Biased Model Checking Using Flows

Muralidhar Talupur¹ and Hyojung Han²

¹ Strategic CAD Labs, Intel

² University of Colorado at Boulder

Abstract. We describe two new state exploration algorithms, called *biased-dfs* and *biased-bfs*, that bias the search towards regions more likely to have error states using high level hints supplied by the user. These hints are in the form of priorities or markings describing which transitions are important and which aren't. We will then describe a natural way to mark the transitions using flows or partial orders on system events. Apart from being easy to understand, flows express succinctly the basic organization of a system. An advantage of this approach is that assigning priorities does not involve low level details of the system. Using flow-derived priorities we study the performance of the biased algorithms in the context of cache coherence protocols by comparing them against standard bfs, dfs and directed model checking. Preliminary results are encouraging with *biased-bfs* finding bugs about 3 times faster on average than standard bfs while returning shortest counter examples almost always. *Biased-dfs* on the other hand is couple of orders of magnitude faster than bfs and slightly faster than even standard dfs while being more robust than it.

1 Introduction

We present two new state exploration procedures, called *biased-bfs* and *biased-dfs*, that steer the search towards regions more likely to have error states by using high level user supplied hints. These hints take the form of marking transitions as important or not. In practice, model checking problems often have high level structure that is completely ignored by the standard model checkers. The basic premise behind our work is that this high level information, if used properly, can help model checkers scale to the complex real world systems.

More concretely, consider the class of distributed message passing protocols, an important and industrially relevant class of systems. These protocols are often built around a set of transactions or *flows* [12]. In other words, the protocol follows an implicit set of partial order on actions/events, such as sending and receipt of messages. Even very large industrial protocols have a fairly concise and easy to understand set of flows¹.

Empirically, not all the flows are equally critical to the functioning of a protocol. By identifying the important flows and eagerly exploring those regions of the state space where these flows are active we can increase the chances of hitting deep bugs. The information about important flows is transferred to the model checkers by marking (or assigning higher priorities) to actions involved in those flows.

¹ This is what makes designing a distributed protocol 5000 lines long possible in the first place.

Given a set of marked transitions biased model checking algorithms work as follows. Biased-bfs, a variant of breadth first search(bfs), begins by exploring the model in standard breadth first manner. But when computing the next frontier, that is, states reachable from the current frontier, it accumulates all the states that have marked transitions enabled. For each such state s , biased-bfs explores the part of state space reachable from s using only the marked transitions. All the states encountered during this specialized sub-search are added to the next frontier and the search continues from the next frontier. As an effect of firing marked transitions eagerly, the states reachable using those transitions are explored much quicker than other states.

While biased-bfs does not assume anything about the underlying system, biased-dfs on the other hand is targeted specifically towards systems composed of multiple agents executing asynchronously. For these systems it helps to not just explore the marked transitions early, but also explore the different inter-leavings of marked transitions early. For instance, suppose we have a system with two agents A and B . A trace $t_1^A, t_2^B, t_3^A, t_4^B, \dots, t_n^A$ with inter-leaved firing of important transitions of A, B is much more likely to have a bug than a trace $t_1^A, t_2^A, \dots, t_m^A, t_{m+1}^B, \dots, t_n^B$ that has a series of marked transitions of A followed by a series of marked transitions of B . To reach such traces quickly, biased-dfs starts like standard dfs but it avoids switching to a different agent unless the currently executing agent has reached a state where it has some marked transitions enabled. That is, biased-dfs forces each agent to a state where it has a marked transition enabled. Once a state s that has a desired percentage of agents with marked actions enabled is reached, a specialized sub search which does a more thorough exploration is invoked on s . Thus, biased-dfs tries to co-ordinate execution of multiple agents. Like biased-bfs, biased-dfs will explore a system exhaustively if let to run to completion.

An advantage of biased model checking is that, unlike heuristic functions in dmc, priority functions in our case do not involve low level details of the system. They simply amount to saying which transitions, more specifically which syntactic *rules* occurring in the protocol description, are marked. As we will demonstrate in Section 6, figuring out which transitions to mark is not hard.

1.1 Applications to Distributed Message Passing Protocols

In this paper we apply the above general algorithms to an important class of systems namely distributed message passing protocols, in particular, cache coherence protocols. For these protocols, we show how to derive markings on transitions using *flows* or partial orders on system events.

As shown in [12,7], these systems have implicit partial orders on system events, like sending and receiving messages, built into them. In fact, the protocol architects explicitly reason in terms of flows and *they are readily available in the design documents*.

Given the flows of a protocol, we can identify which flows are important for finding violations to properties we are checking and within the flows which specific rules or transitions are important. This involves some human insight – just as is in dmc – but since flows are high level artifacts involving only names of *rules* they tend to be far easier to reason about than the protocol itself.

1.2 Evaluation

We introduced a series of bugs into German's and Flash cache coherence protocols and evaluated the biased model checkers by comparing them against bfs, dfs and also dmc.

Our preliminary results indicate that biased-bfs runs around 3 times faster than standard breadth first search on deep bugs and returns shortest counter examples in most cases.

Biased-dfs, is much faster – in many cases up to couple of orders of magnitude faster – than standard bfs and faster than even dfs in most cases. Moreover, unlike dfs the performance of biased-dfs is stable and does not fluctuate too much with the order of transitions in the system.

The rest of the paper is organized as follows. The next section covers the related work and the section after that introduces the preliminary concepts and the system model. Section 4 presents the biased-bfs and Section 5 presents biased-dfs. The section following that presents our experimental results and Section 8 concludes the paper.

2 Related Work

The idea of providing guidance to state exploration is a natural one and several forms of guided model checkers have been proposed over the years, including works by Hajek [4], by Yang and Dill [13], work by Ravi and Somenzi [9] and by Edelkamp et al. on symbolic dmc [10] and on explicit dmc [3,2].

The key idea in dmc, the most well known of the above approaches, is to adapt the A* algorithm to state exploration. The A* algorithm works by reordering the work queue (containing the states to be explored) according a cost function, which is a measure of the “goodness” of the state. For dmc, the user has to provide a cost function that gives for each state an approximate measure of how close it is to a buggy state. Tuning the cost function provides a way for controlling the search. In theory we can use cost functions on states to mark transitions as well (by making the cost function correspond to the guard of a transition). But this will be very tedious in practice as the guards can be complicated.

Another work that prioritizes finding bugs over full coverage is *iterative context bounded execution* [6], which explores a concurrent system starting from traces with the lowest number of context switches and working upwards towards those with more context switches. Biased-dfs generalizes the algorithm in [6]: the main exploration loop of biased-dfs avoids switching contexts (i.e. the executing agent) till it reaches a state s where multiple marked transitions are enabled. Once at s , a different exploration mechanism is invoked that does a thorough exploration of states reachable from s . Thus, the main outer search follows the context switching order but the sub-search does not. If we did not specify any special transitions then biased-dfs would be same as context bounded execution.

Recently randomized exploration routines, such as Cuzz [5], have been successfully applied to concurrent software. The idea is to sample the different inter-leavings (for the same input to the system) efficiently. These methods have not yet been extended to message passing systems as far as we are aware.

BT Murphi [1] also uses the notion of transactions, which are similar to flows, to speed up bug hunting. But the notion of transaction is central to their algorithm which bounds the search by limiting the number and types of transactions active. Biased-bfs and biased-dfs on the other hand do not limit the number of flows active but only guide the search to those regions where more flows are active.

3 Preliminaries

Given a set of state variables V , each system P we consider is given as a collection $\mathcal{R}$ of *guard-action* pairs (called *rules* in Murphi). Each rule is of the form

$$rl : \rho \rightarrow a$$

where rl is the name of the rule, ρ is a condition on the current state and action a is a piece of sequential code specifying next state values of the variables in V . A subset of these rules are *marked* to indicate that they can be used to bias the state exploration.

For biased-dfs, we require that P be composed of a collection of asynchronously executing agents, not necessarily all identical, with indices from an ordered set $\mathcal{I}$. The examples considered in this paper are directory based cache coherence protocols, which consist of a collection of asynchronously executing (and identical) caching agents. Thus, the set of state variables of V is $\bigcup_i V_i$ where V_i is the state variables of agent i .

Semantically viewed, each system P is a quadruple of form (S, I, T, T^*) where S is a collection of states, $I \subseteq S$ is the set of initial states and $T \subseteq S \times S$ is the transition relation. In addition to these standard components, a subset $T^* \subset T$ consisting of *marked* transitions is also specified. Each transition $t \in T$ corresponds to unique a $rl \in \mathcal{R}$.

For the biased-dfs procedure the underlying system P is a collection of agents. Thus, $S = \prod_{i \in \mathcal{I}} S_i$ where each S_i is the set of states of agent i . We can then identify, for agent i , the set $T_i \subseteq T$ of transitions that change the state of agent i . Thus, $T = \bigcup_i T_i$. To keep the presentation of the biased-dfs algorithm simple, we assume that the T_i 's are disjoint, that is, each transition in T can affect the state of only one agent. Extension of our work to the general case is not difficult.

We denote by *next* the successor function that takes a collection of states $S' \subseteq S$ and a collection of transitions $T' \subseteq T$ and returns all states reachable from S' in one step using transitions from T' . If the set of transitions is not specified it is assumed to be T and we write $next(S')$ instead of $next(S', T)$. In case of singleton sets, we just write $next(s)$ instead of $next(\{s\}, T)$ and $next(s, t)$ instead of $next(\{s\}, \{t\})$.

4 Biased BFS Algorithm

Biased-bfs works by trying to reduce the number of times bfs has to compute the next frontier. The size of the new frontier in bfs increases very rapidly with the depth of the search and if biased-bfs is able to reduce this depth by even a small number, say 3 or 4, the savings in terms of states explored and time would be significant. This reduction in depth is achieved by eagerly exploring marked transitions before exploring others.

Given a system $P = (S, I, T, T^*)$ biased bfs works as follows: the main outer exploration loop, shown in Figure 1, does standard breadth first search starting with the initial set of states and computing the next frontier $next_front$ from the current frontier $curr_front$. While $next_front$ is being computed biased-bfs collects states that have marked transitions enabled into another queue $front_{red}$. Before going onto the next frontier, the outer loop invokes a reduced breadth first search bfs_{red} on $front_{red}$ which does bfs search using only marked transitions. The set of states reached by this reduced search are added to the next frontier and the outer loop then proceeds as before.

Note that the search frontier during biased-bfs does not grow by the shortest path order but parts of state space reachable using the marked transitions are explored faster than the rest of the state space. Another way to view is to consider the marked transitions as having 0 weight while computing the path length. With this definition of shortest path the states are indeed explored according to the shortest path order.

The reduction in the search depth is obtained at the expense of making $next_front$ larger (by adding states returned by bfs_{red}). Having more marked transitions means more transitions in a counter example trace are likely to be marked, which means it will be found in fewer iterations of the main **while** loop in Figure 1. But, as the number of markings goes up, the number of states bfs_{red} returns will also go up. This would make $next_front$ large and negate the benefits of reduction in search depth. Thus, we have to mark as few rules as possible while marking as many of the important ones, that is, the ones that are likely to be present on counter example traces, as possible.

Theorem 1. *Biased bfs algorithm given above terminates for finite systems and fully explores the state space.*

Note that biased-bfs makes no assumption about the underlying system except for assuming that some of the transitions are marked. If no transitions are marked then it reduces to normal bfs and if all transitions are marked then its equivalent to doing bfs twice.

4.1 Optimizations

There are two different ways of realizing the basic outline presented above: we can either accumulate all the states in current frontier that have marked transitions enabled and call reduced bfs on them in one step (as shown in Figure 1) or call reduced bfs on the individual states as and when they are discovered. We found that the version presented is slightly faster than the greedy version.

Adding $front_{add}$ to $next_front$ can make it grow quickly, especially if there are many states with marked rules enabled in the current frontier. This growth can be controlled by limiting how many states are added to $front_{red}$ and indirectly limiting the size of $front_{add}$. Remarkably, we found during our experiments that limiting the size of $front_{red}$ to just 5 states gives all the benefits of biasing!

5 Biased DFS Algorithm

A straight-forward analogue of biased-bfs would be to modify standard dfs so that it fires marked rules first before firing un-marked rules. This would be equivalent to sorting the list of transitions so that marked rules appear before unmarked rules. But most

```

Function  $bfs_{red}(front_{red})$  {
   $curr\_front_{red} = front_{red}$ 
   $next\_front_{red} = \phi$ 
   $front_{return} = front_{red}$ 
  while ( $curr\_front_{red} \neq \phi$ ) do
    while ( $curr\_front_{red} \neq \phi$ ) do
      pick  $s$  from  $curr\_front_{red}$ 
      if ( $s \notin visited_{red}$ ) then
        add  $s$  to  $visited_{red}$  and
        to  $front_{return}$ 
         $successors\_set = next(s, trans_{red})$ 
        add  $successors\_set$ 
        to  $next\_front_{red}$ 
      end if
    end while
     $curr\_front_{red} := next\_front_{red}$ 
  end while
  return  $front_{return}$ 
}

Procedure  $biased\_bfs()$  {
   $visited = \phi$ 
  {Set of visited states}
   $curr\_front = I$ 
  {Queue holding states in the current
   frontier. Holds initial states at start}
   $next\_front = \phi$ 
  {Queue holding states in the
   next frontier}
   $visited_{red} = \phi$ 
   $curr\_front_{red} = \phi$ 
   $next\_front_{red} = \phi$ 
  while ( $curr\_front \neq \phi$ ) do
    while ( $curr\_front \neq \phi$ ) do
      pick  $s$  from  $curr\_front$ 
      if ( $s \notin visited$ ) then
         $successors\_set = next(s)$ 
        add  $successors\_set$ 
        to  $next\_front$ 
        if ( $s$  has marked transitions
         enabled) then
          add  $s$  to  $front_{red}$ 
        end if
      end if
    end while
     $front_{add} = bfs_{red}(front_{red})$ 
    add  $front_{add}$  to the front
    of  $next\_front$ 
     $curr\_front := next\_front$ 
     $next\_front := \phi$ 
  end while
}

```

Fig. 1. Biased-bfs algorithm

bugs in concurrent protocols happen when multiple processes & multiple flows are active in a trace and this procedure would only ensure that marked rules for an agent get fired before other rules but would not ensure co-ordination between agents.

Consider for instance a system S with three processes A, B and C . Consider a state (a^*, b, c) , where a, b, c are local states of A, B, C and $*$ denotes that agent A has a marked transition enabled. Naive biasing will result in marked transitions of A being greedily fired first. Consequently, the traces we will see first will have a stretch of marked transitions from A perhaps followed by a stretch of marked transitions from B and so on. The traces with tight interleaving of marked transitions from A, B, C won't be seen until the former traces have been explored whereas the correct thing to do would be to explore the interleaved traces first.

Biased-dfs presented in Figure 2 rectifies this by actively seeking out and exploring regions where multiple processes with marked rules enabled are present. It accomplishes this by having two sub-routines, namely, *bounded-execution*, which seeks out states with multiple processes having marked rules enabled, and *dfs-explore*, which does a thorough exploration once such a state is found.

To see how *bounded-execution* works consider a state (a_0, b_0, c_0) in which none of the processes has a marked transition enabled. *Bounded-execution* starts executing

transitions of A until it reaches a state (a_m^*, b_0, c_0) where A has a marked rule enabled. Now it switches to executing transitions from the next process², B in this case, until the system reaches a state (a_m^*, b_m^*, c_0) where A, B both have marked transitions enabled. Depending on how many processes in marked states we consider sufficient, *bounded-execution* can either switch to executing transitions from process C or call *dfs-explore* to thoroughly explore inter-leavings of A, B starting from (a_m^*, b_m^*, c_0) . Once *dfs-explore* is invoked on a state s it explores the descendants of s using all the transitions stopping only at those states in which none of the processes has any marked transitions enabled. All these border states at which *dfs-explore* stops are transferred over to *bounded-execution* which again tries to find interesting states starting from them.

Note that the set of transitions *bounded-execution* picks to explore at a state depends on the process it is currently executing. So in addition to storing states in the visited state set (*visited* in Figure 2) we also need to store for which ids a state can be considered visited. Thus, at a state $s = (a_m, b_n, c_k)$ if *bounded-execution* is currently executing transitions from A then (s, A) is added to *visited*. The exploration of transitions of B, C starting from s is deferred to later by adding $(s, B), (s, C)$ to *next_stage*. Intuitively, (s, B) and (s, C) would correspond to switching contexts at s and these are deferred to later.

Unlike *bounded-execution*, *dfs-explore* considers all the transitions enabled at a state. If any of them is a marked transition then all the transitions are taken. If it reaches a states $s = (a_m, b_n, c_k)$ where none of the processes has a marked transition enabled then *dfs-explore* adds $(s, A), (s, B)$ and (s, C) to *curr_stage* for *bounded-execution* to explore further. Since *dfs-explore* does not track which process it is executing, once it is done all the states it visited will be added to *visited* paired with all possible ids.

Finally, the main procedure *biased-dfs* in Figure 2 initializes *curr_stage* with all initial states paired with all process ids and invokes *bounded-execution* on them. Note that if no transition is marked then *dfs-explore* never gets invoked and the resulting search is just context bounded execution [6]³.

Theorem 2. *Biased dfs terminates and explores all the states of a given model.*

6 Flows and Markings

For an important class of systems, namely distributed message passing protocols, we show a simple way to derive markings on transitions by considering *flows* [12]. Recall that the systems we consider are described as a collection of guarded action *rules*. A flow [12] is simply a partial order on interesting system events or rule firings.

Figure 3 shows a couple of flows from German's cache coherence protocol. The *ReqShar* flow on left says that first event is agent i sending a request for shared access (by firing rule *SndReqS*). The second event is directory receiving that request (by firing *RecvReqS*), followed by directory granting the access to i (by firing *SndGntS*). The flow terminates with i firing *RecvGntS* to receive the grant message. The *ReqExcl* flow on right is similar.

² Bounded execution also switches to a different process if the current process reaches a dead-end state.

³ If all the transitions are marked then biased-dfs reduces to plain dfs.

```

Function bounded-execution (st, pid) {
  if ((st,pid)  $\in$  visited) then
    do nothing
  else
    if (number of processes with marked transitions
      enabled in st crosses threshold)
      then dfs-explore(st)
    else
      let  $trans_{en}$  = all transitions enabled
        for process pid in state st
      if ( $trans_{en} = \phi$ ) then
        if (pid is not the last id in  $\mathcal{I}$  and
          next id is pid')
          for all(id  $\in$  pids and
            id notin {pid,pid'}) do
            add (st,id) to nextstage
          end for
          add (st,pid) to visited
          bounded-execution(st,pid')
        else
          return
        end if
      end if
    else
      for all(id  $\in$  pids) do
        add (st,id) to nextstage
      end for
      add (st,pid) to visited
      for all(t  $\in$   $trans_{en}$ ) do
         $nxt\_st = next(st,t)$ 
        bounded-execution( $nxt\_st$ ,pid)
      end for
    end if
  end if
}

Function dfs-explore (st) {
  if ( $\forall$  id. (st,id)  $\notin$  visited) then
    if (no process has a marked transition
      enabled in state st) then

```

```

    for all(id  $\in$  pids) do
      add (st,id) to currstage
    end for
  else
    for all(id  $\in$  pids) do
      add (st, id) to visited
    end for
    let  $trans_{en}$  = enabled transitions
      in state st
    for all(t  $\in$   $trans_{en}$ ) do
       $nxt\_st = next(st,t)$ 
      dfs-explore( $nxt\_st$ )
    end for
  end if
}

Procedure biased-dfs ( ) {
  visited =  $\phi$ 
  {Set of (state,id) pairs visited so far}
  currstage =  $\phi$ 
  {Queue of (state,id) pairs to explore
  in the current stage}
  nextstage =  $\phi$ 
  {Queue of pairs deferred to the next
  stage}
  for all(id  $\in$  pids) do
    for all( $st_{init} \in \mathcal{I}$ ) do
      add ( $st_{init}$ , id) to currstage
    end for
  end for
  while (currstage  $\neq \phi$ ) do
    while (currstage  $\neq \phi$ ) do
      pick (st,id) from currstage
      bounded-execution(st,id)
    end while
    currstage = nextstage
    nextstage =  $\phi$ 
  end while
}

```

Fig. 2. Biased-dfs algorithm

It is easy to see that flows capture succinctly and intuitively the basic organization of the protocol in terms of events such as sending and receiving of messages. In fact the protocol designers explicitly use the flows during design and for the distributed protocols we are interested *the full set of flows are already available in design documents*. So there is no effort on our part in coming up with the flows.

Fig. 3. Linear sequence events illustrating *ReqShar* (left) and *ReqExcl* (right) flows in German's protocol

Finally, even a large industrial protocol has only about 40 different flows and the full set of flows is orders of magnitude smaller than the protocol itself. Reasoning in terms of flows consequently tends to be a lot simpler than reasoning in terms of the protocol variables. A full treatment of flows is beyond the scope of this paper and the reader is referred to [12,7] for more information. For the rest of this section, we will assume we have the full set of flows available.

Using these flows, we identify which rules to mark as important and this in turn gives us markings for transitions with the interpretation that if a rule rl is marked then all the transitions in T corresponding to rl are marked.

Note that more the number of marked rules greater the overhead on biased model checkers. So we should to mark only a small number to reduce the overhead while ensuring that as many of the “important” ones as possible are marked. To mark rules we proceed backwards from the property which are verifying. The property we considered was the standard MESI property

$$\forall i, j. i \neq j \Rightarrow (Excl[i] \Rightarrow \neg(Shar[j] \vee Excl[j]))$$

where $Shar[i]$ mean i is in *shared* state and similarly for $Excl[i]$.

Markings for biased-bfs. Given flows and the property to be verified, we identify the flows that are likely to involved in the potential violations of the property. In our case, a counter example to safety property is most likely to involve the flow for getting exclusive access as at least one agent must be in exclusive state. For German's protocol this is *ReqExcl* flow and the rules present in it, namely, *RecvReqE*, *SendGntE* and *RecvGntE* are marked. We did not mark the first rule, *SndReqS*, for sending request, because this rule can fire without any constraint, and the core of the flow begins with directory receiving the request.

For Flash protocol, *ni_local_getx_putx* (corresponding to the directory receiving a request for exclusive and sending a grant message) and *ni_remote_putx* (corresponding to an agent receiving a grant exclusive message) were marked. The first rule for sending the request is not marked again. Thus, for German's protocol 3 out of 12 rules were marked and for Flash 2 out of 33 were marked. The Murphi models of the protocols along with the bugs we introduced are available at [11].

Markings for biased-dfs. Markings are derived slightly differently in this case. Here the focus is on multi-agent interactions that are likely to expose safety violations. So we identify flows whose interaction is likely to cause property violations. In our case, the violation to mutual exclusion will most likely involve flows for gaining exclusive and shared accesses, we mark rules appearing in these flows. For German's this means, in addition to the above marking, we should also mark *RecvReqS*, *SndGntS*, *RecvGntS* and for Flash *ni_local_get_put* and *ni_remote_put* in addition to the ones above. To keep the experimental section brief we will use the markings used for biased-bfs for biased-dfs as well. Even with these sub-optimal markings biased-dfs performs extremely well.

Finally, note that to derive markings we are reasoning based on the property and not based on any specific bug. Though deriving markings requires user intuition, having flows makes the job quite a lot simpler.

7 Experimental Evaluation

We wrote prototype implementations of the biased procedures and also of the bfs and dfs routines in OCaml. While OCaml is relatively slow and not amenable to low level optimizations required to make model checkers run fast, it is adequate to study the relative performance of the biased procedures versus the standard procedures. There were no optimizations except for symmetry reduction which was common to all procedures. The input protocols models were also in OCaml – they were translated from Murphi to OCaml automatically using a translator we wrote.

To study the performance of the biased procedures, we introduced bugs into German's and Flash protocols. These are standard examples in protocol verification and are non-trivial protocols – in terms of lines of Murphi code German's protocol is about 150 lines and Flash about 1000 lines. The bugs were introduced consecutively and they all led to violations of the MESI property.

7.1 Results

Tables 1 and 2 summarize the results of our experiments for German's and Flash protocols respectively. Each entry is either of the form trace-length/time-in-seconds or gives the number of states in thousands (k) rounded to the nearest integer. Apart from considering different bugs, we also considered systems with 4, 5, and 6 agents to study the scaling behavior of the biased procedures. The columns *bbfs*, *bdfs* and *dmc* stand for biased-bfs, biased-dfs and directed model checking respectively. Fastest running times are shown in bold and the second fastest times are italicized. All the experiments were run on a 2.6 GHz and 8GB Intel Xeon machine with a time out of 10 mins.

Performance of bfs. Standard bfs finishes on all the examples and in one case (Flash bug 3) it is close to being fastest. Apart from returning the shortest counter example always, performance of bfs is predictable and does not vary much with the order in which the transitions are presented. The markings on transitions have no effect on the performance of bfs.

Performance of dfs. Standard dfs, when it finishes within the time out, finds bugs extremely quickly but the counter examples it returns are extremely long. The other major disadvantage of dfs is that its performance fluctuates wildly with the order in which the transitions are listed. For instance, when we reverse the order of the transitions for Flash protocol, bugs 1 and 2 were caught by dfs in under a second with relatively short counter examples (< 25 states long). But performance on bug 3 deteriorated dramatically, with dfs timing out on 6 agent system and on 5 agent system it returned a counter example that was more than 50K states long after 130 seconds.

Performance of biased-bfs. Biased-bfs also finishes on all the examples within time out and it returns the shortest counter examples in all the cases. On two bugs (Flash bugs 1 and 2) it is the second fastest.

Significantly, biased-bfs outperforms standard bfs on the deeper bugs (bugs 1 and 4 in German and bugs 1 and 2 in Flash) by a factor of around three and this factor gets better as the number of processes increases. On shallower bugs standard bfs is faster than biased bfs.

This result validates the basic premise behind biased-bfs: reducing the number of frontier expansions required by standard bfs search leads to bugs quicker but at the cost of increased frontier sizes. Thus, for deeper bugs we see big gains for biased-bfs but not so much for shallower bugs, where the increase in frontier sizes seems to outweigh the decrease in the exploration depth.

While depth-first routines outperform biased-bfs, performance of biased-bfs is more robust and predictable with all the bugs being found within the time out. Further, counter examples returned by biased-bfs are considerably shorter. In fact, on all the examples, biased bfs returns the shortest counter examples (though this need not be the case always).

Performance of biased-dfs. Biased-dfs is the fastest in most cases except for bug 2 in German (where it is the second fastest) and bug 3 in Flash. It finishes within time out on all cases except Flash bug 3 on system with 6 agents.

Further, biased-dfs is less fickle than standard dfs. To see if the order of transitions has any effect on biased-dfs, we reversed the transitions lists in both German and Flash protocol. Biased-dfs continued taking around the same time as before, indicating that it is relatively immune to changes in the transition order.

The only disadvantage of biased-dfs is that it does not return short counter examples.

Performance of dmc. The original paper on dmc [3] presented several different variations of dmc with varying performance different examples. We implemented the straight forward adaptation of A* algorithm. The priority of each state is given by 100 minus

Table 1. Results for German's protocol

Bug procs	Trace length and run times					States explored			
	bfs	bbfs	dfs	bdfs	dmc	bfs	bbfs	dmc	
1	4	12/7.8	12/2.8	70/0.06	>75/ 0.05	12/24	19k	6k	25k
	5	12/38.2	12/10.6	86/0.09	>91/ 0.07	12/111	60k	15k	97k
	6	12/249.3	12/55.6	102/0.16	>107/ 0.1	t.o	172k	37k	-
2	4	9/1.0	9/1.6	34/ 0.05	>23/0.05	13/6	2k	3k	9.2k
	5	9/3.14	9/5.4	35/ 0.05	>24/0.06	10/23.5	5k	8k	34k
	6	9/14.6	9/24.5	36/ 0.11	>25/0.15	13/135	10k	17k	118k
3	4	9/1.1	9/1.6	58/0.06	>28/ 0.05	13/4.4	2k	3k	7k
	5	9/3.5	9/5.3	71/0.09	>28/ 0.06	11/16	5k	8k	25k
	6	9/16.7	9/24.5	84/0.15	>28/ 0.1	14/82	12k	17k	92k
4	4	12/9.3	12/3.3	240/0.17	>31/ 0.1	14/29	22k	7k	34k
	5	12/45.6	12/12.5	256/0.22	>31/ 0.13	13/147	69k	17k	133k
	6	12/283.5	12/62.1	272/0.36	>31/ 0.2	t.o	193k	40k	-

the number marked transitions enabled at the state. Thus, larger the number of marked transitions enabled the lower the priority and the earlier the state will be picked for consideration. As Tables 1 and 2 show dmc comes last in almost all the examples. The first four routines, bfs, bbfs, dfs, bdfs all use the same data structure to store the states and perform similar state related operations. So the time taken is also a good indication of number of states explored. But the underlying data structure used by dmc⁴ is different and to get another view we compare the number of states explored by bbfs and dmc. We don't show numbers for dmc as in almost all the cases the entry for bdfs would be 0k after rounding to nearest integer. As the last two columns of the tables show dmc explores a lot more states than bbfs before finding the bug⁵.

7.2 Biased Procedures in Murphi

Encouraged by the performance of OCaml implementations, we recently modified the Murphi model checker to perform biased-bfs, biased-dfs. This speeds up the performance considerably and also allows us to access a wider set of models.

We tried out modified Murphi on a large micro-architectural model of the core-ring interface (CRI) used in one of the multi-processors developed at Intel. The model, written in Murphi, captures the various in and out buffers, arbiters and other control structures managing the traffic between a core and the communication ring. The model also includes L1 and L2 caches of the core and is parameterized by the number of addresses

⁴ It requires a structure for storing sets of states that supports membership queries and minimum priority extraction.

⁵ The amount of computation for each visited state also differ with dmc taking the most resources.

Table 2. Results for Flash protocol

Bug procs	Trace length and run times						States Explored		
	bfs	bbfs	dfs	bdfs	dmc		bfs	bbfs	dmc
1	4	9/2.0	9/0.9	t.o	>13/ 0.04	>8/2.6	2k	0k	3k
	5	9/7.3	9/2.5	t.o	>13/ 0.05	>8/7.4	3k	1k	8k
	6	9/23.7	9/7.6	t.o	>13/ 0.1	>8/30	7k	2k	22k
2	4	9/5.1	9/1.8	t.o	>23/ 0.04	>5/4.8	3k	1k	5.5k
	5	9/21.4	9/5.8	t.o	>23/ 0.07	>5/15.4	9k	2k	15k
	6	9/77.6	9/20.6	t.o	>23/ 0.15	>5/65	21k	5k	45k
3	4	7/0.54	7/1.4	119/ 0.12	>11K/6.7	7/1.1	0k	1k	1k
	5	7/1.5	7/4.6	476/ 0.43	>34K/ 34.9	7/3.3	1k	1k	3k
	6	7/4.6	7/13.5	1996/ 2.8	t.o	7/13	3k	3k	9k

in the system. The property we considered was the inclusion property, namely, if a data item is present in L1 cache then it is present in the L2 cache as well.

We asked the designer who wrote the model to specify 4 rules that he thought were most relevant to the inclusion property and also to introduce a bug into the model. Coming up with 4 rules took the designer, who did not know the details of our approach, just couple of minutes.

Table 3 shows the result of this experiment. We ran bfs, dfs and biased-bfs on models with 4, 6 and 8 addresses with a time out of 20 hrs. Biased-dfs is not applicable to this model which deals with a single agent system. The table shows the number of states explored by bfs, dfs and biased-bfs. With the exception of the model with 4 addresses, bfs runs out of memory on all cases (after exploring considerably more states than biased-bfs) even when Murphi is given 12GB of memory. Interestingly, dfs hits the bug fairly quickly after exploring between 2-4M states but it fails to generate a counter example trace even after 20 hrs. This is because Murphi has to undo the symmetry reduction applied during the search phase to generate a counter example and for large models with deep counter examples this can be a very expensive operation⁶. In contrast, even for the largest model, biased-bfs generates a counter example trace in under 3.5 hrs after exploring less than 20M states. This result confirms that biased model checking does scale to industrial strength systems.

7.3 Discussion

Experimental results presented above amply demonstrate that the biased procedures can be very effective in catching bugs. When dealing with systems with asynchronously executing agents, biased-dfs is probably the best choice for checking existence of bugs. Random simulation is often used during early debugging but, at least for concurrent

⁶ This is a relatively unknown facet of dfs that only people who have dealt with large models seem to be aware of.

Table 3. Results for CRI model

Addr	bfs	dfs	bbfs
4	16M	t.o	12M
6	>25M	t.o	17M
8	>25M	t.o	18M

protocols, they are good only at finding shallow bugs and quickly get overwhelmed by the non-determinism.

For debugging, which requires short counter example traces, biased-bfs seems to be the best bet, especially for deep bugs. Apart from getting significant speed up compared to standard bfs and also standard dfs on some examples, biased bfs returns more or less the shortest counter example.

8 Conclusion

We have described two new biased model checking algorithms that use high level information implicit in protocols to reach error states faster. The mechanism of marking transitions as important or not is used to pass the high level information in flows to the biased model checkers.

Our early results indicates that the direction holds promise in tackling large model checking problems. A natural extension of this work is to combine priorities on transitions with priorities on states. This will provide us finer control on the exploration process potentially leading to an even more powerful method.

For instance, consider communication fabrics on a modern multi-core processor. One worrying scenario is the *high water-mark* scenario where all the buffers are full, potentially resulting in a deadlock. To bias the search towards such scenarios we can mark all those actions that *add* or *delete* some entry from a buffer. This will bias the search towards the corner cases where the buffer is full or empty. To gain finer control we can make the marking dependent on the state as well. Thus, an add action will be marked only in states where the buffer is already close to being full and a delete action will be marked only for those states where the buffer is close to empty. This will ensure that we hit the corner case scenarios and potential bugs even more quickly.

References

1. Chen, X., Gopalakrishnan, G.: Bt: a bounded transaction model checking for cache coherence protocols. Technical report UUCS-06-003, School of Computing, University of Utah (2006)
2. Edelkamp, S., Leue, S., Lluch-Lafuente, A.: Directed Explicit state model checking in validation of Communication protocols. In: Proceedings of STTT (2004)

3. Edelkamp, S., Lluch-Lafuente, A., Leue, S.: Protocol Verification with Heuristic Search. In: AAAI Symposium on Model-based validation of Intelligence (2001)
4. Hajek, J.: Automatically verified data transfer protocols. In: Proceedings of the International Computer Communications Conference (1978)
5. Musuvathi, M., Burckhardt, S., Kothari, P., Nagarakatte, S.: A randomized scheduler with probabilistic guarantees of finding bugs. In: ASPLOS (2010)
6. Musuvathi, M., Qadeer, S.: Iterative context bounding for systematic testing of multithreaded programs. In: PLDI (2007)
7. O’Leary, J., Talupur, M., Tuttle, M.R.: Protocol Verification using Flows: An Industrial Experience. In: Proc. FMCAD (2009)
8. Qadeer, S., Wu, D.: Kiss: keep it simple and sequential. In: PLDI (2004)
9. Ravi, K., Somenzi, F.: Hints to accelerate Symbolic Traversal. In: Proceedings of the 10th IFIP WG 10.5 Advanced Research Working Conference on Correct Hardware Design and Verification Methods (1999)
10. Reffel, F., Edelkamp, S.: Error detection with directed symbolic model checking. In: Woodcock, J.C.P., Davies, J. (eds.) FM 1999. LNCS, vol. 1708, pp. 195–211. Springer, Heidelberg (1999)
11. Talupur, M.: Murphi models of German’s and Flash protocols along with the bugs, www.cs.cmu.edu/~tmurali/fmcad10
12. Talupur, M., Tuttle, M.R.: Going with the Flow: Parameterized Verification using Message Flows. In: Proc. FMCAD (2008)
13. Yang, C.H., Dill, D.L.: Validation with guided search of the state space. In: Proceedings of the 35th Annual Design Automation Conference (1998)