Bounded Transaction Model Checking

Xiaofang Chen and Ganesh Gopalakrishnan

UUCP-06-003

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

February 27, 2006

Abstract

Industrial cache coherence protocol models often have too many reachable states, preventing full reachability analysis even for small model instances (number of processors, addresses, etc.). Several partial search debugging methods are, therefore, employed, including lossy state compression using hash compaction, and bounded model checking (BMC, or equivalently, depth-bounded search). We show that instead of a BMC approach, a *bounded transaction* approach is much more effective for debugging. This is because of the fact that the basic unit of activity in a cache coherence protocol is that of a *transaction* - e.g., a complete causal cycle of actions beginning with a node making a request for a line and obtaining the line. The reduced effectiveness of BMC mainly stems from the fact that by limiting only the search depth, it cannot be guaranteed that complete transactions get selected, or that the right kind maximal number of interacting transactions.

Thus, instead of bounded model-checking, which explores all possible interleavings in BFS, we propose a bounded *transaction* model-checking approach for debugging cache coherence protocols, where the criterion is to allow a certain *number of transactions* chosen from a set of potentially interfering set of transactions, to be explored. We have built a bounded transaction version for the Murphi model checker and shown that it can find seeded bugs in protocols far more effectively, especially when full BFS runs out of memory and misses these bugs. We compare our work with similar ideas - such as debugging communicating push-down systems[1] by bounding the number of interleavings (a similar idea, but different in detail).

1 Introduction

Scalable industrial cache coherence protocols are very complex, often spanning 50 or more pages of "rule-style" descriptions written in languages such as Murphi, TLA+ [2], or Blue-Spec [3]. Even when running on small instances (number of processors, addresses, etc.) using today's enumerative model checkers under symmetry reduction[4] or other exact state reduction methods, their number of reachable states far exceed that available in today's large machines.

Abstraction/refinement based verification methods [5, 6], symbolic methods [7, 8], or symbolically assisted methods (e.g., [9]), while showing great promise, have not been demonstrated on industrial coherence protocols. Consequently, a number of limited search debugging methods are in use today, including bounded model checking (in the enumerative sense of depth-bounded searching), lossy state compression using hash compaction, etc.

We argue that the currently used methods for doing bounded search debugging of cache protocols do not use the available state resources wisely. This is because they employ criteria such as bounded model checking (BMC) which do not track the basic unit of "work" in these protocols - which is *transaction*. In most coherence protocols, transactions start with a node requesting a sharable or an exclusive copy of a cache line. Such requests typically go to directories which then forward the requests to the current sharers, collect invalidation acknowledgments (in case of an exclusive request), and send the line to the requesting node.

We observe that if such complete transactions are allowed to form during search, but we limit the *number* – and also *variety* – of such transactions that can form, the memory resources are far more efficiently utilized. For example, consider a model checker exploring a cache protocol running on four processors and having 10 different things that the processors can do using BFS: the search graph would begin with a fan-out of 40. A few frontiers of BFS later, the model-checker resources would be exhausted, with typically very few full transactions examined – and more importantly, with very few full transaction interactions examined. Naturally, bugs are missed, as we show in controlled experiments using seeded bugs.

In contrast, with a *bounded transaction* (BT) search, we allow full transactions to form, but limit their number and variety. This is very similar to what others have discovered in related contexts. For instance, in [1], the model checking of communicating push-down systems – where reachability is inherently undecidable [10] – is reduced to bounded interleaving checking. Their results show that many bugs are caught in this manner. There are however many differences. Limited process interleaving and bounding the number of transactions allowed to be alive at a point are related, but not the same idea.

In cache coherence protocols, behaviors are expressed in a "rule style" - as an apparently unordered collection of rules. This writing style promotes a declarative expression of the intent. In approaches such as [3], this declarative and "maximally concurrent" writing style does lead to opportunities for synthesizing efficient hardware for cache coherence controllers. Thus, a communicating sequential process style of writing is not preferred, as it may make the extraction of efficient hardware difficult. In addition, the rule-style writing style allows certain *cross-cutting* aspects of behavior to be naturally expressed. For instance, there are often rules that describe how "spent" messages (invalidations whose purpose has been lost) are to be flushed out. These rules can be stated orthogonal to rules that take a per-processor perspective.

Thus, the important question of *how to identify transactions* has to be addressed. In our BT approach and tool, we employ concrete executions in very small instances to determine potential transactions, as will be explained in Section 3.1. The last question is, of course, how one might make BT complete. While we don't have a solution, several ideas in this regard are expressed in Section 5.

2 Related Work

The transactional nature of cache coherence protocols was first described by Park and Dill[11]. They aggregated the implementation step of each transaction into a single atomic transaction in the specification. Completing(commit) step was defined as the implementation step which first causes a change in the specification variables. However, this approach does not consider any interleaving in the real implementation.

Several other searching heuristics have also been proposed to optimize model checking of cache coherence protocols. For example, Yang and Dill[12] proposed using the minimum hamming distance as a heuristic with the hope that states with very few bits differing from the error state will require fewer cycles to reach the target. Abts et al.[13] proposed that by choosing the next state via maximum hamming distance, the search will move toward the error state. They also proposed to use the "cache-score", which is a subset of the state information, to determine the best rule to fire. We think these criteria are too coarse to be applied on all protocols.

Recently, Bhattacharya et al.[9] also proposed to exploit the transactional nature of cache coherence protocols to aid partial order reduction. By selecting appropriate seed transitions in a transaction in the ample set computation, they can effectively take the current transaction forward and delay the scheduling of new transactions.

3 Bounded Transaction-Based Testing

To take advantage of the transactional nature of cache coherence protocols, `BT` only explores a subset of all possible next states for a given state s. This subset is chosen based on the type of the current transaction s is in, e.g. "shared request", and the interleavings the current transaction is allowed to interact with other new transactions. Once the upbound of the number of new transactions spawned by the current transaction is reached, `BT` will force all the transactions work toward finishing themselves. The states ending these transactions will constitute a terminal frontier, which will serve as the initial states for the next round of testing.

As a result, the subset of state space which `BT` chooses will not only contain full transaction activities, but also include interleavings among different transactions. In Section 4, we can see that `BT` is far more effective than the BFS model checking on seeded bugs in the German[14] and FLASH[15] protocols. In the following section, we will describe the implementations of `BT`.

3.1 Specifying Transactions

The natural "unit" of work in cache coherence protocols usually starts a node requesting a sharable or an exclusive copy of a cache line, or evicting a cache line from its local cache. Such requests typically go to directories which will then forward the request to the current sharers if necessary. The end of a transaction is indicated by a data copy or NACK message being sent back to the requester. Such a full transaction can be taken as a causal sequence of actions.

We can assume that protocol designers are very clear about the details of a protocol, so in a rule-based specification system it would be easy for them to write a simple file specifying which rules are the starting, or the completing transitions in a transaction. In the `BT` tool, we provide an interface $set\_srulesets()$ and a variable $s\_rulesets[]$ to let users specify transactions. A simple example is shown as following. Users simply need to replace "ruleset_num" with the total number of rulesets, and specify weights for transaction starters and enders. In this example, weight 1 is used for starting shared transactions, 2 for starting exclusive transactions, and 3 for completing transactions.

```
void set_srulesets() {
  for (int i = 0; i < ruleset_num; i ++)
    s_rulesets[i] = 0;

  // req: Shrd
  s_rulesets[9] = 1;      // SendReqS
  // req: Excl
  s_rulesets[5] = 2;      // SendReqE
  // completion
  s_rulesets[7] = 3;      // RecvGntE
  s_rulesets[2] = 3;      // RecvGntS
}
```

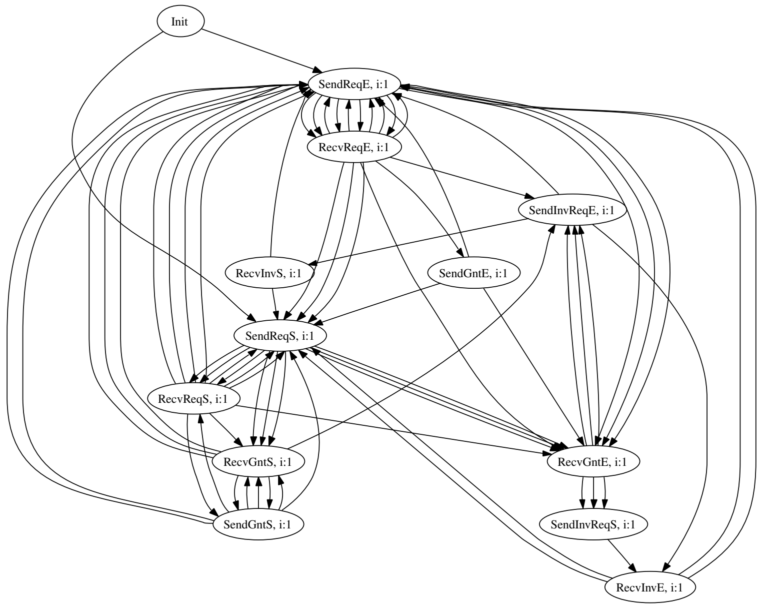

Figure 1: Rule firing sequence before pruning

Our BT tool also provides functionalities to obtain potential transactions in a protocol automatically. It employs concrete execution by running BFS of standard model checking, on a small instance of the model. The causal graph, which is the firing sequence of each rule in the concrete execution, is recorded and later simplified. The final result is a set of "pure" transactions in the graph. Take an example, Figure 1 shows a causal graph on a simple protocol with just one node. It records all the "$rule_i \rightarrow rule_j$" sequences, such that in the concrete execution $rule_j$ is fired right after $rule_i$ is fired.

To perform simplification on a causal graph, `BT` prunes duplicate edges and "backward" edges. Duplicate edges are a set of edges which fire a pair of two same rules, and backward edges are indicated by a rule firing another rule which is in a lower depth of the BFS execution. For example in Figure 1, "SendGntS,i:1 → SendReqS,i:1" is a backward edge as the BFS depth of "$SendGntS, i : 1$" is 3, higher than the depth of "$SendReqS, i : 1$" which is 1. The logic behind such pruning is that most backward edges will mix at least two transactions under BFS executions. So removing these edges will leave a set of "pure" transactions: from starting to ending transitions without interfering with other transactions. Figure 2 is the simplified causal graph on Figure 1. It shows two pure transactions, both starting with a request for a cache line, and ending with the request is granted. Interested readers please refer to[16] for more examples.

After the causal graph is simplified, we can take rules on depth 1 as transitions starting transactions, and rules on the leaf nodes as transitions ending transactions. Different values can be assigned as the weights of rules in these two categories. We believe this approach of determining transactions can be complete[1], by having multiple initial states other than the standard one where all caches are Invalid. For example, in the Flash protocol[15], we can have the following 5 initial states in a 3-node model, including $\{I, I, I\}$, $\{S, I, I\}$, $\{S, S, I\}$, $\{S, S, S\}$ and $\{E, I, I\}$. Here I, S and E represent $Invalid$, $Shared$, and $Exclusive$ individually.

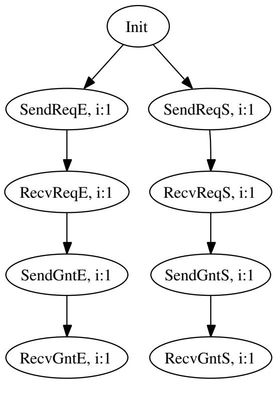

Figure 2: Rule firing sequence after pruning

[1]Thanks for the discussion with Dr. Ching-Tsun Chou.

3.2 Implementation

Assume we already have a rule weight file which specifies the starting and ending transitions of transactions in a protocol. We implemented the bounded transaction-based testing algorithm "$bounded\_test()$" in Murphi. At each state, the model checker first finds out all the enabled rules as BFS usually does, and then selects a subset of rules to fire. This subset is chosen according to the attribute of the current state, i.e. the transaction type, and the quota of leftover transactions to be spawned.

The attribute of a state is an extra part of information `BT` stores. For all initial states in model checking, they have no attribute. When a state is generated after a shared request, the state will have attribute "[sh]", and this attribute will be kept until a reply is sent back. If another exclusive request is made before the reply of the shared request, then the state attribute will be "[sh;ex]".

The quota of leftover transactions will decrease by 1 each time a new transaction is initiated during the lifetime of the current transaction. When its value reaches 0, the current transaction can only work toward finishing itself: no new transactions(i.e. interleavings) are allowed to be initiated. This approach allows `BT` to control how many interleavings one transaction can interact with others. For example, when the maximum quota equal to 0, `BT` is working on "complete sequential executions"; while the maximum quota equal to a enough big value, `BT` is simply doing BFS, enumerating all possible interleavings.

3.2.1 Notations and Definitions

Several notations are defined to help understand the algorithm of `BT` in **Algorithms**.

$$
\begin{array}{lcl}
trax & : & \texttt{Abbr. of ``transaction''} \\
Q & : & \texttt{state list for current round of testing} \\
Q' & : & \texttt{state list for next round of testing} \\
tr\_quota\_left & : & \texttt{the quota left to spawn new transactions} \\
Max\_Quota & : & \texttt{maximum value of quota\_left} \\
Max\_Round & : & \texttt{maximum rounds of testing} \\
enabled(s) & : & \texttt{all the enabled rules at state 's'} \\
attrib(s) & : & \texttt{attribute of state 's'} \\
 & & = \{[], [\texttt{sh}], [\texttt{ex}], [\texttt{sh; ex}], [\texttt{ex; sh}], [\texttt{ex, ex}]\}
\end{array}
$$

3.2.2 Algorithms

Following is the algorithm BT uses in $bounded\_test()$. As we can see, when the tr_quota_left equal to 0, only those rules that will finish the current transaction are allowed to be fired. And when a new transaction is initiated inside another transaction, the value of tr_quota_left is assigned to 0. This means that all possible interleavings within these two transactions are allowed, and only after both these transactions have finished, can the third transaction be initiated. This is a simple heuristic BT currently uses. However, allowing a third transaction to be initiated in the lifetime of two other transactions is also fine.

```
random_subset(rules, type, var quota) =
{  if (quota=0) then
      return rules except trax starters
   else
     match type with
      | sh_or_ex -> random choose 1 Excl trax starter +
                    random choose 1 Shrd trax starter +
                    rules  except all trax starters
      | ex       -> random choose 1 Excl trax starter +
                    rules  except all trax starters
      | sh       -> random choose 1 Shrd trax starter +
                    rules  except all trax starters
     quota := quota - 1
}

choose_S_or_E(attrib, rules, var tr_quota_left) =
{  match attrib with
     | []   ->  random_subset(rules, sh_or_ex, tr_quota_left)
     | [sh] ->  random_subset(rules, ex,       tr_quota_left)
     | [ex] ->  random_subset(rules, sh_or_ex, tr_quota_left)
     | _    ->  random_subset(rules, sh_or_ex, 0)
}

terminal(state) =
{  if (state is a trax ender and it can only start new trax) then
      return true
   else
      return false
}

BT (Q, tr_quota_left, round_left, Q') =
{  if Q=[] and round_left=0 then
```

```
      return
   else if Q=[] then
      BT(Q', Max_Quota, round_left-1, [])
   else
      s := front(Q);
      Q := rest(Q);
      if terminal(s) then
         BT(Q, tr_quota_left, round_left, append(Q',s))
      else
         let sub_rules = choose_S_or_E(attrib(s), enabled(s), tr_quota_left) in
         for each rule in sub_rules do
           let nexts = rule(s) in
           let attrib(nexts) =  modify_attribute(nexts) in
             append(Q, nexts)
         end
         BT(Q, tr_quota_left, round_left, Q')
}

BT( [init_state], Max_Quota, Max_Round, [] );
```

4 Experiment Results

We have applied BT on the Stanford FLASH protocol[15]. It was effective to find out the bug injected by removing the condition "$Sta.RpMsg[src].Cmd\,! =\ RP\_Replace$" in the guard of Ruleset "$NI\_Local\_Get\_Get$", where the shortest path to this bug is 18. Table 1 shows the result.

Table 1: Performance of BT on a buggy FLASH protocol

Node	Data	BT			Murphi BMC		
		# of states	Time(s)	Found bug	# of states	Time(s)	Found bug
3	2	106812	21.04	yes	213275	27.16	yes
5	2	404924	215.35	yes	3112000	821.74	no
8	3	389749	415.72	yes	1836000	1726.99	no
10	3	361344	465.81	yes	1507000	2417.51	no

We also applied BT on the German protocol[2] [14] devised by Steven German in 2004. The protocol contains 3 processor nodes and 2 addresses. A bug was injected by commenting

[2]No symmetry is used in this protocol.

the condition of "$if\ forall\ n:\ node\_id\ do\ directory[n] = cache\_invalid\ endforall$ in the ruleset "home processes invalidate acknowledgment". Table 2 shows the result.

Table 2: Performance of BT on a buggy German protocol

Node	Addr	BT			Murphi BMC		
		# of states	Time(s)	Found bug	# of states	Time(s)	Found bug
3	2	79854	14.97	yes	2187000	265.28	no
5	2	1404	4.51	yes	426000	46.62	no
8	2	1936	4.72	yes	169000	33.71	no
10	2	476	4.41	yes	116000	21.03	no

All the above experiments were run on Intel Xeon 3.06GHz with 1000MB memory. "no" in the "Found bug" column means the model checking runs out of resources. Maximum round of 6 and maximum quota of 1 are used in the experiments. Interested readers can refer to [16] on how transactions are specified on these protocols.

5 Conclusion

BT is a bounded transaction-based testing tool developed on Murphi. It does model checking by controlling the number of transactions and the types of interleavings among different transactions. The preliminary results on the FLASH and German protocol show that BT is an effective testing tool. We have also proposed several approaches to make it complete. These ideas include employing abstraction on certain nodes and keeping concrete information on other nodes, applying mover-based partial order reduction on protocols, and aggregating states which are not explored by BT, etc. These will be our future work.

References

[1] S. Qadeer and D. Wu. Kiss: Keep it simple and sequential. In *ACM SIGPLAN Conference on Programming Language Design and Implementation(PLDI)*, 2004.

[2] L. Lamport. Specifying concurrent systems with tla+. 1999.

[3] Arvind. Bluespec: A language for hardware design, simulation, synthesis and verification. In *MEMOCODE*, 2003.

[4] C. Norris Ip and David L. Dill. Verifying systems with replicated components in mur*hi*. In Rajeev Alur and Thomas A. Henzinger, editors, *CAV*, volume 1102 of *Lecture Notes in Computer Science*, pages 147–158. Springer, 1996.

[5] Satyaki Das, David L. Dill, and Seungjoon Park. Experience with predicate abstraction. In Nicolas Halbwachs and Doron Peled, editors, *CAV*, volume 1633 of *Lecture Notes in Computer Science*, pages 160–171. Springer, 1999.

[6] Ching-Tsun Chou, Phanindra K. Mannava, and Seungjoon Park. A simple method for parameterized verification of cache coherence protocols. In *Formal Methods in Computer-Aided Design*, 2004.

[7] Ásgeir Th. Eiríksson and Kenneth L. McMillan. Using formal verification/analysis methods on the critical path in system design: A case study. In Pierre Wolper, editor, *CAV*, volume 939 of *Lecture Notes in Computer Science*, pages 367–380. Springer, 1995.

[8] Ilan Beer, Shoham Ben-David, Cindy Eisner, Daniel Geist, L. Gluhovsky, Tamir Heyman, Avner Landver, P. Paanah, Yoav Rodeh, G. Ronin, and Yaron Wolfsthal. Rulebase: Model checking at ibm. In *CAV '97: Proceedings of the 9th International Conference on Computer Aided Verification*, pages 480–483, London, UK, 1997. Springer-Verlag.

[9] Ritwik Bhattacharya, Steven M. German, and Ganesh Copalakrishnan. Exploiting symmetry and transactions for partial order reduction of rule based specifications. In *13th International SPIN Workshop on Model Checking of Software*, March 2006.

[10] G. Ramalingam. Context-sensitive synchronization-sensitive analysis is undecidable. In *ACM Transactions on Programming Languages and Systems*, 2000.

[11] Seungjoon Park and David L. Dill. Verification of flash cache coherence protocol by aggregation of distributed transactions. In *Proceedings of 8th ACM Symposium on Parallel Algorithms and Architectures*, pages 288–296, June 1996.

[12] C. Han Yang and David L. Dill. Validiation with guided search of the state space. In *Proceedings of the 35th Annual Design Automation Conference (DAC98)*, June 1998.

[13] Dennis Abts, Ying Chen, and David J. Lijia. Heuristics for complexity-effective verification of a cache coherence protocol implementation. In *Laboratory for Advanced Research in Computing Technology and Compilers Technical Report N. ARCTIC 03-04*, 2003.

[14] Steven German and Geert Janssen. Tutorial on verification of distributed cache memory protocols. In *FMCAD*, 2004.

[15] Jeffrey Kuskin, David Ofelt, Mark Heinrich, John Heinlein, Richard Simoni, Kourosh Gharachorloo John Chapin, David Nakahira, Joel Baxter, Mark Horowitz, Anoop Gupta, Mendel Rosenblum, and John Hennessy. The stanford flash multiprocessor. In *Proceedings of the 21st International Symposium on Computer Architecture*, pages 302–313, 1994.

[16] http://www.cs.utah.edu/formal_verification/software/murphi/BT/.