CMC: A Pragmatic Approach to Model Checking Real Code

Madanlal Musuvathi*, David Y.W. Park†, Andy Chou, Dawson R. Engler, David L. Dill

{madan, parkit, acc, engler, dill}@cs.stanford.edu
Computer Systems Laboratory
Stanford University
Stanford, CA 94305, U.S.A

Abstract

Many system errors do not emerge unless some intricate sequence of events occurs. In practice, this means that most systems have errors that only trigger after days or weeks of execution. Model checking [4] is an effective way to find such subtle errors. It takes a simplified description of the code and exhaustively tests it on all inputs, using techniques to explore vast state spaces efficiently. Unfortunately, while model checking systems code would be wonderful, it is almost never done in practice: building models is just too hard. It can take significantly more time to write a model than it did to write the code. Furthermore, by checking an abstraction of the code rather than the code itself, it is easy to miss errors.

The paper's first contribution is a new model checker, CMC, which checks C and C++ implementations directly, eliminating the need for a separate abstract description of the system behavior. This has two major advantages: it reduces the effort to use model checking, and it reduces missed errors as well as time-wasting false error reports resulting from inconsistencies between the abstract description and the actual implementation. In addition, changes in the implementation can be checked immediately without updating a high-level description.

The paper's second contribution is demonstrating that CMC works well on real code by applying it to three implementations of the Ad-hoc On-demand Distance Vector (AODV) networking protocol [7]. We found 34 distinct errors (roughly one bug per 328 lines of code), including a bug in the AODV specification itself. Given our experience building systems, it appears that the approach will work well in other contexts, and especially well for other networking protocols.

1 Introduction

Complex systems have complex errors. Real systems have a variety of mishandled corner cases triggered by intricate sequences of events. In practice, this leaves a residue of errors that cause system crashes but only after days or weeks of continuous execution. When detected, such problems are often very difficult to diagnose because the errors are not reproducible and the sequence of events leading to them cannot be reconstructed.

Formal verification methods are a possible way to find and diagnose such deep errors [23, 24, 29]. One option is explicit model checking, which systematically enumerates the possible states of the system. A basic model checker starts from an initial state and recursively generates successive system states by executing the nondeterministic events of the system. States are stored in a hash table to ensure that each state is explored at most once. This process continues either until the whole state space is explored, or until the model checker runs out of resources. When it works, this style of state graph exploration can achieve the effect of impractically massive testing by avoiding the redundancy that would occur in conventional testing.

Conventional model checkers usually assume that the design is described at a high level that abstracts away many details of the actual implementation. Verifying actual code using such a tool requires reconstructing this abstract description from the code. This process requires a great deal of manual effort,

*Supported by GSRC/MARCO Grant No:SA3276JB

†Supported under a National Science Foundation Graduate Research Fellowship

hampering the use of model checking in actual system design. Moreover, human errors in the manual abstraction process result in missing bugs and cause false alarms during verification, further increasing the cost and reducing the usefulness of model checking. Such errors can be introduced both when constructing the model and as a result of "drift" as the actual system evolves [5]. For these reasons, it is a notable curiosity when software is model checked, rather than an everyday occurrence.

We introduce CMC (C Model Checker) to address some of these issues. CMC works on unmodified C or C++ implementations and explores large state spaces efficiently by storing states. Like traditional model checkers, CMC achieves the equivalent of executing astronomical numbers of tests in reasonable time. However, CMC does not require writing a separate high-level model of the code, nor extracting such a model from an implementation. More importantly, it finds the bugs that are actually *in* the implementation: it does not miss implementation bugs that would be omitted from a model, nor does it waste the user's time with bugs that appear in the model but *not* in the implementation.

The idea of model checking actual implementation code has been advocated in a small number of other tools. Verisoft [15], for instance, systematically executes C implementation code but does not store states. Other software model checking tools such as [3] [25] are specialized to work only with certain classes of Java programs. CMC was designed to combine effective techniques from various research efforts within the verification community and apply them to software written in C and C++, the predominant programming languages in industry.

The ultimate goal of this work is to check systems code in general, but the initial focus is on networking code. Such code naturally follows an "event-driven" execution model which makes it a good fit for model checkers. The correctness of networking protocol implementations is especially important, since they are not only at the core of many services, but also the first target of external security attacks. Unfortunately, network protocols are difficult to design, implement and test because they involve complex interactions among multiple machines across a network and deal with various network failures such as packet losses or link failures, which are difficult to control in a test environment. Model checkers excel at checking such interactions.

CMC works well on real code, as demonstrated by the results of applying it to three implementations of the AODV networking protocol [7]. The first implementation, *mad-hoc* [21], was released two years ago and has since been under active development. The second implementation, *Kernel AODV* [20], derives from the *mad-hoc* implementation and was released less than a year ago. The third implementation, *AODV-UU* [13] was released a year ago. The AODV specification is also in active development: the first version came out in 1997 and has subsequently undergone ten revisions. While it is difficult to measure quality absolutely, one measure is that there is a formal group devoted to testing AODV implementations that has used their testbed to check the *mad-hoc* and the *AODV-UU* implementations [12].

CMC found 34 unique errors in total (as of the date of this publication), a rate of roughly one bug per 328 lines of code. Several bugs were non-trivial ones, difficult to find by any other method. In an ironic twist, model checking the *implementation* found a bug in the *specification* of AODV itself (this last error was confirmed by the authors of the AODV specification [26]). Many protocol implementations are similar to that of AODV, and CMC has enhancements that will broaden its applicability to other concurrent systems. Hence, there is good reason to believe that CMC will be useful for many systems that are difficult to debug by any other means.

2 Model Checking Overview

Fundamentally, explicit state model checking is a systematic search for error states in a state graph, which represents the behavior of some system. It is usually best to generate the graph *on the fly* so that the search can find and report errors even if the state graph is too large to search completely. This is especially important since the state graphs of systems with errors are often much larger than those of correct systems. As with most search algorithms, newly discovered states are stored in a queue. According to some policy (e.g., depth-first, breadth-first, or best-first), states are removed from this queue and the successors generated are expanded and themselves enqueued (there are usually multiple successors because of nondeterminism in the system). States that have already been searched are stored in a hash table so that their successors are not expanded more than once.

Model checking is sometimes used to prove that a system satisfies a specified property. However, it is usually more practical to use it as a bug-finding method. When model checking is applicable, it can be more effective than conventional testing in discovering bugs because of its thoroughness at exploring the state space of the system, including corner cases that might otherwise be overlooked. Model checking can be more efficient than random testing because the former searches each state at most once.

Given some code to model check, it is necessary to model the environment (e.g., the relevant aspects of the network and operating system calls). The environment model is necessary to avoid false error reports resulting from illegal inputs or state changes that would never occur in actual system execution. Many parts of the environment model would be necessary even for unit testing.

Even after the system to be checked is put into a model checker, one of the most apparent problems with model checking is that a relatively small system description can result in a huge state graph. This is called the *state explosion problem*. It has been addressed in many ways, including various methods of suppressing details of the input description (abstraction) and various optimizations to save time and, even more importantly, space. Nevertheless, the state explosion problem remains a serious difficulty in most applications of model checking. (Note that without state pruning, randomized testing will typically fare significantly worse in such situations.)

By addressing the issues described above, this paper presents an approach to pragmatically apply model checking to actual implementation code (in C or C++) to find bugs. To this end, we implemented a tool called CMC that was used to find bugs in network protocol implementations, as described in the following sections.

3 Design of CMC

CMC is a model checker that generates the state space of a given system by directly executing its C or C++ implementation. This section describes the design of CMC, beginning with a description of the tool's infrastructure. The steps required to set up a system for checking are described and illustrated with an example. The actual model checking algorithm then follows. Finally, some techniques to cope with the *state explosion problem* are discussed.

3.1 CMC Infrastructure

CMC models a system as a collection of interacting concurrent agents called *processes*. Each process runs unmodified C or C++ code from the implementation, but the CMC model checker is responsible for scheduling and executing the processes of the system being checked. CMC along with all the processes of the system run as a single operating system process. Unlike an operating system, however, CMC tries to search many *possible* system states that can be reached by alternative scheduling decisions and other nondeterministic events. To search these different possibilities, CMC must be able to save and restore the complete state of the modelled system.

Every process of the system executes in its own heap and stack. At any instant, the state of a process consists of a copy of its global (and static) variables, heap, stack and its context registers. The processes communicate with each other through a shared memory that is accessible in the context of all processes. The state of the system is defined as the union of the states of all processes along with the contents of the shared memory.

Once scheduled, a process is allowed to execute a deterministic and non-blocking set of instructions defined as a *transition*. A transition is an atomic step the system can take and determines the degree of interleaving among processes. The protocols to which CMC has been applied follow an *event-driven* execution model, where a set of event handler routines process incoming events such as packet arrivals and timeouts. In an event-driven protocol, each event handler can be mapped to a transition in CMC. Moreover, as each event handler preserves the state of the stack and registers, only the global variables and the heap need to be saved and restored.

Many protocols are written in an event-driven style, so the event-driven model may not be as restrictive as it may seem. However, it is feasible to save and restore full states (including the stack) of modelled processes as well. This feature has been implemented and is currently being evaluated, but was not necessary for the results reported in this paper.

3.2 Creating a CMC Model from the Implementation

Figure 1 shows a skeleton event-driven implementation of a routing protocol, very similar to the AODV protocol that was actually checked (see section 4). This implementation is used as a running example in the following discussion. The `main()` function of the implementation calls the initialization function (line 34), and then enters an event dispatch loop (line 35). Depending on the input event, it calls one of four event handlers defined in lines 1 through 26. Each handler processes an event: a user request for a route to a destination, a request from another node, a response from another node to one of its previous requests, and a timer event requiring the protocol to invalidate its old routes.

This subsection describes the three steps that the user must perform to apply CMC to a protocol. Steps 1 and 2 essentially provide a unit test scaffold, which would be required for most test environments, such as running the implementation in a simulator. Step 3 is the only additional requirement for CMC.

Step 1: Specifying Correctness Properties. Before any system can be tested, the user must specify some correctness properties. Some properties are domain independent; for example, the program should not access illegal memory and should not leak memory. Some domain-specific properties can be specified as assertions at particular points in the implementation (e.g., the example protocol should never return an invalid route in line 4). In many cases, a careful implementer will have placed these assertions in the code long before CMC is applied. Other properties are inherently global, such as the requirement that there are no loops in the routing table. Such properties are specified as Boolean functions (written in C) that access the datastructures of process contexts.

Step 2: Specifying the Environment. Next, the user must build a test environment that adequately represents the behavior of the actual environment in which the protocol is executed. For networking protocols, the environment model fakes an operating system and anything outside of the protocol that is necessary for it to function. (The decision as to which part of the system is to be checked and which is the environment is decided by the user.) The environment model is a collection of substitute API functions and data structures to emulate its state. The environment should be modelled in as little detail as possible – otherwise, many superfluous states will be generated to model environmental behavior irrelevant to checking the protocol.

Many functions can be replaced by simple “stubs.” For example, `gettimeofday()` might return a constant or contain a counter. In the example, the model requires a network for exchanging routing packets. A simple network could be modelled as an unordered queue of bounded length. The model would include its own versions of interface functions, such as the `broadcast_request()` function that sends packets to the network. The environment may also contain several processes. For ex-

```
 1 /* event handlers */
 2 on_user_request (dest_ip) {
 3   if(route_table has a route for dest_ip)
 4     return route for dest_ip
 5   else
 6     broadcast_request(dest_ip)
 7 }
 8
 9 on_recv_request (dest_ip) {
10   if(route_table has a route for dest_ip)
11     send_response (route for dest_ip)
12   else
13      broadcast_request(dest_ip)
14 }
15
16 on_recv_response (route) {
17   install route in route_table
18   if(route needs to be forwarded)
19     send_response (route)
20 }
21
22 on_timeout(){
23   for each route in route_table
24     if(route too old)
25       remove route from route_table
26 }
27
28 init(){
29   route_table = null
30   insert self route for my_ip
31 }
32
33 main(){
34   init()
35   while(1){
36     /* event dispatch loop */
37     depending on event, call one of
38        on_user_request(...)
39        on_recv_request(...)
40        on_recv_response(...)
41        on_timeout(...)
42   }
43 }
44
```

Figure 1: A simple routing protocol implementation.

```
void* malloc(size_t n){
  if(CMCChoose(2) == 0)
     return 0; //nondeterministic failure

  // alloc n bytes from heap and return
}
```

Figure 2: Implementation of `malloc()` in CMC. `malloc()` nondeterministically fails to allocate memory.

ample, a process that nondeterministically removes a packet from the network can be used to model a lossy network.

To represent nondeterminism in the environment, CMC provides a `CMCChoose()` function (similar to *VS_toss* in Verisoft[15]). CMCChoose takes an integer argument n and returns an integer in the range $(0 \dots n-1)$. CMCChoose arbitrarily selects one out of the n possibilities in the environment. An example, shown in Figure 2, is the `malloc()` implementation that can either allocate the requested memory from the CMC heap or fail by returning NULL. CMCChoose is used to make one of these two choices. CMC will attempt to try all possible return values for each call to CMCChoose. Since calls to CMCChoose appear in the environment code or in implementations of standard system functions (such as `malloc() and select()`), it is generally not necessary to modify the actual implementation.

Providing an environment model can be time consuming. It is therefore important to reduce the modelling effort required to apply CMC to a previously unchecked protocol. A first obvious step is to engineer the models so that they are as re-usable as possible, reducing the incremental effort of checking a new protocol. This is especially beneficial when related protocols are checked (which is a large part of the reason this paper checks three different implementations of the same protocol). Finding other ways to reduce the cost of environmental modelling is an interesting area for future work.

Step 3: Identifying the Initialization Functions and Event Handlers. Given an event driven system, the user should provide the initialization functions and the event handlers for each process in the system.

The user should also provide a *guard function* for each event handler, which is a Boolean function that determines when that event handler is enabled in a given state. For instance, the guard function for the `on_recv_request()` handler returns true only if a request is pending for this particular process in the network.

3.3 CMC Model Checking Algorithm

Given a model of the system built as described above, CMC explores the state space of the system by executing various traces of interleaving transitions. The pseudocode for the algorithm is shown in Figure 3. The algorithm maintains two data structures: a hash table of states seen during the search, and a queue of states seen but whose successors are yet to be generated. The hash table guarantees that the algorithm explores the subgraph rooted at a state at most once.

3.3.1 Generating the Initial State

CMC computes the initial state as follows. Starting from a copy of the global variables (as initialized by the linker), CMC calls the initialization function for each process. The initial state consists of the states of the processes immediately after their initialization functions have been called, along with the values of the initialized shared memory.

3.3.2 Generating Successor States

To generate the state graph on-the-fly, CMC needs to be able to compute the set of possible immediate successors of a state. Each state in the state space of the system may have several successors because of nondeterminism, which arises from several sources: the choice of which process to execute, the choice of which enabled transition within the process to execute, and nondeterministic values returned by calls to CMCChoose.

From a given state, CMC chooses a process and one of its enabled event handlers to schedule. Next, CMC restores the context of the process by copying the contents of the heap and the global variables of the process from the state. The event handler is then called. This function eventually returns because it is guaranteed to be atomic. At this point, the context of the process state is saved, yielding a new system state. CMC generates all successors of a state by repeating the above process for all nondeterministic choices.

3.3.3 Checking Correctness Properties

During model checking, CMC checks for a range of correctness properties, from simple pointer access

```
void modelCheck(){
   SystemState{
      sharedMem;   // Contains Network
      procState[N];// N processes
   } initial, current, successor;

   Queue StateQ;
   Hash VisitedStates;

  // Build initial state.
  forall processes (0 <= pid < N){
    call pid's initFn();
    initial.procState[pid] = saveCtxt();
  }
  call sharedMem's initFn();
  initial.sharedMem = getSharedMem();

  StateQ.insert(initial);
  while(current = StateQ.pop()) {
    VisitedStates.add(current);

    //Repeat forall nondeterministic choices
    forall processes (0 <= pid < N)
    forall event handlers e of pid
    forall return values of CMCChoose calls{

      // set proc context and sharedMem
      restoreCtxt(current.procState[pid]);
      setSharedMem(current.sharedMem);

      if(e is not enabled)
         continue;

      e(); // Call event handler

      // Construct next state
      successor.sharedMem = getsharedMem();
      successor.procState[pid] = saveCtxt();
      forall processes with pid' != pid{
        // the state did not change
        successor.procState[pid']
          = current.procState[pid'];
      }
      if(successor in VisitedStates)
        continue;
      if(successor fails assertions)
        generate error
      StateQ.insert(successor);
    }
  }
}
```

Figure 3: Pseudocode for the CMC model checking algorithm.

violation errors to complex protocol bugs.

During the execution of an event handler, CMC runs the implementation code directly, automatically catching errors such as pointer access violations and program assertion failures present in the code. In addition, CMC detects use-after-free bugs by overwriting any freed memory with a random value.

Once a state is generated, CMC checks for violations of user-provided system invariants (such as the absence of global routing loops). Also, CMC detects memory leaks in each generated state. While this can be achieved by a standard mark-and-sweep algorithm to find all reachable memory, such an algorithm is not yet implemented in CMC. Instead, in our case study, CMC detects memory leaks as follows: starting from a copy of the current state, CMC calls various cleanup functions present in the implementation itself. Any heap memory that is left allocated is reported as leaked. This approach, while requiring additional manual effort, can also potentially find bugs in the cleanup code.

In the future, the CMC approach could easily be coupled with other dynamic debugging tools such as Purify[27] or StackGuard[6]. These tools can catch run-time errors such as uses of uninitialized memory, stack overflows, etc. Such tools would be more effective when used with CMC than with ordinary testing, because CMC would achieve greater effective test coverage for a given level of user effort than conventional software testing methods.

3.4 Handling State Space Explosion

One of the most serious problems with model checking in practice is the so-called "state explosion problem." The state space of a system can be very large, or even infinite. Thus, at the outset, it is impossible to explore the *entire* state space with limited resources of time and memory. However, CMC provides various techniques to search the state space efficiently before running out of resources. Though unable to formally prove the correctness of the implementation, CMC is able to catch a wide range of errors, including errors involving intricate interactions among multiple processes.

For model checkers, memory is more critical a resource than time. During model checking, most of the memory is consumed by the hash table containing the states visited and the queue of states whose successors are yet to be generated.

CMC uses *hash compaction* [28] to reduce the memory requirements in the hash table by several orders of magnitude. For each state, CMC computes a small signature (usually four to eight bytes). Instead of storing the entire state, which can be on the order of kilobytes, its signature is stored in the hash table. Compacting states can lead to conflicts in the hash table where two different states compute to the same signature. However, for state spaces on the order of hundred million states with practical hash table sizes of several hundred megabytes, the probability of missing even a single state due to a signature conflict can be reduced to 0.1% or lower [28].

The states in the queue cannot be compacted because all the information in them is needed to compute successor states. However, the queue has good locality of reference, so much of it can be swapped to disk during model checking. Moreover, successive states in the queue usually have a lot of commonality and can thus be compressed. For instance, every transition in CMC changes at most one process state; therefore, it is sufficient to store only this difference when generating a successor state.

Standardizing Data Structures: CMC, by default, interprets states as streams of bits. However, two equivalent data structures in memory might have different representations. For example, if two states differ only in the order in which objects were allocated on the heap, they should be considered effectively the same. CMC can automatically transform states by deterministically traversing pointer data structures, arranging objects in the heap by the order they are visited. The signature for the transformed state can then be saved in the state table. This process could be performed simultaneously with the mark-and-sweep algorithm used to detect memory leaks. A mostly automatic tool for this traversal is under development using the MC framework [11]. For the case study discussed in Section 5, the traversal code was written manually.

There may be additional equivalences between states that depend on the particular use of data structures in a program. For example, when an implementation uses a linked list to store an *unordered* collection of objects, the behavior of the implementation is independent of the order of objects in the list. In this case, the user can provide a function to sort the list before the automatic standardization transformations are applied.

Finally, some of the most effective reductions in the state space are achieved through methods that risk missing some errors for the benefit of catching the remaining ones more efficiently.

Down-scaling: One obvious approach is to reduce the scale of the system being described [10]. In figure 1, for instance, the model might restrict the number of routing nodes in the network to, say, three or four. Hard-to-find bugs usually involve complex interactions among a *small* number of processes, and are therefore preserved even after down-scaling. Of course, this may miss bugs that only occur for larger instances of the system.

Abstraction of States: In addition to standardizing distinct but equivalent states, it is also possible to eliminate information that the user judges to be unimportant for the properties checked. This abstraction process is done by ignoring certain memory locations when computing the hash signature of the state. By abstracting states, it is possible to miss errors. However, as the abstraction is done during the hash computation, and not on the actual (concrete) state, this does *not* produce any false positives.

Heuristics: When exhaustive checking of the entire state space is infeasible and all else fails, CMC can act as an automated testing framework whereby a large number of scenarios can be checked intelligently. The mere fact that CMC is able to cache states already prevents redundant simulations. The goal, however, is to exercise as many interesting scenarios as possible before memory is exhausted.

To that end, we have done some preliminary work in using heuristics to prioritize the state space search. The first class of heuristics involves dropping states altogether if they are deemed uninteresting. The second class of heuristics involves exploring more interesting states first using best-first search. CMC contains a module to monitor state variables to keep a history of which state bits have changed during checking. The basic idea is that if the number of bit positions that have changed since the initial state suddenly increases or if variables take on less frequented values, the state is considered more interesting and explored earlier. This heuristic tends to bias the search toward cases where outliers occur or where states seem to diverge from the norm. This idea was adapted from DIDUCE [17], a tool that flags such divergent cases and reports them to the user during program testing.

Preliminary results indicate that all the errors discovered with the use of the heuristics could be discovered with simple depth-first search. But, the use of heuristics often accelerated the discovery of errors and produced shorter examples of executions leading to a given error. However, much more experimentation with various heuristics is needed on a wider range of protocols to arrive at reliable conclusions.

The next three sections describe the application and results of using CMC to check three AODV protocol implementations.

4 Description of the AODV Protocol

AODV (Ad-hoc On-demand Distance Vector)[7] is a loop-free routing protocol for ad-hoc networks. It is designed to be self-starting in an environment of mobile nodes, withstanding a variety of network behaviors such as node mobility, link failures and packet losses. This section describes the AODV protocol in brief; the reader is referred to [7] for complete details of the protocol.

At each node, AODV maintains a routing table. The routing table entry for a destination contains three essential fields: a next hop node, a sequence number and a hop count. All packets destined to the destination are sent to the next hop node. The sequence number acts as a form of time-stamping, and is a measure of the freshness of a route. The hop count represents the current distance to the destination node.

Suppose we have two nodes a and b such that b is the next hop of a to some destination d. Also, suppose the sequence number and hop count of the routes to d at a and b are $(seq_a, hcnt_a)$ and $(seq_b, hcnt_b)$ respectively. Then the AODV protocol maintains the following property at all times:

$$(seq_a < seq_b) \vee (seq_a = seq_b \wedge hcnt_a > hcnt_b)$$

In other words, b either has a newer route to d than a, or b has a shorter route that is equally recent. Under this partial order constraint, the protocol is guaranteed to be free of routing loops [2].

In AODV, nodes discover routes in request-response cycles. A node requests a route to a destination by broadcasting an *RREQ* message to all its neighbors. When a node receives an RREQ message but does not have a route to the requested destination, it in turn broadcasts the RREQ message. Also, it remembers a *reverse-route* to the requesting node which can be used to forward subsequent responses to this RREQ. This process repeats until the RREQ reaches a node that has a valid route to the destination. This node (which can be the destination itself) responds with an *RREP* message. This RREP is unicast along the reverse-routes of the intermediate nodes until it reaches the original requesting node. Thus, at the end of this request-response cycle a *bidirectional* route is established between the requesting node and the destination. When a node loses connectivity to its next hop, the node invalidates its route by sending an *RERR* to all nodes that potentially received its *RREP*.

On receipt of the three AODV messages: RREQ, RREP and RERR, the nodes update the next hop, sequence number and the hop counts of their routes in such a way as to satisfy the partial order constraint mentioned above [7].

5 The AODV model

This section describes the AODV model for three implementations of the AODV protocol: *mad-hoc* (Version 1.0) [21], *Kernel AODV* (Version 1.5) [20], and *AODV-UU* (Version 0.5) [13]. The mad-hoc implementation runs as a user space daemon and contains approximately 5500 lines of code. The Kernel AODV implementation is built by NIST and is based on the mad-hoc implementation. It contains 7500 lines of code and runs as a loadable kernel module in Linux and ARM based PDAs. The AODV-UU implementation runs as a user space daemon on Linux and has been ported to the ns-2 [22] simulator. It contains roughly 7700 lines of code.

The AODV model was reused with minor modifications for all three implementations. The model is built as follows:

Correctness Properties: Table 1 lists the correctness properties checked by the AODV model. Apart from the generic assertions checked by CMC, the model contains a global invariant that checks for routing loops. The model also performs sanity checks on the routing table entries and the network messages, such as range violations of the fields.

The Environment: The environment of the model consists of a network modelled as a bounded-length, unordered message queue. The model simulates a message loss by nondeterministically dequeuing a message. The message queue is shared by all of the nodes and thus models a completely connected topology.

The implementations use a wrapper function to send network packets. The model provides an alternate definition to the wrapper function to copy packets to the network model. Additionally, for the Kernel AODV implementation, the model provides implementations for twenty-two kernel functions (such as

Types of Checks	Examples
Generic Assertions	Segmentation violations, memory leaks, dangling pointers.
Routing Loop Invariant	The routing tables of all nodes do not form a routing loop.
Assertions on Routing Table Entries	At most one routing table entry per destination. No route to self in the AODV-UU implementation. The hop count of the route to self is 0, if present. The hop count is either infinity or less than the number of nodes in the network.
Assertions on Message Fields	All reserved fields are set to 0. The hop count in the packet can not be infinity.

Table 1: Properties checked in AODV.

Enabling Condition	Event
Invalid or no route to destination	Initiation of route request
Pending message in the network	Receipt of AODV message
Pending message in the network	Message loss
Valid route in the routing table	Timeout of a route
Always enabled	Detection of link failure
Always enabled	Node reboot

Table 2: The set of event handlers used in AODV model checking.

kmalloc and printk) and a user space version of the socket buffer library.

Initialization Functions and the Event Handlers: All three implementations have an event dispatch loop that calls various event handlers. The initialization functions of the model are obtained by executing the code before the event dispatch loop. The model maps every event handler called from the dispatch loop to a transition. The model simulates a node reboot by calling the initialization function which implicitly resets the contents of the routing table. The list of transitions and their respective enabling conditions is shown in Table 2.

Table 3 shows the lines of code from the three implementations executed within our framework against the lines of code for the model itself. The correctness specifications are mostly shared by the three implementations. AODV-UU uses a different representation of the routing table and thus required additional correctness specifications. The network model of the environment is shared by all implementations.

Dealing with State Space Explosion

The state space of the AODV protocol is essentially infinite. The protocol allows an arbitrary number of nodes in a network. Also, each node has two types of unbounded counters, a *sequence number* to measure the freshness of a route and a *broadcast id* that is incremented by a node on each broadcast. To do any effective search in such an infinite state space, it is necessary to bound the search. In our experiments, we downscaled the AODV model to run with 2 to 4 processes. The model discarded any state in which the sequence numbers or the broadcast ids exceeded a predefined limit. Also, the size of the message queue in the network was bounded to sizes of 1 to 3. These processes may cause CMC to miss errors. However, even after applying such bounds, the remaining state space contained enough interesting behavior to uncover numerous bugs (Section 6).

Time values stored in the state are another source of state space explosion. For instance, every route response (RREP) contains a lifetime field that determines the freshness of the route. On receipt of this packet, a node adds the lifetime to the current clock value to determine the time at which the route becomes stale. This absolute value is stored in the routing table and can thus increase the state space size. The AODV model gets around this problem by modelling route timeouts as nondeterministic events and setting all time variables to predefined constants. Also, the environment of the model contains a definition of the `gettimeofday()` function that always returns a constant value. The handling of time in the model can miss timing related errors

Protocol	Checked Code	Correctness Specification	Environment			State Canonicalization
			network	stubs	*skbuff*	
mad-hoc	3336	301	400	100	-	165
Kernel AODV	4508	301	400	266	1210	179
AODV-UU	5286	332	400	128	-	185

Table 3: Lines of implementation code vs. CMC modelling code.

and can potentially lead to false positives when an error reported can be caused by a sequence of timeouts that is impossible in the real protocol.

Also, the AODV model contains hand-written code to traverse the routing table (implemented as a linked list in the mad-hoc and Kernel AODV implementations, and as a hash table in the AODV-UU implementation). This traversal code created a canonicalized representation of the routing table, which along with the global variables formed the state of an AODV node of the model. The amount of lines required for this traversal code is shown in the last column of Table 3.

6 Results

Table 4 summarizes the set of bugs found using CMC in the three AODV implementations. The bugs range from simple memory errors to protocol invariant violations. We found a total of 40 bugs of which 34 were unique. The Kernel AODV implementation has 5 bugs (shown in parenthesis in the table) that are instances of the same bug in mad-hoc. Also, the AODV specification bug causes a routing loop in all three implementations.

Currently, CMC stops after finding the first bug in the model. It prints the failed assertion and a trace of events starting from the initial state to the error state. After a bug is fixed, CMC is run again to find bugs iteratively. Most bugs were found within minutes of model checking time; the longest took roughly 40 minutes.

We describe the bugs below at a high level to give a feel for the breadth of coverage and focus on four of the more interesting bugs to give a feel for its depth.

Memory errors. The first three error classes illustrate the mishandling of dynamically allocated memory: not checking for allocation failure (12 errors), not freeing allocated memory (8 errors), and using memory after freeing it (2 errors).

All implementations checked that the pointer returned by `malloc` was not null. However, functions that call `malloc` can also indirectly return null pointers when allocations fail. The code only erratically checked such cases. Since CMC directly executes the implementation, such errors were manifested as segmentation faults.

Most of the memory-related bugs were straightforward. However, there were several interesting errors where the code would correctly check for allocation failure, but its recovery code was broken. Figure 4 gives a representative error. Here, the code attempts to allocate `rerrhdr_msg.dst_cnt` temporary message buffers. It correctly checks for `malloc` failure and breaks out of the loop. However, the code after the loop assumes that `rerrhdr_msg.dst_cnt` list entries were indeed allocated. This assumption leads to two bugs. The first (intraprocedural) error

```
/*aodv_deamon.c:aodv_recv_message:*/
...
for(rerri=0; rerri<rerrhdr_msg.dst_cnt;rerri++)
{
   if (!(tp = malloc(sizeof(*tp))))
      break; /* Skip to next packet */
   tp->next = rerrhdr_msg.unr_dst;
   rerrhdr_msg.unr_dst = tp;
   ...
}
// BUG: assumes rerrhdr_msg.dst_cnt buffers
// were allocated!
rec_rerr(info_msg, &rerrhdr_msg);

// Free the list of structs sent to rec_rerr()
for(rerri=0; rerri<rerrhdr_msg.dst_cnt;rerri++)
{
  // BUG: Can be NULL if malloc failed above!
  tp = rerrhdr_msg.unr_dst;
  rerrhdr_msg.unr_dst=rerrhdr_msg.unr_dst->next;
  free(tp);
}
```

Figure 4: Mishandled `malloc` failure: if `malloc` fails, the loop will exit after allocating less than `rerrhdr_msg.dst_cnt` buffers. The two errors are in code that assumes `rerrhdr_msg.dst_cnt` buffers were allocated. Both lead to segmentation faults.

	mad-hoc	Kernel AODV	AODV-UU
Mishandling `malloc` failures	4	6	2
Memory Leaks	5	3	0
Use after free	1	1	0
Invalid Routing Table Entry	0	0	1
Unexpected Message	2	0	0
Generating Invalid Packets	3	2 (2)	1
Program Assertion Failures	1	1 (1)	1
Routing Loops	2	3 (2)	1 (1)
Total	18	16 (5)	6 (1)

Table 4: Number of bugs of each type in the three implementations of AODV. The figures in parenthesis show the number of bugs that are instances of the same bug in the mad-hoc implementation.

attempts to dequeue `rerrhdr_msg.dst_cnt` buffers off of the `rerrhdr_msg.unr_dst` list in order to free them. Since the list has fewer entries than expected, the code will attempt to use a null pointer and get a segmentation fault. The second (interprocedural) error, in `rec_rerr`, similarly tries to walk over the `rerrhdr_msg.dst_cnt` list entries and seg faults because the list is too short.

Most of the memory leaks were similarly caused by mishandled allocation failures. Commonly, code would attempt to do two memory allocations and, if the first allocation succeeded but the second failed, would return with an error, leaking the first pointer.

Unexpected messages. CMC detected two places where unexpected messages would cause mad-hoc to crash with a segmentation violation. Figure 5(a) shows one of the errors. The error happens because AODV encodes state in its messages. In this error:

1. The current node n receives a Route Request (RREQ) message from node *req* requesting a route to node *dst*.
2. Node n inserts a *reverse route* to *req* in its routing table.
3. Then, n looks up the route to *dst* in its routing table.
4. If the route is not there, n re-broadcasts the RREQ message. The RREQ message contains the IP address of both the destination node *dst* and the requesting node *req*.
5. The response to this request, a Route Response (RREP) message, includes both the route to *dst* and the IP address of *req*.
6. Node n inserts the new route to *dst* in its routing table. It then attempts to relay this route to *req* by looking up the route to *req*. In the normal case, this lookup will return the reverse route inserted in Step 2.

This last step causes the error. The code assumes the normal case and uses the result of the routing table lookup for *req* without checking for null. However, the lookup could fail for two reasons. First, if the machine has rebooted, the implementation will start with an empty routing table. If an old RREP message arrives after the reboot, the lookup of *req* will return a null pointer. Second, an attacker could send a bogus RREP with a node address that does not exist, crashing the router.

Invalid messages. There were 4 cases of invalid packets being created, 2 cases of using uninitialized variables (these could not be detected by gcc -Wall), and 2 cases where invalid routes were used to send routing updates, violating the AODV specification (Figure 5(b) gives a representative example). CMC also detected 2 instances of integer overflow which resulted in program assertion failures. The implementations use an 8 bit integer to store the hop counts and use 255 to represent a hopcount of infinity. In these error cases, an infinite hopcount was erroneously incremented to 0.

Routing loops. CMC found three routing loops. Two of these bugs are caused by implementation errors. The third routing loop is due to an error in the AODV protocol specification.

The first routing loop is caused when the implementation fails to increment a sequence number while processing specific RERR messages.

Another loop is caused when the implementation performs a sequence number comparison before a subsequent increment, while the AODV specification requires the comparison to be done after the increment.

```
/* madhoc:rrep.c:rec_rrep */
...
/* If I'm not the destination of the RREP
   I forward it */
if(my_rrep->src_ip != my_info->ip_pkt_my_ip) {
  ...
  // Get the entry to the source from RT.
  rt_src = getentry(my_rrep->src_ip);

  // BUG: rt_src may not exist!
  if (add_precursor(rt_src, rt->nxt_hop) == -1)
  ...
  // Send gratuitous RREP to destination
  // BUG: rt_src can be invalid
  // (i.e rt_src->hop_cnt == 255 )
  // must check after getentry.
  my_rrep.hop_cnt = rt_src->hop_cnt;
  if (send_datagram(my_info, &my_rrep,
                    sizeof(my_rrep)) == -1)
    ...
```

Figure 5: Two bugs: an unexpected message and an invalid route response. (a) An unexpected route-response (RREP) message causes `getentry` to return null, crashing the machine. (b) If a route returned by `getentry` has been invalidated the hop-count will be 255. However, the code does not check for this and sends the message.

The specification bug. This bug involved the handling of RERR ("route error") messages. When a node receives an RERR from its next hop, it sets the sequence number of its route to the sequence number in an RERR message. Under normal conditions this is the right thing to do. However, when the underlying link layer can reorder messages, the RERR message might have an outdated sequence number resulting in the node setting its sequence number to an older version. This can ultimately result in a routing loop. This bug was mentioned to the authors of the protocol with a suggested fix. Both the bug and the fix were accepted by the protocol authors[26]. Figure 6 gives both the error and the fix.

The specification bug was found by running 4 AODV nodes using a depth-first search of the state space. CMC came up with an error trace of length 93. Using best-first search, it was possible to find traces as short as 27. Performing a breadth-first search of the state space would give the shortest trace. However, breadth-first search on AODV ran out of resources without finding the bug. A carefully hand-crafted simulation of the bug required at least 20 transitions. Such a complex error would be very difficult to catch using conventional means of testing.

```
/* madhoc:rerr.c:rec_rerr */
...
// Get pointer to route table for destination IP
tmp_rtentry = getentry(tmp_unr_dst->unr_dst_ip);
if(tmp_rtentry != NULL && ...) {
  // BUG: uses sequence number from incoming
  // message in tmp_unr_dst without validation.
  // Should check:
  //   if(tmp_rtentry->dst_seq >=
  //          tmp_unr_dst->unr_dst_seq)
  //      return -1;
  tmp_rtentry->dst_seq = tmp_unr_dst->unr_dst_seq;
```

Figure 6: The specification bug: the sequence number from an incoming message is used without validation, causing "time" to go backwards when messages are reordered. Fortunately, while the error was not obvious (surviving 6 rounds of specification revisions) the fix is trivial.

7 Related Work

This paper proposes an initial approach to systematically and efficiently verify a large class of C and C++ software without having to create abstract models in a different language. The following compares our work using CMC to other efforts in traditional model checking, software model checking, and static analysis.

Traditional Model Checking: The basic idea of using state graph search to verify network and communication protocols is quite old, dating back to at least 1978 [16, 30]. In recent decades, model checking has made significant progress in tackling the verification of complex, concurrent systems. Tools such as SMV[19], SPIN[18], and Murphi[10] have been used to verify hardware and software protocols by exhaustively searching the state space. By caching states and employing sound state reduction techniques, these tools can detect non-trivial bugs.

The drawback of traditional model checkers is that the system to be verified must be modeled in a particular description language, requiring a significant amount of manual effort that can easily be error prone. CMC was specifically designed with the goal of reducing the amount of work that is required to go from software development to systematic verification.

Software Model Checking: Some recent formal verification tools have already used the idea of executing and checking systems at the implementation level. Verisoft [15], for instance, systematically executes and verifies actual code and has been used to successfully check communication protocols written in C.

However, Verisoft does not store states and can thus potentially explore a state more than once. This problem is alleviated to some degree by partial order reduction, a sound state space reduction technique implemented in Verisoft that eliminates the exploration of redundant interleavings of transitions created by commutative operations. Nevertheless, this technique requires hints to be provided by the user and/or some static analysis of the code to determine dependencies between transitions; indeed, when the set of possible transitions in a system have a high degree of interdependence, as is the case with the handlers in the protocol code we verified, partial order methods become less effective. Finally, interesting systems almost always have state spaces with cycles and in such cases Verisoft is limited to checking only up to a fixed depth.

Java PathFinder [3] uses model checking to verify concurrent Java programs for deadlock and assertion failures. It relies on a specialized virtual machine that is tailored to automatically extract the current state of a Java program. Much like CMC, Java PathFinder compresses and stores states in a table to prevent redundant searches and relies on various abstraction techniques to curb the state space explosion problem. The infrastructure on which JPF relies, however, can not be applied to software written in C or C++, which are still the predominant languages used in system software development.

SLAM [1] is a tool that converts C code into abstracted skeletons that contain only Boolean types. SLAM then model checks the abstracted program to see if an error state is reachable. One difficulty in using a tool like SLAM is giving a specification of the correct behavior of the system. Because SLAM is a static tool, writing a specification that "no routing loops are possible" would be difficult because it depends on the interleaved event behavior of multiple nodes. Furthermore, SLAM does not deal with concurrent environments that contain multiple processes, queues, etc.

Static Analysis: Static analysis has also gained ground in recent years in detecting bugs in software. Tools such as ESC [9], LCLint [14], ESP [8], and the MC Checker [11] have been used to check source code for errors that can be statically detected with minimal manual effort. While static techniques are good for finding a specific set of errors, the CMC approach can find deep conceptual errors in the code such as emergent routing loops that are difficult to find statically. In addition, CMC does not suffer from too many false positives since every scenario checked is a valid execution path.

8 Conclusion and Future Work

This paper has described CMC, a model checker targetting subtle bugs in systems code, and experimental results from using CMC to check three implementations of the AODV routing protocol. The key features of CMC are that it checks implementation code directly and stores states to avoid redundant state explorations. Initial experiences with CMC are very encouraging: CMC is powerful enough to discover non-trivial bugs both in the implementation and the specification of protocols.

We are currently using CMC to verify larger and more complex protocols. For wider use, it is essential to automate the process of converting an implementation of a system to its CMC model as much as possible. While the results reported here did require considerable manual effort, future improvements to CMC should significantly reduce this.

We are also exploring the use of heuristics to efficiently search the state space. Our initial findings suggest that simple heuristics provide huge improvements in the state space search. For instance, we have implemented a monitor that detects counters and other "rogue" variables (such as uninitialized variables and statistics variables). This monitor abstracts away such variables from the system state, automatically pruning an otherwise infinite state space. Another interesting avenue for research is to use simple facts discovered through static analysis of the code to direct the search to interesting parts of the state space.

9 Acknowledgments

We thank Satyaki Das for thoughtful discussions on the paper. Also, we thank Miguel Castro and various anonymous reviewers for providing valuable comments and suggesting improvements on previous versions of this paper.

References

[1] Thomas Ball, Rupak Majumdar, Todd Millstein, and Sriram K. Rajamani. Automatic predicate abstraction of C programs. In *Proceedings of the SIGPLAN '01 Conference on Programming Language Design and Implementation*, 2001.

[2] K. Bhargavan, D. Obradovic, and C. Gunter. Formal verification of standards for distance vector routing protocols, 1999.

[3] G. Brat, K. Havelund, S. Park, and W. Visser. Model checking programs. In *IEEE International Conference on Automated Software Engineering (ASE)*, 2000.

[4] E.M. Clarke, O. Grumberg, and D. Peled. *Model Checking*. MIT Press, 1999.

[5] J.C. Corbett, M.B. Dwyer, J. Hatcliff, S. Laubach, C.S. Pasareanu, Robby, and H. Zheng. Bandera: Extracting finite-state models from java source code. In *ICSE 2000*, 2000.

[6] Crispan Cowan, Calton Pu, Dave Maier, Jonathan Walpole, Peat Bakke, Steve Beattie, Aaron Grier, Perry Wagle, Qian Zhang, and Heather Hinton. StackGuard: Automatic adaptive detection and prevention of buffer-overflow attacks. In *Proc. 7th USENIX Security Conference*, pages 63–78, San Antonio, Texas, jan 1998.

[7] C.Perkins, E. Royer, and S. Das. Ad Hoc On Demand Distance Vector (AODV) Routing. IETF Draft, http://www.ietf.org/internet-drafts/draft-ietf-manet-aodv-10.txt, January 2002.

[8] Manuvir Das, Sorin Lerner, and Mark Seigle. Esp: Path-sensitive program verification in polynomial time. In *Conference on Programming Language Design and Implementation*, 2002.

[9] David L. Detlefs, K. Rustan M. Leino, Greg Nelson, and James B. Saxe. Extended static checking, 1998.

[10] David L. Dill, Andreas J. Drexler, Alan J. Hu, and C. Han Yang. Protocol verification as a hardware design aid. In *IEEE International Conference on Computer Design: VLSI in Computers and Processors*, pages 522–525, 1992.

[11] D.R. Engler, B. Chelf, A. Chou, and S. Hallem. Checking system rules using system-specific, programmer-written compiler extensions. In *Proceedings of the Fourth Symposium on Operating Systems Design and Implementation*, October 2000.

[12] Erik Nordstrom *et al.* Ad hoc protocol evaluation testbed. http://apetestbed.sourceforge.net/.

[13] Erik Nordstrom *et al.* AODV-UU Implementation. http://user.it.uu.se/ henrikl/aodv/.

[14] David Evans, John Guttag, James Horning, and Yang Meng Tan. LCLint: A tool for using specifications to check code. In *Proceedings of the ACM SIGSOFT '94 Symposium on the Foundations of Software Engineering*, pages 87–96, 1994.

[15] P. Godefroid. Model Checking for Programming Languages using VeriSoft. In *Proceedings of the 24th ACM Symposium on Principles of Programming Languages*, 1997.

[16] J. Hajek. Automatically verified data transfer protocols. In *Proceedings of the 4th ICCC*, pages 749–756, 1978.

[17] Sudheendra Hangal and Monica S. Lam. Tracking down software bugs using automatic anomaly detection. In *Proceedings of the International Conference on Software Engineering*, May 2002.

[18] Gerard J. Holzmann. The model checker SPIN. *Software Engineering*, 23(5):279–295, 1997.

[19] McMillan K. *Symbolic Model Checking*. Kluwer Academic Publishers, 1993.

[20] Luke Klein-Berndt and et.al. Kernel AODV Implementation. http://w3.antd.nist.gov/wctg/aodv_kernel/.

[21] F. Lilieblad and et.al. Mad-hoc AODV Implementation. http://mad-hoc.flyinglinux.net/.

[22] S. McCanne and S. Floyd. UCB/LBNL/VINT network simulator - ns (version 2), April 1999. http://www.isi.edu/nsnam/ns/.

[23] K.L. McMillan and J. Schwalbe. Formal verification of the gigamax cache consistency protocol. In *Proceedings of the International Symposium on Shared Memory Multiprocessing*, pages 242–51. Tokyo, Japan Inf. Process. Soc., 1991.

[24] G. Nelson. *Techniques for program verification*. Available as Xerox PARC Research Report CSL-81-10, June, 1981, Stanford University, 1981.

[25] D. Park, U. Stern, J. Skakkebaek, and D. L. Dill. Java model checking. In *IEEE International Conference on Automated Software Engineering (ASE)*, 2000.

[26] Charles E. Perkins, Elizabeth M. Royer, and Samir R. Das. Private Email Communication.

[27] Rational Software. Purify: Advanced runtime error checking for C/C++ developers. http://www.rational.com/products/purify_unix/.

[28] U. Stern and D. L. Dill. A New Scheme for Memory-Efficient Probabilistic Verification. In *IFIP TC6/WG6.1 Joint International Conference on Formal Description Techniques for Distributed Systems and Communication Protocols, and Protocol Specification, Testing, and Verification*, 1996.

[29] U. Stern and D.L. Dill. Automatic verification of the SCI cache coherence protocol. In *Correct Hardware Design and Verification Methods: IFIP WG10.5 Advanced Research Working Conference Proceedings*, 1995.

[30] C.H. West. General technique for communications protocol validation. *IBM Journal of Research and Development*, 22(4), 1978.