Mutual Exclusion Property Verification of FLASH Cache Coherence Protocol by Inductive Invariant Checking

Sudhindra Pandav, Konrad Slind, and Ganesh Gopalakrishnan

UUCS-04-010

School of Computing
University of Utah
Salt Lake City, UT 84112 USA

July 29, 2004

Abstract

We discuss a verification of the cache coherence protocol of the Stanford FLASH multiprocessor for arbitrary number of processors using the UCLID system to specify and verify the protocol. UCLID has a SAT-based decision procedure for a logic of counter arithmetic with lambda expressions and uninterpreted functions. We verified the mutual exclusion property for the protocol using just 8 inductive invariants. We discovered the invariants frugally, and solely from the analysis of counterexamples that UCLID returns, on deciding a property. The proof process is described step by step.

Mutual Exclusion Property Verification of FLASH Cache Coherence Protocol by Inductive Invariant Checking

Sudhindra Pandav, *Konrad Slind*, and *Ganesh Gopalakrishnan*
School of Computing, University of Utah
{sudhindr, slind, ganesh}@cs.utah.edu

July 29, 2004

Abstract

We discuss a verification of the cache coherence protocol of the Stanford FLASH multiprocessor for arbitrary number of processors using the UCLID system to specify and verify the protocol. UCLID has a SAT-based decision procedure for a logic of counter arithmetic with lambda expressions and uninterpreted functions. We verified the mutual exclusion property for the protocol using just 8 inductive invariants. We discovered the invariants frugally, and solely from the analysis of counterexamples that UCLID returns, on deciding a property. The proof process is described step by step.

1 Introduction

In shared memory multi-processors, cache-coherence protocols maintain consistency of multiple copies of cached data. The protocols control a number of readable and writable copies of each memory line for multiprocessors. Modification of one copy of a datum line may require updating of other copies to maintain consistency among them. One of the key property in such protocols is the property of mutual exclusion i.e., no two processors can possess writable copies of a memory line.

Formal Verification is desirable because there could be subtle bugs as the complexity of protocols increases, which cannot be detected in simulation. The FLASH cache coherence protocol has already been verified by Seungjoon Park [3], McMillan [2] and several others. Park used the PVS theorem prover to verify by his method of *aggregation* of distributed transactions. But this is a laborious process, because it requires generating a lot of inductive invariants (around 72), the theorem prover has to be manually guided and its time consuming. McMillan used his SMV model checker for the verification employing the method of *compositional* model checking. Though his proof is elegant, it puts a huge burden on the model checker. The proof was completed in two days.

We verify the mutual exclusion property for the FLASH protocol in UCLID system. The UCLID [4] system is used to specify and verify parameterized sys-

tems of unbounded state. It has a decision procedure based on a logic of counter arithmetic with lambda expressions and uninterpreted functions [1]. The advantage of using UCLID is that its decision procedure is completely automated and very well suited for parameterized systems. Its language is sufficiently rich to model commonly used data structures by lambda expressions and uninterpreted functions. The verification is done by the method of *inductive invariant checking*. Using this method, one verifies whether the next state transition function preserves the safety property at the most general state of the protocol. One must verify that the transition relation preserves the safety property at the initial state of the protocol. We have to design auxiliary invariants to prune the unreachable state space. In addition to the challenge of finding these invariants, there is the challenge of avoiding irrelevant invariants since the cost of checking the system increases exponentially with the number of invariants. We have devised a systematic way of discovering invariants as and when needed. It took just 8 invariants to prove the safety property. We do a thorough analysis of the counterexample trace that UCLID outputs (when a property fails) to discover or strengthen the invariants. We describe in detail this analysis step by step.

Clearly, we can't yet compare our method with Park's method, since we have verified only the safety property. As we have obtained promising results in terms of the number of invariants and proof time, we hope that we can prove other properties too more efficiently and elegantly.

We briefly explain the FLASH Cache Coherence Protocol in Section 2. In Section 3, we give a short background of UCLID theorem prover. Then, in Section 4, the protocol model is described. The next section gives the overview of inductive invariant checking method for verification of mutual exclusion property of FLASH. We give a step-by-step detail of generating the invariants in section 6. Finally, we conclude in section 7 with future work to be done.

2 FLASH Cache Coherence Protocol Description

The cache-coherence protocol is a directory based protocol. Each cache-line sized block in memory is associated with directory header which keeps information about the line. For a memory line, the node on which that piece of memory is physically located is called home and the other nodes are called remote. The home maintains all the information about memory lines in its main memory in the corresponding directory headers.

The system consists of a set of nodes, each of which contains a processor, caches and a portion of global memory of the system. The nodes communicate using asynchronous messages. The state of a cached copy is either *invalid*, *shared* (readable) or *exclusive* (readable and writable).

If a read miss occurs in a processor, the corresponding node sends out a **Get** request to the home. Receiving the **Get** request, the home consults the directory corresponding to the memory line to decide what action the home should take. If the line is pending, meaning another request is already being processed, the home sends a **NAK** to the requesting node. If the directory indicates there is a dirty copy in a remote node, then the home forwards the **Get** request to that node. Otherwise the home grants the request by sending

a **Put** to the requesting node and updates the directory properly. When the requesting processor receives the **Put** reply, the processor sets its cache to *shared* and proceeds to read.

If a write miss occurs in a processor, the corresponding node sends out a **GetX** request to the home. Receiving the **GetX** request, the home consults the directory. If the line is pending, the home sends a **NAK** to the requesting node. If the directory indicates there is a dirty copy in a remote node, then the home forwards the **GetX** to that node. If the directory indicates there are shared copies of the memory line in other nodes, the home sends invalidations (**INV**) to those nodes. The FLASH multiprocessor has two modes, in which it can be run. In Eager mode, the home grants the request by sending the **PutX** message to the requesting node, without waiting for invalidation acknowledgments to be received. In Delayed mode, this grant is deferred till all the invalidation acknowledgments are received. If there are no shared copies, the home sends a **PutX** to the requesting node and updates the directory. On receiving **PutX** message, the requesting node changes its cache state to *exclusive*.

During the read miss transaction, an operation called sharing write-back is necessary in the following "three hop" case. This occurs when a remote processor in node R_1 needs a shared copy of a memory line of which an exclusive copy is another node R_2. When the **Get** request from R_1 arrives at home H, the home consults the directory to find that the line is dirty at R_2. Then H forwards the **Get** request to R2 with the source faked as R_1. When R_2 receives the forwarded **Get**, the processor sets its cache state to *shared* and issues a **Put** to R_1. Unfortunately the directory in H does not have R_1 on its sharer list yet and the main memory does not have an updated copy when the cache line is in *shared state*. The solution is for R_2 to issue a **ShWB** (sharing writeback) conveying the dirty data to H with the source faked as R_1. When H receives this message, it writes the data back into memory and puts R_1 on sharer list.

When a remote node receives an **INV**, it invalidates its copy and sends **InvAck** to the home. A processor which is waiting for a **Put** reply may get an **INV** before it gets the shared copy. In such a case, the requested line is marked as invalidated, and the **Put** reply is ignored when it arrives.

A shared copy can be replaced by issuing a replacement message to the home. An exclusive copy is written back to the memory by a **WB** (write-back) request to the home. Receiving the **WB**, the home updates the line in main memory and the directory properly.

3 Background : UCLID

The UCLID system [4] is used to specify and verify systems that have infinite or unbounded state. The language include uninterpreted function and predicate symbols, counter arithmetic and non-nested lambda expressions. It can be used to specify a state machine, where each state variable can be boolean, enumerated or an uninterpreted symbol. The lambda expressions can be used to define data structures like arrays, which are common in the FLASH protocol. Also, using lambda expressions multiple positions of an array can be updated in a single step.

The UCLID verification engine comprises of a symbolic simulator that can be "configured" for different kinds of verification tasks, and a SAT based deci-

sion procedure for a logic of counter arithmetic with lambda Expressions and uninterpreted functions [1]. We describe below the verification tasks we used in proving the cache coherence property.

- **Bounded Model Checking** The symbolic simulation of the model can be done using the *simulate* command, specifying the number of steps as an argument. The *decide* command can then be used to check the safety property of interest in the state the system attained after the simulation. This is usually a very useful method to find simple bugs such as typographical mistakes in the specification.

- **Inductive Invariant Checking** In this technique, the starting state is initialized to the most general state. The system is symbolically simulated for one step. Then, a property of the form $\mathcal{P} \Rightarrow \mathit{Next}(\mathcal{P})$ is checked, where $\mathcal{P}$ denotes a formula for the safety property we wish to verify and $\mathit{Next}(\mathcal{P})$ is its next-state version.

  In general, the user will have to supply the verification engine with several other auxiliary invariants along with the safety property $\mathcal{P}$. The process of deciding these auxiliary invariants is automatic. If the invariant is found to be false, UCLID returns a counterexample. This counterexample can be used to further strengthen the invariant or discover new ones.

  Most of the time, its very difficult to understand these counterexamples in a complex system like FLASH involving numerous state variables and rules. The idea is to focus on the *state variables* in (i) the *transition rule* in the counterexample, (ii) the *safety property* of interest (iii) the false *invariant* and (iv) *source rules* that can possibly fire the *transition rule*. Once we determine these "state variables of interest", we try to figure out relationship(s) between them. There seems to be no systematic way of discovering the relationship(s), given any counterexample. However, there are some "intelligent" tricks that can be used to find them. We can use these relationships to further strengthen the invariant or discover new ones. We explain in details in section 6 how to interpret the counterexample and the tricks to determine the relationships in the case of FLASH.

4 Modeling of FLASH Cache Coherence Protocol in UCLID

Our model closely follows McMillan's model in SMV [2]. We model processors, data and memory as uninterpreted TERMs. We model a cache as an array where the index represents the processor. Directory fields are declared as TRUTH or TERM as the case may be. The field *dir_real* which gives the number of shared copies of a cache-line is modeled as a boolean array where the index is the processor. *dir_real* is set for the processor with a shared copy. The message passing takes on 4 different kinds of network. They are **unet**, **wbnet**, **shwbnet** and **invnet**. As the names suggest, **unet** deals with messages of type {**Get**, **GetX**, **Put**, **PutX**}. **wbnet** deals with messages of type **WB**. **shwbnet** deals with messages of type {**ShWB**,**FAck**} . **invnet** carries messages of type {**Inv**,**InvAck**} . The entire four networks are modeled as an array.

Rules in FLASH are of form : **{A_B_C_N}** where
A = {PI, NI}
B = {Local, Remote}
C = {Get, GetX, Put, PutX, NAK, Inv, WB, ShWB, FAck, InvAck, Replace}
N = {1,2,3}

The PI rules are initiated by a requesting processor, while NI rules are initiated by a message from the network. The notation 'Local' or 'Remote' imply whether the processing node is home node or not.

The rules in FLASH are nested if-then-else statements, with each condition being a boolean formula involving at least 4 state variables. But, unlike SMV, UCLID doesn't support nested ITE's. To deal with this, we divide the rule R into many simpler non-nested ITE's. For example, consider a rule R of the form below (for simplicity, we assume the conditions to be made of single clause, unlike in real model).

R: if (c1) then b2 else if (c2) then b3 ;

So we divide the rule R into following 2 rules :

R_1 : if (c1) then b2 ;
R_2 : if (~c1 & c2) then {b3} ;

There are about 30 such rules in the system.

The other notable difference is that the SMV model uses existential quantification to define auxiliary state variables *some_others_left* and *not_need_invs*. The variable *some_others_left* is set when the home node has yet to receive all *InvAcks* from the nodes who had the copy of the cache line. *not_need_invs* is set when no node has a copy of the cache line. But UCLID doesn't support the existential quantifier.

These state variables were of the form, **s′** := $\otimes$(∃ **i : (i != src) & s(i)**), where **s′** is the auxiliary state variable, $\otimes$ is an unary boolean operator, **src** is the source node and **s** is a state variable already defined. For example, **not_need_invs** := ~(∃ **i : (i != src) & dir_shlist(i)**), where *dir_shlist(i)* is the list of sharer nodes. This means that, *not_need_invs* is set, if there doesn't exist any non-source node in the list of sharers.

To deal with this, we consider the set of rules $\mathcal{R}$, which assign the next state of the state variable **s** to be 1. For each such rule in $\mathcal{R}$, we set or reset the value of **s′**, depending on $\otimes$ and taking care that **(i != src)** .
Formally in UCLID,

$next(s') := lambda(i)$. ITE $(((i != src)\ \&\ (rule \notin \mathcal{R}))\ ,\ (\otimes(1))\ ,\ s'(i))$;

In the case of *not_need_iinvariants*, we reset its value for all those rules that set the state variable *dir_shlist* for non source nodes.

5 Verification

We verify whether the system satisfies the property of Mutual Exclusion (*ME*), represented in UCLID as

$$ME := Lambda\ (i,j)\ .\ ((i\ !=\ j)\ \&\ (cache_state\ (i)\ =\ exclusive)) \Rightarrow (cache_state\ (j)\ !=\ exclusive)\ ;$$

We use the method of Inductive Invariant Checking to prove the above property. There are two parts involved :

Part A : Prove that the next state transition relation preserves the safety property at the initial state. Initial state for the FLASH system is the state where all the networks are empty and the caches are in *invalid* state.

Part B : Prove that the next state transition relation preserves the safety property at the most general state. We model the most general state as the state where the caches and networks can non-deterministically have any values.

Part A is easy to prove. So we focus on Part B.

It is possible that the most general state, as determined above, can appear in unreachable state space. We have to care only about the reachable state space satisfying the safety property. To do this, we construct invariants such that they restrict the state space to reachable state space relevant to the safety property. There is no systematic way to construct invariants. We construct them based on counterexample obtained on running UCLID. Once we construct the invariants, we decide the following formula :

$$FORALL\ (i,\ j).\ (init_invs\ (i,j) \Rightarrow next_ME\ (i,j))\ ;$$

where *init_invs* represents the state of invariants and safety property *ME* before simulating for one step, and *next_ME* represents the state of safety property *ME* after simulation.

We decide the formula that, given that all invariants and *ME* are satisfied at the most general state before simulation, the safety property is valid at the next state. We describe in detail how the invariants are discovered in the next section.

6 Invariant Generation in FLASH

The invariants were generated as needed from the structure of the counterexample. We explain below how the invariants were discovered and strengthened.

First we try to check the safety property *MESym* (mutual exclusion is symmetric) without providing any invariant to the system.

$$ME := Lambda(i,j).\ ((i\ !=\ j)\ \&\ (cache_state(i)\ =\ exclusive)\ ) \Rightarrow (cache_state(j)\ !=\ exclusive)\ ;$$

$MESym := Lambda(i,j).\ (ME(i,j)\ \&\ ME(j,i))$;

This gave a counterexample on running UCLID. Counterexample in UCLID is presented in two parts. Part 1 assigns interpretations to function and predicate symbols. Part 2 is the Counter Example Trace, showing two states. State 0 is the *initial* state, and state 1 is the *next* state which violates the invariant. The *next* state is obtained by applying some transition *rule*, chosen nondeterministically, on the *initial* state.

For each counterexample trace, we will do an analysis of the trace to help discover or strengthen invariants. We will look at the state variable assignments, relevant to the safety property, in both the *initial* and *next* states. We will also look at the $rule(s)$, which are responsible for taking the system to the *next* state, where it violates the property. Since we will be analyzing the counterexamples in great detail, it would be helpful for the reader to have a copy of the appendix A of Park's [3] paper. The appendix A gives a detailed description of FLASH protocol(EAGER mode).

Following is the counterexample obtained on running UCLID to decide *MESym*.

Counterexample 1
Initial State

$i := home$, $j := remote$, $dir_dirty := false$
$cache_state(i) := shared$, $cache_state(j) := exclusive$
$rule := PI_Local_GetX$ (rule to be simulated)

Clearly the initial state satisfies the invariant, as it doesn't satisfy the antecedent part of the invariant.

Next State

The rule *PI_Local_GetX* grants exclusive access to the *home* node and sets *dir_dirty*. So we have,

$dir_dirty := true$, $cache_state(i) := exclusive$

thus contradicting with *MESym*.

Analysis

Consider the initial state. *dir_dirty* wasn't set even though there was an exclusive copy in the system. This forced the rule *PI_Local_GetX* to grant exclusive access to the *home* node.

So, we can get rid of this incorrect initial state by generating an auxiliary invariant, which says that, if there exists a node in an *exclusive* state then *dir_dirty* is true. Let's call this auxiliary invariant A.

$A := Lambda\ (i)\ .\ (cache_state(i) = exclusive) \Rightarrow dir_dirty$;

Now, using invariant A, we try to decide the safety property *MESym*. We run UCLID and obtain the following counterexample.

Counterexample 2
Initial State

$i := r_1$ (remote 1), $j := r_2$ (remote 2)
cache_state(i) := *shared*, *cache_state*(j) := *exclusive*
dir_dirty := *true* (as expected because of invariant A)
unet_mtype(i) := *PutX* (means i has received a *PutX* message)
rule := *NI_Remote_PutX*

Next State

The rule *NI_Remote_PutX* simply puts the exclusive copy into cache. So we have *cache_state*(i) := *PutX* thus contradicting *MESym*.

Analysis

Again there is a problem in the initial state. If there is *PutX* in the network, then there can't be a node in *exclusive* state, because if there is a node in *exclusive* state then *dir_dirty* is true and if *dir_dirty* is true then no rule allows *PutX* message to be sent.

So we get another invariant B.

$B :=$ *Lambda* (i,j) . (*unet_mtype*(i) = *PutX*) $\Rightarrow$ (*cache_state*(j) != *exclusive*) ;

And we run UCLID again to decide *MESym* using both A and B invariants. This time, UCLID doesn't find any counter-examples and the formula is found to be valid. This proves the safety property using invariants A and B.

The job, now in hand, is to prove the invariants A and B. So we run UCLID now to decide invariant A. And we get counter-example in proving A.

Counterexample 3
Initial State

i := *remote*, *cache_state*(i) := *exclusive*, *dir_dirty* := *true*
rule := *NI_Writeback*, *wbnet_mtype* := *WB*

So the initial state satisfies A.

Next state

The rule *NI_Writeback* on receiving *WB* message resets *dir_dirty*.
So, *dir_dirty* := false, thus contradicting A.

Analysis

If we look at the source of *WB*, from rule *PI_Remote_PutX*, *WB* can only be initiated by the *remote* node in an *exclusive* state. Since the initial state had a *remote* node in an *exclusive* state, unlike previous counterexamples the initial state assignment is correct.

We consider the source rule of *WB* message, *PI_Remote_PutX*. This rule states that the node with exclusive access changes its state to *invalid* when *WB* message is sent. And since there can be only one node in *exclusive* state as per *MESym*, we can't have a *WB* message if there is a node in *exclusive* state. So

we strengthen A further to A_1.

$$A_1 := Lambda\ (i)\ .\ (cache_state(i) = exclusive) \Rightarrow ((wbnet_mtype\ != WB)\ \&\ dir_dirty)\ ;$$

We run UCLID to decide the strengthened invariant A_1. UCLID outputs a counterexample of the following form.

Counterexample 4
Initial State

$i := remote$
$cache_state(i) := exclusive$, $dir_dirty := true$
$shwbnet_mtype := ShWB$
$rule := NI_SharingWriteback$

Next State
Like *NI_Writeback*, *NI_SharingWriteback* resets *dir_dirty*. so $dir_dirty := false$, contradicting A_1.

Analysis
Once again we have to further strengthen A_1 to A_2, based on the same argument as in the previous case. So,

$$A_2 := Lambda\ (i)\ .\ (cache_state(i) = exclusive) \Rightarrow ((wbnet_mtype\ != WB)\ \&\ (shwbnet_mtype\ != ShWB)\ \&\ dir_dirty)\ ;$$

We run UCLID to decide A_2 and get the following counterexample.

Counterexample 5
Initial State

$i := remote$, $cache_state(i) := shared$
$unet_mtype(i) := PutX$ (*remote* node in *shared* state receives a *PutX* message)
$dir_dirty := false$
$wbnet_mtype := WB$
$rule := NI_Remote_PutX$

Next state
The rule *NI_Remote_PutX* puts the *remote* node in *exclusive* state. So, $cache_state(i)$:= *exclusive*, which contradicts invariant A_1 since *dir_dirty* is *false* and $wbnet_mtype = WB$.

Analysis
We look at the initial state assignments, especially the source rule of *PutX* (as *PutX* fired the rule *NI_Remote_PutX*). From the rules, *NI_Local_GetX* and *NI_Remote_GetX*, we can see that whenever there is *PutX* message in the network *dir_dirty* is set. Also, if there is *PutX* message in the network, then there can't be a node in *exclusive* state, by invariant B. Since there can't be a node in *exclusive* state, there can't be a *WB* message in the network, as only the

exclusive node has the privilege to write back. A similar argument shows that a *ShWB* cannot be in the system.

The idea in strengthening of an invariant is to understand the source rules of the message that fired the transition rule in the counterexample trace, and determine how the source rules affect the state variables in the invariant.

So we further strengthen A_2 to A_3.

$$A_3 := Lambda\ (i)\ .\ ((cache_state(i) = exclusive)\ |\ (unet_mtype(i) = PutX)) \Rightarrow ((wbnet_mtype\ != WB)\ \&\ (shwbnet_mtype\ != ShWB)\ \&\ dir_dirty)\ ;$$

Now we run UCLID which gives the following counterexample.

Counterexample 6

Initial State

$i := home$, $cache_state(i) := shared$,
$dir_dirty := false$,
$wbnet_mtype := WB$,
$rule := PI_Local_GetX$

Next State

The *home* node requests *exclusive* access and since *dir_dirty* is not set, it gets *exclusive* status by rule *PI_Local_GetX*. So,

$$cache_status(i) := exclusive$$

which contradicts A_3 since *dir_dirty* is *false* and $wbnet_mtype := WB$.

Analysis

Since the initiator of the rule *PI_Local_GetX* is not a message, unlike previous case we don't venture into looking at other rules. Instead we focus on the initial state and the rule *PI_Local_GetX*.

We could have avoided i being granted *exclusive* access, had *dir_dirty* been set. Since a node being in *shared* state doesn't imply directory being dirty, we look for relationship between *dir_dirty* and *WB* message.

Since *WB* message can only be issued by a node in *exclusive* state, and a node in *exclusive* state implies *dir_dirty* being set, we get a new invariant here. If there is a *WB* message in the network, then *dir_dirty* must be set. So, we have a new entrant.

$$C := (wbnet_mtype = WB) \Rightarrow dir_dirty\ ;$$

With invariant C in the system too, we go for deciding A_3. Once again, we get a counter-example, which is exactly similar to *counterexample* 6 but in place of *WB* we have *ShWB*.

Counterexample 7

This counterexample is similar to *counterexample 6*, except we have *ShWB* message instead of *WB*, i.e.,

shwbnet_mtype := *ShWB*.

As in the case of *counterexample 4*, we strengthen invariant C to include *ShWB*. So, we have

C_1 := ((*wbnet_mtype* = *WB*) | (*shwbnet_mtype* = *ShWB*)) ⇒ *dir_dirty* ;

We get a counterexample.

Counterexample 8
Initial State

i := *remote*, *cache_state*(*i*) := *exclusive*,
unet_mtype(*home*) := *Put* (*home* receives a *Put* message)
dir_dirty := *true*.
rule := *NI_Local_Put*

Next State
This rule simply resets *dir_dirty*, giving contradiction.

Analysis
Once again we rely on our technique of attacking the source rule of *Put* message to *home* (that initiated *NI_Local_Put*).

The source rule of *Put* message to *home* is *NI_Remote_Get*. Before issuing *Put* message, the remote node changes its state from *exclusive* to *shared*. But we have a node in initial state with *exclusive* state, which shouldn't happen as there can be exactly one node in *exclusive* state. So, we coin a new invariant D.

D := *Lambda* (*i*,*j*) . (*unet_mtype*(*i*) = *Put*) ⇒ (*cache_state*(*j*) != *exclusive*);

This again gives a counterexample almost similar to *counterexample 8*, except that we have node i receiving *PutX* instead of being in *exclusive* state. So, we strengthen D further to

D$_1$:= *Lambda* (*i*,*j*) . (*unet_mtype*(*i*) = *Put*) ⇒ ((*cache_state*(*j*) != *exclusive*)
& (*unet_mtype*(*j*) != *PutX*));

This time UCLID outputs the formula to be valid.

So, with invariant B and new invariants C_1 and D_1 we proved the invariant A_3. Now we go on to prove invariant B.

B := *Lambda* (*i*,*j*) . (*unet_mtype*(*i*) = *PutX*) ⇒ (*cache_state*(*j*) != *exclusive*);

We obtain a counterexample on running UCLID to decide B.

Counterexample 9
Initial State

$i := r_1$, $j := r_2$

unet_mtype(i) := *PutX*, *unet_mtype*(j) := *PutX* (Both i & j receive *PutX*)
cache_state(i) := *cache_state*(j) := *shared*.
dir_dirty := *true*
rule := *NI_Remote_PutX*

Next State

The rule *NI_Remote_PutX* applied on node j, puts j in *exclusive* mode.
So, we have *cache_state*(j) := *exclusive*, which gives a contradiction with B.

Analysis

We use our technique of considering the initial state interpretation, especially we focus on that state variable which fires the rule *NI_Remote_PutX*, and try to figure out whether it can be a possible reachable state. The initial state says that there are two *PutX* messages in the network. Now, if we look at the possible sources of the *PutX* message to the *remote* node, the message can be send either by *home* (by rule *NI_Local_GetX*) or some other *remote* node (by rule *NI_Remote_GetX*). If there is already a *PutX* message in the network (at node i), then *dir_dirty* should be set (by invariant A_3) and the *home* (or *remote*) node shouldn't have sent *PutX* message. Since *dir_dirty* was set (as per invariant A) in the initial state, still we got an invalid trace with two *PutX* messages, suggesting that the initial state should be an unreachable state.

So we key in a new invariant E, that prunes such states, viz., there can be exactly one *PutX* message in the network.

$$E := \mathit{Lambda}\ (i,j)\ .\ ((i\ !=\ j)\ \&\ (\mathit{unet_mtype}(i) = \mathit{PutX})) \Rightarrow (\mathit{unet_mtype}(j)\ !=\ \mathit{PutX})\ ;$$

With invariant E in the set of invariants, we are able to prove the invariant B, as UCLID outputs the formula to be valid.

The next invariant to be checked is C_1.

$$C_1 := ((\mathit{wbnet_mtype} = \mathit{WB})\ |\ (\mathit{shwbnet_mtype} = \mathit{ShWB})) \Rightarrow \mathit{dir_dirty}\ ;$$

We run UCLID to decide the above property, which returns the following counterexample. As before, we will consider only those state variable assignments, that are either present in the property or fire a transition rule that leads to invalid state.

Counterexample 10

Initial State

wbnet_mtype := *WB*, *shwbnet_mtype* := *ShWB*
dir_dirty := *true*
rule := *NI_Writeback*

Next State

The rule *NI_Writeback* resets *dir_dirty* flag. So, we have *ShWB* message with *dir_dirty* not being set, thus giving contradiction.

Analysis

The initial state shows both *WB* and *ShWB* messages in the system. But this is not possible, because if a node sends a *WB* message, then that node should be the one which has the *exclusive* copy (by rule *PI_Remote_PutX*). Also, the node which sends *ShWB* message also should be in *exclusive* state (by rule *NI_Remote_Get*). Since there can be only one node in *exclusive* state, we get a new invariant.

$$F := (wbnet_mtype = WB) \Rightarrow (shwbnet_mtype \mathrel{!}= ShWB) \ ;$$

We run UCLID with this invariant. We get a different counterexample this time.

Counterexample 11

Initial State

$wbnet_mtype := WB,$
$unet_mtype(home) := Put$
$rule := NI_Local_Put$

Next State

The rule *NI_Local_Put* resets *dir_dirty*, thus giving contradiction.

Analysis

Initial state has *home* receiving *Put* message. But the only node which can send *Put* message to *home* is a *remote* node with *exclusive* copy (by rule *NI_Remote_Get*). And only this node can send a *WB* message too, because we can't have two nodes with exclusive copies. Also this node can send only one of the two messages, because immediately after sending these messages, the node drops its *exclusive* status. This suggests further strengthening of invariant F. So we have,

$$F_1 := (wbnet_mtype = WB) \Rightarrow ((shwbnet_mtype \mathrel{!}= ShWB) \ \& \ (unet_mtype(home) \mathrel{!}= Put)) \ ;$$

We decide this property F_1, but it gives a counterexample exactly as earlier, except this time we have *ShWB* message in the system in place of *WB* message.

Counterexample 12

This counterexample is exactly similar to 11, except we have $shwbnet_mtype := ShWB$ in place of $wbnet_mtype := WB$.

But we can't modify property F_1 to include $(shwbnet_mtype = ShWB)$ in the antecedent as it would directly conflict with the consequent $(shwbnet_mtype \mathrel{!}= ShWB)$. So we key in a new invariant, to state that we can't have *home* receiving *Put* message, if there is *WB* or *ShWB* message in the system. As we can see here, we had to develop a new invariant simply because its not possible to strengthen F_1 further. Also, since the new invariant would create redundancy with F_1, we simplify F_1 to its earlier version, viz., F.

$$G := ((wbnet_mtype = WB) \mid (shwbnet_mtype = ShWB)) \Rightarrow (unet_mtype(home) \mathrel{!}= Put) \ ;$$

This time, we succeed in proving property C_1 and UCLID outputs the for-

mula to be valid. We now venture to prove the invariant D_1.

D_1 := *Lambda* (*i*,*j*) . (*unet_mtype*(*i*) = *Put*) ⇒ ((*cache_state*(*j*) != *exclusive*) &
(*unet_mtype*(*j*) != *PutX*))

This was the most difficult invariant to prove. We get a counterexample on deciding the above property.

Counterexample 13
Initial State

i := *remote*, *j* := *home*
cache_state(*i*) = *cache_state*(*j*) := *shared*,
unet_mtype(*i*) := *Put*, *unet_mtype*(*j*) := *None*
dir_dirty := *false*
rule := *PI_Local_GetX*

So the initial state satisfies D_1, since there is no node in *exclusive* state or receiving *PutX* when some node has received a *Put* message.

Next State
Since the directory is not dirty and *home* wishes for exclusive access, it is granted. Thus, *cache_state*(*j*) := *exclusive*.

Analysis
The initial state seems to be a correct and reachable state, as it is possible to have a remote node receiving a *Put* message while directory is not dirty and there are shared copies in the nodes. So we strengthen D_1 further, providing *home* node the exemption.

D_2 := *Lambda* (*i*,*j*) . ((*j* != *home*) & (*unet_mtype*(*i*) = *Put*))
⇒ ((*cache_state*(*j*) != *exclusive*) &
(*unet_mtype*(*j*) != *PutX*));

And we test this new invariant. We obtain a different counterexample this time, suggesting that the strengthening was effective in getting rid of the earlier counterexample.

Counterexample 14
Initial State

i := r_1, *j* := r_2
cache_state(*i*) = *cache_state*(*j*) := *shared*,
unet_mtype(*i*) := *Put*, *unet_mtype*(*j*) := *GetX*
dir_dirty := *false*
rule := *NI_Local_GetX*

Next State
Since dirty flag is not set, the *home* node grants exclusive access to node *j*. So, *cache_state*(*j*) := *exclusive*

Analysis

Since it is possible to have a node requesting an exclusive access to the *home* and a *remote* node receiving a *Put* message in the network, the above counterexample can happen in reachable state space. So the invariant is not true for reachable states. We would have to refine it.

The essence of the invariant is the *Put* message. If we look at the source rule of the *Put* message to any node, it can be either from *home* node using rule *NI_Local_Get* or from *remote* node using rule *NI_Remote_Get*. In the case of the former, as we can see from the counterexample, its not necessary for directory to be dirty when *Put* message is sent and so some other node can be granted *PutX* message or *exclusive* state. But in the later case, the remote node is in *exclusive* state before sending the *Put* message and so the directory must be dirty thus forbidding *home* from granting *PutX* or *exclusive* state to any node.

We refine the invariant D_2 further, to have the *Put* message not being received from the *home* node. So, we have D_3 to be

$$D_3 := Lambda\ (i,j)\ .\ ((j\ !=\ home)\ \&\ (unet_mtype(i)\ =\ Put)\ \&\ (unet_proc(i)\ !=\ home))\ \Rightarrow\ ((cache_state(j)\ !=\ exclusive)\ \&\ (unet_mtype(j)\ !=\ PutX));$$

UCLID returns this refined invariant to be valid, but it generates a counterexample in invariant A_3. So we have to prove A_3 again. Let's recollect A_3.

$$A_3 := Lambda\ (i)\ .\ ((cache_state(i)\ =\ exclusive)\ |\ (unet_mtype(i)\ =\ PutX))\ \Rightarrow\ ((wbnet_mtype\ !=\ WB)\ \&\ (shwbnet_mtype\ !=\ ShWB)\ \&\ dir_dirty)\ ;$$

Counterexample 15

Initial State

$i := remote$,
$cache_state(i) := exclusive$,
$unet_mtype(i) := None$, $unet_mtype(home) := Put$,
$unet_proc(home) := home$,
$dir_dirty := true$,
$rule := NI_Local_Put$

Next State

$dir_dirty := false$

Analysis

The initial state deals with *home* receiving the *Put* message from itself. This is simply impossible, as *home* can receive *Put* message only from a *remote* node.

So we bring in a simple new invariant.

$$H := (unet_mtype(home)\ =\ Put)\ \Rightarrow\ (unet_proc(home)\ !=\ home)\ ;$$

And we test again the invariant A_3, and UCLID returns the formula to be valid.

Having proved A_3, B, C_1 and D_3 we move forward to prove the next invariant E. Recall that,

$$E := Lambda\ (i,j)\ .\ ((i\ !=\ j)\ \&\ (unet_mtype(i) = PutX)) \Rightarrow (unet_mtype(j)\ !=\ PutX)\ ;$$

We run UCLID to decide the above property This time, we are lucky and the property is returned to be valid in first attempt itself.

So, we move ahead to prove the simple property F.

$$F := (wbnet_mtype = WB) \Rightarrow (shwbnet_mtype\ !=\ ShWB)\ ;$$

We run UCLID to check this property. We get lucky second time, and the property is found to be true.

Next invariant to be proven is G.

$$G := ((wbnet_mtype = WB)\ |\ (shwbnet_mtype = ShWB)) \Rightarrow (unet_mtype(home)\ !=\ Put)\ ;$$

This time also, we are successful in proving without any hindrance, the invariant G. Next and hopefully the last invariant to be proven is H.

We check this using UCLID. And with the result being positive, we finish the innings of proving the safety property completely using just 8 invariants.

All the invariants and safety property were proved automatically by UCLID. It takes only 5.510 seconds to finish the proof. The UCLID code has 1000 lines, including the model, invariants and comments.

7 Conclusion

We have seen a formal proof of FLASH cache coherence protocol satisfying the mutual exclusion property using the method of inductive invariant checking in UCLID. The safety property was verified using just 8 auxiliary invariants. These invariants were generated very frugally and in a systematic way. We have kept the proof as simple and elegant as possible. We have seen how the automated verification engine of UCLID helped us in coming up with the auxiliary invariants. But, along with this advantage of having a fast and efficient verification engine, there are also disadvantages in terms of the UCLID language. The language does not provide conventional way of programming. As nested ITE's are not allowed, the user has to be very careful in dividing them and accordingly, assigning the state variables that fall within their scope. Also, since existential quantifiers are not allowed, it puts additional burden on modeling those state variables that are existentially dependent on state variables. We have considered such cases in FLASH, but those variables were existentially dependent on a single state variable. It would be much more complicated to define them, if they were existentially dependent on multiple state variables.

Our future work would be to apply this technique of discovering invariants in the verification of coherence property for FLASH protocol in delayed mode.

We believe that it is possible to prove coherence with fewer invariants. We are also interested in using this method for the verification of other complex protocols/systems. We also think that the tricks deployed in generating invariants can be generalized to be applicable on other complex systems/protocols. Such a generalization would help one in automating (or semi-automating) the invariant generation procedure. Also, it would help us in keeping the number of invariants as low as possible and sufficient enough to prove the desired safety property.

References

[1] R. Bryant, S. Lahiri, and S. Seshia. Modeling and verifying systems using a logic of counter arithmetic with lambda expressions and uninterpreted functions. In E. Brinksma and K. G. Larsen, editors, *Proc. 14th Intl. Conference*, 2002.

[2] Kenneth L Mcmillan. Parameterized verification of the flash cache coherence protocol by compositional model checking. In Tiziana Margaria and Thomas F. Melham, editors, *Proceedings of the CHARME 2001*, volume 2144 of *Lecture Notes in Computer Science*, pages 179–195. Springer, September 2001.

[3] Seungjoon Park and David L. Dill. Verification of flash cache coherence protocol by aggregation of distributed transactions. In *Proceedings of the eighth annual ACM symposium on Parallel algorithms and architectures*, pages 288–296. ACM Press, 1996.

[4] Randal E. Bryant Sanjit A. Seshia, Shuvendu K. Lahiri. *A User's Guide to UCLID version 1.0.* Carnegie Mellon University, Pittsburg, PA 15213, June 2003.