_vector(home_requests(addr).invalidate_list);
            buf2.msg.dest   <= dest;
            buf2.valid  <= true;
            home_requests_next(addr).invalidate_list(dest) <= false;
--            report "home: Prepared invalidate req in output buffer 2";
            exit;
          end if;
        elsif home_requests(addr).status = completed then
          -- rule "‘home’ sends grant for ‘addr’"
          buf2.msg.source <= self;
          buf2.msg.dest   <= home_requests(addr).source;

          case home_requests(addr).op is
            when read_shared =>
              buf2.msg.op <= grant_shared;
              directory(addr)(home_requests(addr).source) := cache_shared;
            when req_upgrade =>
              buf2.msg.op <= grant_exclusive;
              directory(addr)(home_requests(addr).source) := cache_exclusive;
            when read_exclusive =>
              buf2.msg.op <= grant_exclusive;
              directory(addr)(home_requests(addr).source) := cache_exclusive;
            when others =>
              report "home: Unexpected request" severity failure;
              null;
          end case;

          buf2.msg.data <= home_requests(addr).data;
          buf2.msg.addr <= addr;
          home_requests_next(addr) <= home_req_0;
          buf2.valid <= true;
--          report "home: Prepared grant in output buffer 2";
```

```
        exit;
      end if;
    end loop;
  end if;

--------------------------------------------------------------------------------

  -- rule "‘home’ accepts a request message"
  if in1.valid then
    if home_requests(in1.msg.addr).status = inactive then
--        report "home: Accept request from input buffer 1";

      -- In case of an upgrade request that got its cached data invalidated
      -- by another request, we simply treat that upgrade as an exclusive:
      if in1.msg.op = req_upgrade
        and directory(in1.msg.addr)(in1.msg.source) = cache_invalid then
        real_in1_op := read_exclusive;
      else
        real_in1_op := in1.msg.op;
      end if;

      home_requests_next(in1.msg.addr).source <= in1.msg.source;
      home_requests_next(in1.msg.addr).op     <= real_in1_op;

      -- Now we decide how to process the request.

      if real_in1_op = read_shared then
        if directory(in1.msg.addr)(self) = cache_shared then
          -- 1. request is shared and cached shared locally,
          -- just read from our cache, update directory, and send grant.

          -- get data from my/home cache:
          home_requests_next(in1.msg.addr).data   <= cache(in1.msg.addr).data;
          home_requests_next(in1.msg.addr).status <= completed;
--          report "Case 1";
        elsif directory(in1.msg.addr)(self) = cache_invalid
          and not exists_exclusive(directory(in1.msg.addr)) then
          -- 2. request is shared and not cached locally and directory shows
          -- no exclusive copy, then read from memory, send shared copy.

          -- get data from memory:
          home_requests_next(in1.msg.addr).data   <= memory(in1.msg.addr);
          home_requests_next(in1.msg.addr).status <= completed;
--          report "Case 2";
        elsif exists_exclusive(directory(in1.msg.addr)) then
          -- 5. request is shared and there is an exclusive copy
          -- then invalidate that copy and get data from that cache.
```

```
            -- home (self) could be the exclusive one!

            -- determine who needs to be invalidated:
            invalidations := false;
            for n in node_id loop
              b := directory(in1.msg.addr)(n) /= cache_invalid;
              invalidations := b or invalidations;
              home_requests_next(in1.msg.addr).invalidate_list(n) <= b;
            end loop;
            assert invalidations report "home: Inconsistency" severity failure;
            home_requests_next(in1.msg.addr).status <= pending;
--            report "Case 5";
          else
            report "home: Unexpected case (read_shared)" severity failure;
          end if;
        elsif real_in1_op = req_upgrade then
          -- 6a. request is upgrade and there are any other shared copies
          -- invalidate local+remote copies, send data exclusive to requestor.

          -- see whether others (!= source) must be invalidated (no exclusive!)
          invalidations := false;
          for n in node_id'left to node_id'right loop
            b := directory(in1.msg.addr)(n) /= cache_invalid
                 and n /= in1.msg.source;
            invalidations := b or invalidations;
            home_requests_next(in1.msg.addr).invalidate_list(n) <= b;
          end loop;
          -- A true upgrade: (shared) data in cache is still assumed valid.
          if invalidations then
            home_requests_next(in1.msg.addr).status <= pending;
--            report "Case 6a pending";
          else
            home_requests_next(in1.msg.addr).status <= completed;
--            report "Case 6a completed";
          end if;
        elsif real_in1_op = read_exclusive then
          -- 6b. If request is exclusive and there are any cached copies
          -- invalidate local+remote copies, send data exclusive to requestor.

          -- determine if any and who needs to be invalidated
          -- source must be cache_invalid already
          invalidations := false;
          for n in node_id'left to node_id'right loop
            b := directory(in1.msg.addr)(n) /= cache_invalid;
            invalidations := b or invalidations;
            home_requests_next(in1.msg.addr).invalidate_list(n) <= b;
          end loop;
          if invalidations then
            -- data comes from a node that gets invalidated.
            home_requests_next(in1.msg.addr).status <= pending;
          else
            home_requests_next(in1.msg.addr).data   <= memory(in1.msg.addr);
            home_requests_next(in1.msg.addr).status <= completed;
          end if;
--          report "Case 6b";
        else
          -- 7. If not one of the above cases, error.
          report "Case 7 in home mode" severity failure;
        end if;

        -- clear the request message
--        reset_in1 <= true;
      else
--        report "home: Message waiting in buf 1";
--        reset_in1 <= false;
      end if;
    else
--      reset_in1 <= false;
    end if;
--------------------------------------------------------------------------------
    -- rule "‘home’ processes invalidate ack"
    if in3.valid
      and home_requests(in3.msg.addr).status = pending then

      assert in3.msg.op = invalidate_ack
        report "home: Expect only invalidate ack" severity failure;
--      report "home: Accept invalidate ack from input buffer 3";
      -- if invalidating exclusive cache, save data in request and memory
      if directory(in3.msg.addr)(in3.msg.source) = cache_exclusive then
        memory(in3.msg.addr) := in3.msg.data;
      end if;
      home_requests_next(in3.msg.addr).data <= in3.msg.data;
      directory(in3.msg.addr)(in3.msg.source) := cache_invalid;
      -- try to mark request as completed:
      case home_requests(in3.msg.addr).op is
        when read_shared =>
          -- just invalidated (the single) exclusive; done.
          home_requests_next(in3.msg.addr).status <= completed;
        when req_upgrade =>
          if all_but_1_invalid(directory(in3.msg.addr),
                               home_requests(in3.msg.addr).source) then
            home_requests_next(in3.msg.addr).status <= completed;
          else
--            report "home: upgrade not yet completed";
```

```
          end if;
        when read_exclusive =>
          if all_invalid(directory(in3.msg.addr)) then
            home_requests_next(in3.msg.addr).status <= completed;
          else
--              report "home: exclusive not yet completed";
          end if;
        when others =>
          report "home: Undefined case" severity failure;
      end case;
      --reset_in3 <= true;
    else
      --reset_in3 <= false;
    end if;
--------------------------------------------------------------------------------
  end process;
end behavior;
```

## B.5 Local.vhdl

```
-- Copyright (C) 2004-2005 Geert Janssen - IBM

library ieee;
use ieee.std_logic_1164.all;

use work.cachei_pkg.all;

-- local mode
-- reads from in channel 2
-- writes to out channel 1
entity local is
-- New cache values at cache_addr only valid when reset_in2 is true.
  generic (self: node_id := 0);
  port (
    default_clock: in std_ulogic;
    cache: in  cache_type;              -- observing the cache
    cache_next: out cache_record;       -- new cache line for
    cache_addr: out addr_type;          -- cache address
    cache_valid: out boolean;           -- is the new cache data valid
    cache_ack: in boolean;              -- cache indeed updated
    in2:   in  message_buf_type;        -- input request
    reset_in2: out boolean;             -- reset the input buffer
    out1:  out message_buf_type;        -- output request
    reset_out1: in boolean              -- reset the output buffer
    );
end local;

architecture behavior of local is
  signal buf1: message_buf_type := chan_0;
begin
  out1 <= buf1;
  reset_in2 <= cache_ack and in2.valid;

  -- rule "‘client’ receives reply from home"
  process (in2, cache)
  begin
    cache_addr  <= in2.msg.addr;
    cache_next  <= cache_0;
    cache_valid <= false;

    if in2.valid then
      case in2.msg.op is
        when grant_shared =>
          cache_next.data  <= in2.msg.data;
          cache_next.state <= cache_shared;
          cache_valid <= true;
--          report "local: Accept grant_shared";
        when grant_upgrade =>
          -- data in cache is still valid ???
          cache_next.data  <= cache(in2.msg.addr).data;
          cache_next.state <= cache_exclusive;
          cache_valid <= true;
--          report "local: Accept grant_upgrade";
        when grant_exclusive =>
          cache_next.data  <= in2.msg.data;
          cache_next.state <= cache_exclusive;
          cache_valid <= true;
--          report "local: Accept grant_exclusive";
        when others =>
          -- request handled by remote part.
          null;
      end case;
    end if;
  end process;

  process
    variable local_requests: local_requests_type := (others => false);
  begin
    wait until default_clock = '1';

    if reset_out1 then
      assert buf1.valid
        report "Spurious reset of output buffer 1" severity warning;
      buf1 <= chan_0;
    elsif not buf1.valid then
      outer: for req in request_opcode loop
        for addr in addr_type loop
          -- rule "‘client’ generates new ‘req’ for ‘addr’"
          if not local_requests(addr) and
            ((req=req_read_shared    and cache(addr).state = cache_invalid) or
             (req=req_read_exclusive and cache(addr).state = cache_invalid) or
             (req=req_req_upgrade    and cache(addr).state = cache_shared))
        then
            buf1.msg.source <= self;
            buf1.msg.dest   <= home_node(addr);
            buf1.msg.op     <= req_opcode(req);
            buf1.msg.addr   <= addr;
            buf1.valid      <= true;
            local_requests(addr) := true;
--            report "local: Prepared output buffer 1";
            exit outer;
          end if;
        end loop;
      end loop outer;
    end if;

    if cache_ack then
      local_requests(in2.msg.addr) := false;
--      report "local: Cache acks grant";
    end if;

  end process;
end behavior;
```

## B.6 Node.vhdl

```
-- Copyright (C) 2004-2005 Geert Janssen - IBM

library ieee;
use ieee.std_logic_1164.all;

use work.cachei_pkg.all;

entity node is
  generic (self: node_id);
  port (default_clock: in std_ulogic;
        chans_in:  in  channel_type;
        valid_in: out valid_channel_type;
        chans_out: out channel_type;
        reset_out: in reset_channel_type;
        cache_out: out cache_type
);
end node;

architecture behavior of node is
  -- output of cache
  signal cache_all: cache_type;
  signal cache_next_local: cache_record;
  signal cache_addr_local: addr_type;
  signal cache_next_remote: cache_record;
  signal cache_addr_remote: addr_type;
  signal reset_in: valid_channel_type;
  signal reset_in2_local: boolean;
  signal reset_in2_remote: boolean;
  signal chans_in_buf: channel_type;
  signal valid: boolean_array(0 to 1);
  signal ack: boolean_array(0 to 1);

  component buf
    port (default_clock: in std_ulogic;
          data_in: in message_type;
          data_out: out message_type;
          accept: in boolean;
          valid: out boolean;
          reset: in boolean
          );
  end component buf;

  component local
    generic (self: node_id);
    port (
    default_clock: in std_ulogic;
    cache: in  cache_type;
    cache_next: out cache_record;
    cache_addr: out addr_type;
    cache_valid: out boolean;
    cache_ack: in boolean;
    in2:   in  message_buf_type;
    reset_in2: out boolean;
    out1:  out message_buf_type;
    reset_out1: in boolean
    );
  end component local;

  component home
    generic (self: node_id);
    port (
    default_clock: in std_ulogic;
    cache: in  cache_type;
    in1:   in  message_buf_type;
    reset_in1: out boolean;
    in3:   in  message_buf_type;
    reset_in3: out boolean;
    out2:  out message_buf_type;
    reset_out2: in boolean
    );
  end component home;

  component remote
    generic (self: node_id);
    port (
      default_clock: in std_ulogic;
      cache: in  cache_type;
      cache_next: out cache_record;
      cache_addr: out addr_type;
      cache_valid: out boolean;
      cache_ack: in boolean;
      in2:   in  message_buf_type;
      reset_in2: out boolean;
      out3:  out message_buf_type;
      reset_out3: in boolean
    );
  end component remote;

  component cache
    generic (n: natural);
    port (
      clock: in std_ulogic;
```

```
      datain: in cache_record_array(0 to n-1);
      address: in addr_type_array(0 to n-1);
      valid: in boolean_array(0 to n-1);
      ack: out boolean_array(0 to n-1);
      cacheout: out cache_type
      );
  end component cache;

begin
  reset_in(2) <= reset_in2_local or reset_in2_remote;

  cache_out <= cache_all;

  buffers: for i in channel_id generate
    valid_in(i) <= chans_in_buf(i).valid;
    in_buf: buf
    port map(
      default_clock => default_clock,
      data_in => chans_in(i).msg,
      data_out => chans_in_buf(i).msg,
      accept => chans_in(i).valid,
      valid => chans_in_buf(i).valid,
      reset => reset_in(i)
      );
  end generate buffers;

  a_local: local
    generic map(self => self)
    port map (
      default_clock => default_clock,
      cache => cache_all,
      cache_next => cache_next_local,
      cache_addr => cache_addr_local,
      cache_valid => valid(0),
      cache_ack => ack(0),
      in2 => chans_in_buf(2),
      reset_in2 => reset_in2_local,
      out1 => chans_out(1),
      reset_out1 => reset_out(1)
      );

  a_home: home
    generic map(self => self)
    port map (
      default_clock => default_clock,
      cache => cache_all,
      in1 => chans_in_buf(1),
      reset_in1 => reset_in(1),
      in3 => chans_in_buf(3),
      reset_in3 => reset_in(3),
      out2 => chans_out(2),
      reset_out2 => reset_out(2)
      );

  a_remote: remote
    generic map(self => self)
    port map (
      default_clock => default_clock,
      cache => cache_all,
      cache_next => cache_next_remote,
      cache_addr => cache_addr_remote,
      cache_valid => valid(1),
      cache_ack => ack(1),
      in2 => chans_in_buf(2),
      reset_in2 => reset_in2_remote,
      out3 => chans_out(3),
      reset_out3 => reset_out(3)
      );

  the_cache: cache
    generic map(n => 2)
    port map (
      clock => default_clock,
      datain(0) => cache_next_local,
      datain(1) => cache_next_remote,
      address(0) => cache_addr_local,
      address(1) => cache_addr_remote,
      valid => valid,
      ack => ack,
      cacheout => cache_all
      );
end behavior;
```

## B.7 Router.vhdl

```
-- Copyright (C) 2004-2005 Geert Janssen - IBM

library ieee;
use ieee.std_logic_1164.all;

use work.cachei_pkg.all;

-- rule "transfer message"
-- A very simple router.
-- At any one time only a single path will be established between a
-- certain source input and a destination output.
-- The destination is taking from the source's message buffer if indeed valid.
-- The transfer is activated only when the destination buffer is available.
-- A transfer will imply resetting the input buffer.
entity router is
  generic (k: natural := 1);
  port (default_clock: in std_ulogic;
        chans_in:  in io_fabric;
        chans_out: out io_fabric;
        valid_out: in valid_node_type;
        reset_out: out reset_node_type;
        start: out boolean;
        active: out boolean;
        done: out boolean
);
end router;

architecture behavior of router is
--  alias outchan: message_buf_type is chans_in(source).outchan(ch);
--  alias dest: node_id is outchan.msg.dest;
--  alias inchan: message_buf_type is chans_out(dest).inchan(ch);
  signal source: node_id;
  signal dest: node_id;
  signal ch: channel_id;
  signal transfer: boolean;

  signal turn: node_id := 0;
  signal turn_next: node_id;
begin
  -- (Moore) outputs:
  start  <= false;
  active <= false;
  done   <= true;

  turn_next <= 0 when turn = num_nodes-1 else turn + 1;
  process
  begin
    wait until default_clock = '1';
    turn <= turn_next;
  end process;

  process (turn)
    variable s: node_id;
    variable d: node_id;
  begin
    transfer <= false;
    source <= 0;
    dest <= 0;
    ch <= 1;
    reset_out <= (others => (others => false));
    chans_out <= (others => (others => chan_0));
    s := turn;
    outer: for i in node_id loop
      for c in channel_id loop
        if chans_in(s)(c).valid then
          assert s = chans_in(s)(c).msg.source
            report "router: Mangled source of message" severity failure;
          d := chans_in(s)(c).msg.dest;
          if not valid_out(d)(c) then
            transfer <= true;
            source <= s;
            dest <= d;
            ch <= c;
            reset_out(s)(c) <= true;
            chans_out(d)(c) <= chans_in(s)(c);
            exit outer;
          end if;
        end if;
      end loop;
      if s = num_nodes-1 then
        s := 0;
      else
        s := s + 1;
      end if;
    end loop;
  end process;
end behavior;
```

## B.8 System.vhdl

```
-- Copyright (C) 2004-2005 Geert Janssen - IBM
-- Modified by Jason Baumgartner: new clock.

library ieee;
use ieee.std_logic_1164.all;

use work.cachei_pkg.all;

entity system is
end system;

architecture structure of system is
  signal default_clock: std_ulogic;
  signal chans_in: io_fabric;
  signal chans_out: io_fabric;
  signal resets: reset_node_type;
  signal start: boolean;
  signal active: boolean;
  signal done: boolean;
  signal valid_out: valid_node_type;
  signal caches: cache_node_type;

  component router
  generic (k : natural);
  port (default_clock: in std_ulogic;
        chans_in:  in io_fabric;
        chans_out: out io_fabric;
        valid_out: in valid_node_type;
        reset_out: out reset_node_type;
        start: out boolean;
        active: out boolean;
        done: out boolean
);
  end component router;

  component node
    generic (self: node_id);
    port (default_clock: in std_ulogic;
          chans_in:  in  channel_type;
          valid_in: out valid_channel_type;
          chans_out: out channel_type;
          reset_out: in reset_channel_type;
          cache_out: out cache_type
          );
  end component node;
  signal osc         : std_ulogic := '0';
  signal clockx      : std_ulogic := '0';
  signal beforestart : std_ulogic := '0';
begin

  nodes: for n in node_id generate
    a_node: node
      generic map(self => n)
      port map(default_clock => default_clock,
               chans_in => chans_in(n),
               valid_in => valid_out(n),
               chans_out => chans_out(n),
               reset_out => resets(n),
               cache_out => caches(n)
               );
  end generate nodes;

  the_router: router
    generic map(k => 4)
    port map(default_clock => default_clock,
             chans_in => chans_out,
             chans_out => chans_in,
             valid_out => valid_out,
             reset_out => resets,
             start => start,
             active => active,
             done => done
             );

  clkblock0: process(beforestart)
  begin
   if (beforestart = '0') then
     beforestart <= '1';
   end if;
  end process;

 clkblock1: process(beforestart,osc)
  begin
    if (beforestart = '1') then
     osc <= not osc;         -- pulsewidth
     clockx <= osc;
    end if;
  end process;

--  default_clock <= clockx;
  default_clock <= osc;
  assert check_invariants(caches)
    report "Cache coherence violated!" severity FAILURE;
end architecture structure;
```