Tutorial for SLICC v0.2

Milo Martin
8/25/1999

Overview

This document attempts to illustrate the syntax and expressiveness of the cache coherence protocol specification language through a small example. A "stupido" cache coherence protocol is described in prose and then expressed in the language.

The protocol used as the running example is described. Then each of the elements of the protocol is discussed and expressed in the language: states, events, transitions, and actions.

Protocol Description

In order to make this example a simple as possible, the protocol described is a simple as possible and makes many simplifying assumptions. These simplifications were made only to clarify the exposition and are not indications of limitations of the expressiveness of the description language. We have already specified a more complicated MSI broadcast snooping protocol, a multicast snooping protocol, and a directory protocol. The simplifying assumptions are listed below. The remaining details of the protocol are described in the sections where we give the syntax of the language.

- The protocol uses broadcast snooping that assumes that a broadcast can only occur if all processors have processed all of their incoming address transactions. In essence, when a processor issue an address request, that request will be next in the global order. This allows us to avoid needed to handle the cases before we have observed our request in the global order.
- The protocol has only Modified and Idle stable states. (Note: Even the Shared state is omitted.)
- To avoid describing replacement (PutX's) and writebacks, the caches are in treated as infinite in size.
- No forward progress bit is used, so the protocol as specified does not guarantee forward progress.
- Only the mandatory request queue is used. No optional or prefetch queue is described.
- The above simplifications reduce the need for TBEs (Translation Buffer Entries) and thus the idea of a TBE is not include.
- Each memory module is assumed to have some state associated with each cache block in the memory. This requires a simple directory/memory state machine to work as a compliment to the processor state machine. Traditional broadcast snooping protocols often have no "directory" state in the memory.

Protocol Messages

Cache coherence protocols communicate by sending well defined messages. To fully specify a cache coherence protocol we need to be able to specify the message types and fields. For this protocol we have address messages (`AddressMsg`) which are broadcast, and data messages (`DataMsg`) which are point-to-point. Address messages have an address field (`Address`), a request type (`Type`), and which processor made the request (`Requestor`). Data message have an address field (`Address`), a destination (`Destination`), and the actual cache block being transfered (`DataBlk`). The names in parenthesis are important because those are the names which code later in the specification will reference various message types and fields in the messages.

Messages are declared by creating types with `new_type()` and adding fields with `type_field()`. The exact syntax for declaring types and message is a bit ugly right now and is going to be changed in the near future. If you wish, please see the appendix for an example of the current syntax.

Cache States

Idle and Modified are the two stable states in our protocol. In addition the we have a single transient processor state. This state is used after a processor has issued a request and is waiting for the data response to arrive.

Declaring states in the language is the first of a number of declarations we will be using. All of these declarations have a similar format. Below is the format for a state declaration.

`state(`*identifier*`, `*shorthand*`, `*pair1*`, `*pair2*`, ...);`

identifier is a name that is used later to refer to this state later in the description. It must start with a letter and after than can have any combination of letters, numbers, and the underscore character. *shorthand* is a quoted string which contains the shorthand that should be used for the state when generating tables and such.

The *pair*'s are used to associate arbitrary information with each state. Zero or more pairs can be included in each declaration. For example we want to have a more verbose description of each state when we generate the table which contains the states and descriptions. This information is encoded in the language by adding a `desc` parameter to a declaration. The name of the parameter is followed by an equal sign and a string with the description. The `desc` pair technically optional, however the table generation tool will complain about a missing description if it is not present.

The three states for our protocol are expressed as follows:

```
state(I, "I", desc="Idle");
state(M, "M", desc="Modified");
state(IM, "IM", desc="Idle, issued request but have not seen data yet");
```

Cache Events

Events are external stimulus that cause the state machine to take action. This is most often a message in one of the queues from the network or processor. Events form the columns of the protocol table. Our simple protocol has one event per incoming queue. When a message is waiting in one of these queues and event can occur. We can see a request from the processor in the mandatory queue, another processor's request, or a data response.

Events are declared in the language similarly to states. The *identifier*, *shorthand*, and *pair*'s have the same purpose as in a state declaration.

```
event(identifier, shorthand, pair1, pair2, ...) {
  statement_list
}
```

Events are different in that they have a list of statements which allows exact specification of when the event should "trigger" a transition. These statements are mini-programming language with syntax similar to that of C. For example the `peek` construct in this context checks to see if there is a message at the head of the specified queue, and if so, conceptually copies the message to a temporary variable accessed as `in_msg`. The language also supports various procedure calls, functions, conditional statements, assignment, and queue operations such as peek, enqueue and dequeue. The `trigger()` construct takes an address as the only parameter. This is the address that should be triggered for the event. To give you a feel for what this code looks like, the three events for our simple protocol are below.

```
event(LoadStore, "LoadStore", desc="Load or Store request from local processor") {
  peek(mandatoryQueue_ptr, CacheMsg) {
    trigger(in_msg.Address);
  }
}

event(Other_GETX, "Other GETX", desc="Observed a GETX request from another processor") {
  peek(addressNetwork_ptr, AddressMsg) {
    if (in_msg.Requestor != id) {
      trigger(in_msg.Address);
    }
  }
}

event(Data, "Data", desc="Data for this block from the data network") {
  peek(dataNetwork_ptr, DataMsg) {
    trigger(in_msg.Address);
  }
}
```

Cache Actions

Actions are the privative operations that are performed by various state transitions. These correspond (by convention) to the lower case letters in the tables. We need several actions in our protocol including issuing a GetX request, servicing a cache hit, send data from the cache to the requestor, writing data into the cache, and popping the various queues.

The syntax of an action declaration is similar to an event declaration. The difference is that statements in the statement list are used to implement the desired action, and not triggering an event.

```
action(identifier, shorthand, pair1, pair2, ...) {
  statement_list
}
```

The actions for this protocol use more of the features of the language. Some of the interesting case are discussed below.

- To manipulate values we need assignment statements (notice the use of `:=` as the assignment operator). The action to write data into the cache looks at the incoming data message and puts the data in the cache. Notice the use of square brackets to lookup the block in the cache based on the address of the block.

```
action(w_writeDataToCache, "w", desc="Write data from data message into cache") {
  peek(dataNetwork_ptr, DataMsg) {
    cacheMemory_ptr[address].DataBlk := in_msg.DataBlk;
  }
}
```

- In addition to peeking at queues, we also enqueue messages. The `enqueue` construct works similarly to the `peek` construct. `enqueue` creates a temporary called `out_msg`. You can assign the fields of this message. At the end of the `enqueue` construct the message is implicitly inserted in the outgoing queue of the specified network. Notice also how the type of the message is specified and how the assignment statements use the names of the fields of the messages. `address` is the address for which the event was `triggered`.

```
action(g_issueGETX, "g", desc="Issue GETX.") {
  enqueue(addressNetwork_ptr, AddressMsg) {
    out_msg.Address := address;
    out_msg.Type := "GETX";
    out_msg.Requestor := id;
  }
}
```

- Some times we need to use both `peek` and `enqueue` together. In this example we look at an incoming address request to figure out who to whom to forward the data value.

```
action(r_cacheToRequestor, "r", desc="Send data from the cache to the requestor") {
  peek(addressNetwork_ptr, AddressMsg) {
    enqueue(dataNetwork_ptr, DataMsg) {
      out_msg.Address := address;
      out_msg.Destination := in_msg.Requestor;
      out_msg.DataBlk := cacheMemory_ptr[address].DataBlk;
    }
  }
}
```

- We also need to pop the various queues.

```
action(k_popMandatoryQueue, "k", desc="Pop mandatory queue.") {
  dequeue(mandatoryQueue_ptr);
}
```

- Finally we have the ability to call procedures and functions. The following is an example of a procedure call. Currently all of the procedures and functions are used to handle all of the more specific operations. These are currently hard coded into the generator.

```
action(h_hit, "h", desc="Service load/store from the cache.") {
  serviceLdSt(address, cacheMemory_ptr[address].DataBlk);
}
```

Cache Transitions

The cross product of states and events gives us the set of possible transitions. For example, for our example protocol the empty would be:

	LoadStore	Other GETX	Data
I			
M			
IM			

Transitions are atomic and are the heart of the protocol specification. The transition specifies both what the next state, and also what actions are performed for each unique state/event pair. The transition declaration looks different from the other declarations:

```
transition(state, event, new_state, pair1, pair2, ...)  {
   action_identifier_list
}
```

state and *event* are the pair which uniquely identifies the transition. *state* correspond to the row where *event* selects the column. *new_state* is an optional parameter. If *new_state* is specified, that is the state when the atomic transition is completed. If the parameter is omitted there is assumed to be no state change. An impossible transition is specified by simply not declaring an event for that state/event pair.

We also place list of actions in the curly braces. The *action_identifier*'s correspond to the identifier specified as the first parameter of an action declaration. The action list is a list of operations to be performed when this transition occurs. The actions also knows what preconditions are necessary are required for the action to be performed. For example a necessary precondition for an action which sends a message is that there is a space available in the outgoing queue. Each transition is considered atomic, and thus the generated code ensures that all of the actions can be completed before performing and of the actions.

In our running example protocol it is only possible to receive data in the *IM* state. The other seven cases can occur and are declared as follows. Below are a couple of examples. See the appendix for a complete list.

```
transition(I, LoadStore, IM) {
  g_issueGETX;
}

transition(M, LoadStore) {
  h_hit;
  k_popMandatoryQueue;
}

transition(M, Other_GETX, I) {
  r_cacheToRequestor;
  i_popAddressQueue;
}
```

From the above declarations we can generate a table. Each box can have lower case letters which corresponds to the list of actions possibly followed by a slash and a state (in uppercase letters). If there is no slash and state, the transition does not change the state.

	LoadStore	Other GETX	Data
I	g/IM	i	(impossible)
M	hk	ri/I	(impossible)
IM	z	z	wj/M

There is a useful shorthand for specifying many transitions with the same action. One or both of *event* and *state* can be a list in curly braces. This defines the cross product of the sets in one declaration. If no *new_state* is specified none of the transitions cause a state change. If *new_state* is specified, all of the transitions in the cross product of the sets has the same next state. For example, in the below transitions both IM/LoadStore and IM/Other_GETX have the action `z_delayTrans`.

```
transition(IM, LoadStore) {
  z_delayTrans;
}

transition(IM, Other_GETX) {
  z_delayTrans;
}
```

These can be specified in a single declaration:

```
transition(IM, {LoadStore, Other_GETX}) {
  z_delayTrans;
}
```

Appendix - Sample Cache Controller Specification

```
machine(processor, "Simple MI Processor") {

  // AddressMsg
  new_type(AddressMsg, "AddressMsg", message="yes", desc="");
  type_field(AddressMsg, Address, "address",
     desc="Physical address for this request",
     c_type=PhysAddress, c_include="Address.hh", murphi_type="");
  type_field(AddressMsg, Type, "type",
     desc="Type of request (GetS, GetX, PutX, etc)",
     c_type=CoherenceRequestType, c_include="CoherenceRequestType.hh", murphi_type="");
  type_field(AddressMsg, Requestor, "requestor",
     desc="Node who initiated the request",
     c_type=ComponentID, c_include="ComponentID.hh", murphi_type="");

  // DataMsg
  new_type(DataMsg, "DataMsg", message="yes", desc="");
  type_field(DataMsg, Address, "address",
     desc="Physical address for this request",
     c_type=PhysAddress, c_include="Address.hh", murphi_type="");
  type_field(DataMsg, Destination, "destination",
     desc="Node to whom the data is sent",
     c_type=Set, c_include="Set.hh", murphi_type="");
  type_field(DataMsg, DataBlk, "data",
     desc="Node to whom the data is sent",
     c_type=DataBlock, c_include="DataBlock.hh", murphi_type="");

  // CacheEntry
  new_type(CacheEntry, "CacheEntry");
  type_field(CacheEntry, CacheState, "Cache state", desc="cache state",
    c_type=CacheState, c_include="CacheState.hh", murphi_type="");
  type_field(CacheEntry, DataBlk, "data", desc="data for the block",
    c_type=DataBlock, c_include="DataBlock.hh", murphi_type="");

  // DirectoryEntry
  new_type(DirectoryEntry, "DirectoryEntry");
  type_field(DirectoryEntry, DirectoryState, "Directory state", desc="Directory state",
     c_type=DirectoryState, c_include="DirectoryState.hh", murphi_type="");
  type_field(DirectoryEntry, DataBlk, "data", desc="data for the block",
     c_type=DataBlock, c_include="DataBlock.hh", murphi_type="");
```

```
// STATES
state(I, "I", desc="Idle");
state(M, "M", desc="Modified");
state(IM, "IM", desc="Idle, issued request but have not seen data yet");

// EVENTS

// From processor
event(LoadStore, "LoadStore", desc="Load or Store request from local processor") {
  peek(mandatoryQueue_ptr, CacheMsg) {
    trigger(in_msg.Address);
  }
}

// From Address network
event(Other_GETX, "Other GETX", desc="Observed a GETX request from another processor") {
  peek(addressNetwork_ptr, AddressMsg) {
    if (in_msg.Requestor != id) {
      trigger(in_msg.Address);
    }
  }
}

// From Data network
event(Data, "Data", desc="Data for this block from the data network") {
  peek(dataNetwork_ptr, DataMsg) {
    trigger(in_msg.Address);
  }
}

// ACTIONS
action(g_issueGETX, "g", desc="Issue GETX.") {
  enqueue(addressNetwork_ptr, AddressMsg) {
    out_msg.Address := address;
    out_msg.Type := "GETX";
    out_msg.Requestor := id;
  }
}

action(h_hit, "h", desc="Service load/store from the cache.") {
  serviceLdSt(address, cacheMemory_ptr[address].DataBlk);
}

action(i_popAddressQueue, "i", desc="Pop incoming address queue.") {
  dequeue(addressNetwork_ptr);
}

action(j_popDataQueue, "j", desc="Pop incoming data queue.") {
  dequeue(dataNetwork_ptr);
}

action(k_popMandatoryQueue, "k", desc="Pop mandatory queue.") {
  dequeue(mandatoryQueue_ptr);
}
```

```
  action(r_cacheToRequestor, "r", desc="Send data from the cache to the requestor") {
    peek(addressNetwork_ptr, AddressMsg) {
      enqueue(dataNetwork_ptr, DataMsg) {
        out_msg.Address := address;
        out_msg.Destination := in_msg.Requestor;
        out_msg.DataBlk := cacheMemory_ptr[address].DataBlk;
      }
    }
  }

  action(w_writeDataToCache, "w", desc="Write data from data message into cache") {
    peek(dataNetwork_ptr, DataMsg) {
      cacheMemory_ptr[address].DataBlk := in_msg.DataBlk;
    }
  }

  action(z_delayTrans, "z", desc="Cannot be handled right now.") {
    stall();
  }

  // TRANSITIONS

  // Transitions from Idle
  transition(I, LoadStore, IM) {
    g_issueGETX;
  }

  transition(I, Other_GETX) {
    i_popAddressQueue;
  }

  // Transitions from Modified
  transition(M, LoadStore) {
    h_hit;
    k_popMandatoryQueue;
  }

  transition(M, Other_GETX, I) {
    r_cacheToRequestor;
    i_popAddressQueue;
  }

  // Transitions from IM
  transition(IM, {LoadStore, Other_GETX}) {
    z_delayTrans;
  }

  transition(IM, Data, M) {
    w_writeDataToCache;
    j_popDataQueue;
  }
}
```