Life, Death, and the Critical Transition: Finding Liveness Bugs in Systems Code

Charles Killian, James W. Anderson, Ranjit Jhala, and Amin Vahdat
University of California, San Diego
{*ckillian, jwanderson, jhala, vahdat*}*@cs.ucsd.edu*

Abstract

Modern software model checkers find *safety* violations: breaches where the system enters some bad state. However, we argue that checking *liveness* properties offers both a richer and more natural way to search for errors, particularly in complex concurrent and distributed systems. Liveness properties specify desirable system behaviors which must be satisfied *eventually*, but are not *always* satisfied, perhaps as a result of failure or during system initialization.

Existing software model checkers cannot verify liveness because doing so requires finding an infinite execution that does not satisfy a liveness property. We present heuristics to find a large class of liveness violations and the *critical transition* of the execution. The critical transition is the step in an execution that moves the system from a state that does not currently satisfy some liveness property—but where recovery is possible in the future—to a dead state that can never achieve the liveness property. Our software model checker, MACEMC, isolates complex liveness errors in our implementations of PASTRY, CHORD, a reliable transport protocol, and an overlay tree.

1 Introduction

Hard-to-find, non-reproducible bugs have long been the bane of systems programmers. Such errors prove especially challenging in unreliable distributed environments with failures and asynchronous communication. For example, we have run our MACE implementation of the PASTRY [28] overlay on the Internet and emulated environments for three years with occasional unexplained erroneous behavior: some nodes are unable to rejoin the overlay after restarting. Unable to recreate the behavior, we never succeeded in tracking down the cause of the error.

Motivated by this and similarly subtle bugs, we turned to model checking to assist us in building robust distributed systems. Unfortunately, existing model checkers able to run on systems implementations (rather than specifications) can only find *safety* violations—counterexamples of a specified condition that should always be true. Simple examples of safety properties are `assert()` statements and unhandled program exceptions. For our target systems however, specifying global *liveness* properties—conditions that should always *eventually* be true—proved to be more desirable. In the above example, we wished to verify that eventually all PASTRY nodes would form a ring. Somewhat paradoxically, specifying the appropriate safety property requires knowledge of the nature of the bug, whereas specifying the appropriate liveness property only requires knowledge of desirable high-level system properties. It is acceptable for a node to be unable to join a ring temporarily, but in our case, the bug made it impossible for a node to ever join the ring, thus violating liveness.

Existing software model checkers focus on safety properties because verifying liveness poses a far greater challenge: the model checker cannot know *when* the properties should be satisfied. Identifying a liveness violation requires finding an *infinite* execution that will not ever satisfy the liveness property, making it impractical to find such violating infinite executions in real implementations. Thus, we set out to develop practical heuristics that enable software model checkers to determine whether a system satisfies a set of liveness properties.

We present MACEMC, the first software model checker that helps programmers find liveness violations in complex systems implementations. We built our solution upon three key insights:

Life: To find subtle, complicated bugs in distributed systems, we should search for liveness violations in addition to safety violations. Liveness properties free us from only specifying what ought not happen—that is, error conditions and invariants, which may be hopelessly complicated or simply unknown—and instead let us specify what ought to happen.

Death: Instead of searching for general liveness violations, which require finding violating infinite executions, we focus on a large subset: those that enter *dead* states from which liveness can never be achieved regardless of any subsequent actions. We thereby reduce the problem of determining liveness to searching for violations of previously unknown safety properties. We present a novel heuristic to identify dead states and locate executions leading to them by combining exhaustive search with long random executions.

Critical Transition: To understand and fix a liveness error, the developer must painstakingly analyze the tens of thousands of steps of the non-live execution to find where and how the system became dead. We show how to extend our random execution technique to automatically search for the *critical transition*, the step that irrecoverably cuts off all possibility of ever reaching a live state in the future.

To further help the programmer understand the cause of an error, we developed MDB, an interactive debugger providing forward and backward stepping through global events, per-node state inspection, and event graph visualization. In our experience, MDB, together with the critical transition automatically found by MACEMC, reduced the typical human time required to find and fix liveness violations from a few hours to less than 20 minutes.

Using MACEMC and MDB, we found our PASTRY bug: under certain circumstances, a node attempting to rejoin a PASTRY ring using the same identifier was unable to join because its join messages were forwarded to unjoined nodes. This error was both sufficiently obscure and difficult to fix that we decided to check how FREEPASTRY [1], the reference implementation, dealt with this problem. The following log entry in a recent version of the code (1.4.3) suggests that FREEPASTRY likely observed a similar problem: "Dropped JoinRequest on rapid rejoin problem – There was a problem with nodes not being able to quickly rejoin if they used the same NodeId. Didn't find the cause of this bug, but can no longer reproduce."

We have found 52 bugs using MACEMC thus far across a variety of complex systems. While our experience is restricted to MACEMC, we believe our random execution algorithms for finding liveness violations and the critical transition generalize to any state-exploration model checker capable of replaying executions. It should therefore be possible to use this technique with systems prepared for other model checkers by defining liveness properties for those systems. Although our approach to finding liveness violations is necessarily a heuristic—a proof of a liveness violation requires finding an infinite execution that never satisfies liveness—we have not had any false positives among the set of identified violations to date.

2 System Model

Software model checkers find errors by exploring the space of possible executions for systems implementations. We establish the MACEMC system model with our simplified definitions of programs and properties (see [19] for the classical definitions). We then discuss the relationship between liveness and safety properties.

Distributed Systems as Programs We model-check distributed systems by composing every node and a simulated network environment in a single program (cf. §4.1 for the details of preparing unmodified systems for model checking). A program *state* is an assignment of values to variables. A *transition* maps an input state to an output state. A *program* comprises a set of variables, a set of initial states, and a set of transitions. A *program execution* is an infinite sequence of states, beginning in an initial program state, with every subsequent state resulting from the application of some transition (an atomic set of machine instructions) to its predecessor. Intuitively, the set of variables corresponds to those of every node together with the distributed environment, such as the messages in the network. Thus, a state encodes a snapshot of the entire distributed system at a given instant in time.

Conceptually, each node maintains a set of pending events. At each step in the execution, the model checker selects one of the nodes and an event pending at that node. The model checker then runs the appropriate event handler to transition the system to a new state. The handler may send messages that get added to event queues of destination nodes or schedule timers to add more events to its pending set. Upon completing an event handler, control returns to the model checker and we repeat the process. Each program execution corresponds to a scheduling of interleaved events and a sequence of transitions.

Properties A *state predicate* is a logical predicate over the program variables. Each state predicate evaluates to `TRUE` or `FALSE` in any given state. We say that a state *satisfies* (resp., *violates*) a state predicate if the predicate evaluates to `TRUE` (resp., `FALSE`) in the state.

Safety Property: a statement of the form *always* p where p is a *safety (state) predicate*. An execution *satisfies* a safety property if *every* state in the execution satisfies p. Conversely, an execution *violates* a safety property if *some* state in the execution violates p.

Liveness Property: a statement of the form *always eventually* p where p is a *liveness (state) predicate*. We define program states to be in exactly one of three categories with respect to a liveness property: *live*, *dead*, or *transient*. A live state satisfies p. A transient state does not satisfy p, but some execution through the state leads to a live state. A dead state does not satisfy p, and no execution through the state leads to a live state. An execution *satisfies* a liveness property if every suffix of the execution contains a live state. In other words, an execution satisfies the liveness property if the system enters a live state infinitely often during the execution. Conversely, an execution *violates* a liveness property if the execution has a suffix without any live states.

System	Name	Property	
Pastry	AllNodes	*Eventually*	$\forall n \in$ **nodes** : $n.(successor)^* \equiv$ **nodes** Test that all nodes are reached by following successor pointers from each node.
	SizeMatch	*Always*	$\forall n \in$ **nodes** : $n.myright.size() + n.myleft.size() = n.myleafset.size()$ Test the sanity of the leafset size compared to left and right set sizes.
Chord	AllNodes	*Eventually*	$\forall n \in$ **nodes** : $n.(successor)^* \equiv$ **nodes** Test that all nodes are reached by following successor pointers from each node.
	SuccPred	*Always*	$\forall n \in$ **nodes** : $\{n.predecessor = n.me \iff n.successor = n.me\}$ Test that a node's predecessor is itself if and only if its successor is itself.
RandTree	OneRoot	*Eventually*	**for** *exactly* $1\ n \in$ **nodes** : $n.isRoot$ Test that exactly one node believes itself to be the root node.
	Timers	*Always*	$\forall n \in$ **nodes** : $\{(n.state = init) \| (n.recovery.nextScheduled() \neq 0)\}$ Test that either the node state is $init$, or the recovery timer is scheduled.
MaceTransport	AllAcked	*Eventually*	$\forall n \in$ **nodes** : $n.inflightSize() = 0$ Test that no messages are in-flight (i.e., not acknowledged).
			No corresponding safety property identified.

Table 1: Example predicates from systems tested using MACEMC. *Eventually* refers here to *Always Eventually* corresponding to Liveness properties, and *Always* corresponds to Safety properties. The syntax allows a regular expression expansion '*', used in the AllNodes property.

It is important to stress that liveness properties, unlike safety properties, apply over entire program executions rather than individual states. Classically, states cannot be called live (only executions)—we use the term live state for clarity. The intuition behind the definition of liveness properties is that any violation of a liveness state predicate should only be temporary: in any live execution, regardless of some violating states, there must be a future state in the execution satisfying the liveness predicate.

Table 1 shows example predicates from systems we have tested in MACEMC. We use the same liveness predicate for PASTRY and CHORD, as both form rings with successor pointers.

Liveness/Safety Duality We divide executions violating liveness into two categories: Transient-state and Dead-state. *Transient-state (TS) liveness violations* correspond to executions with a suffix containing only transient states. For example, consider a system comprising two servers and a randomized job scheduling process. The liveness property is that eventually, the cumulative load should be balanced between the servers. In one TS liveness violation, the job scheduling process repeatedly prefers one server over the other. Along a resulting infinite execution, the cumulative load is never balanced. However, at every point along this execution, it is possible for the system to recover, e.g., the scheduler could have balanced the load by giving enough jobs to the underutilized server. Thus, all violating states are transient and the system never enters a dead state.

Dead-state (DS) liveness violations correspond to an execution with any dead state (by definition all states following a dead state must also be dead because recovery is impossible). Here, the violating execution takes a *critical transition* from the last transient (or live) state to the first dead state. For example, when checking an overlay tree (cf. §6), we found a violating execution of the "OneRoot" liveness state predicate in Table 1, in which two trees formed independently and never merged. The critical transition incorrectly left the $recovery$ timer of a node A unscheduled in the presence of disjoint trees. Because only A had knowledge of members in the other tree, the protocol had no means to recover.

Our work focuses on finding DS liveness violations. We could have found these violations by using safety properties specifying that the system never enters the corresponding dead states. Unfortunately, these safety properties are often impossible to identify *a priori*. For instance, consider the liveness property "AllNodes" for CHORD shown in Table 1: eventually, all nodes should be reachable by following successor pointers. We found a violation of this property caused by our failure to maintain the invariant that in a one-node ring, a node's predecessor and successor should be itself. Upon finding this error, we added the corresponding safety property for CHORD. While we now see this as an "obvious" safety property, we argue that exhaustively listing all such safety properties *a priori* is much more difficult than specifying desirable liveness properties.

Moreover, liveness properties can identify errors that in practice are infeasible to find using safety properties. Consider the "AllAcked" property for our implementation of a transport protocol, shown in Table 1. The property is for the test application, which sends a configurable total number of messages to a destination. It states that all sent messages should eventually be acknowledged by the destination (assuming no permanent failures): the transport adds a message to the $inflight$ queue upon sending and removes it when it is acknowledged. The corresponding safety property would have to capture the following: "Always, for each message in the *inflight* queue or retransmission timer queue, either the message is in flight (in the network), or in the destination's receive socket buffer, or the receiver's corresponding *IncomingConnection.next* is less than the message sequence number, or an acknowledgment is in flight from the destination to the sender with a sequence number greater than or equal to the message sequence number, or the same acknowledgment is in the sender's receive socket buffer, or a reset message is in flight between the sender and receiver (in either direction), or ..." Thus, attempting to specify certain conditions with safety properties quickly becomes overwhelming and hopelessly complicated, especially when contrasted with the simplicity and succinctness of the liveness property: "Eventually, for all n in nodes, $n.inflightSize() = 0$," i.e., that eventually there should be no packets in flight.

Thus, we recommend the following iterative process for finding subtle protocol errors in complex concurrent environments. A developer begins by writing desirable high-level liveness properties. As these liveness properties typically define the correct system behavior in steady-state operation, they are relatively easy to specify. Developers can then leverage insight from DS liveness violations to add new safety properties. In Table 1, we show safety properties that became apparent while analyzing the corresponding DS liveness violations. While safety properties are often less intuitive, the errors they catch are typically easier to understand—the bugs usually do not involve complex global state and lie close to the operations that trigger the violations.

3 Model Checking with MACEMC

This section presents our algorithms for finding liveness and safety violations in systems implementations. We find potential liveness violations via a three-step state exploration process. While our techniques do not present proofs for the existence of a liveness violation, we have thus far observed no false positives. In practice, all flagged violations must be human-verified, which is reasonable since they point to bugs which must be fixed. As shown in Figure 1, our process isolates executions leading the system to dead states where recovery to a configuration satisfying the liveness state predicate becomes impossible.

Step 1: Bounded depth-first search (BDFS) We begin by searching from an initial state with a bounded depth-first search. We exhaustively explore all executions up to some fixed depth in a depth-first manner and then repeat with an increased depth bound. Due to state explosion, we can only exhaustively explore up to a relatively shallow depth of transitions (on the order of 25-30); as system initialization typically takes many more transitions (cf. Figure 2), the vast majority of states reached at the periphery of the exhaustive search are not live. We call these states *indeterminate* because at this point we do not yet know whether they are dead or transient.

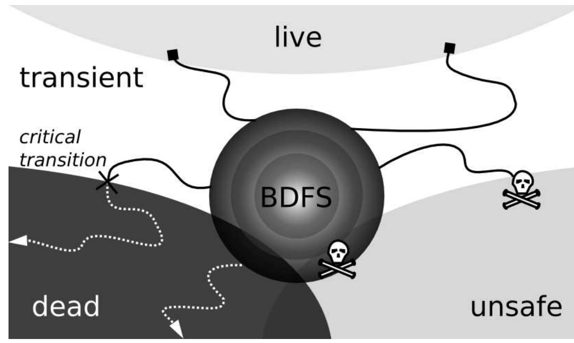

Figure 1: **State Exploration** We perform bounded depth-first search (BDFS) from the initial state (or search prefix): most periphery states are indeterminate, i.e., not live, and thus are either dead or transient. We execute random walks from the periphery states and flag walks not reaching live states as suspected violating executions.

Step 2: Random Walks While the exhaustive search is essential to finding a candidate set of liveness violations, to prune the false positives, we must distinguish the dead from the transient states. To do so, we perform long random walks to give the system sufficient time to enter a live state. If the system still fails to reach a live state by the end of the walk, we flag the execution as a suspected liveness violation. Our random walks typically span tens or hundreds of thousands of transitions to minimize the likelihood of false positives.

Step 3: Isolating the Critical Transition The model checker presents the execution exhibiting a suspected liveness violation to the developer to assist in locating the actual error. The programmer cannot understand the bug simply by examining the first states that are not live, as these are almost always transient states, i.e., there exist executions that would transition these initial indeterminate states to live states. Thus, we developed an algorithm to automatically isolate the *critical transition* that irreversibly moves the system from a transient state to a dead state.

3.1 Finding Violating Executions

We now describe the details of our algorithms. Suppose that MACEMC is given a system, a safety property *always* p_s, and a liveness property *eventually* p_l.

Our algorithm MaceMC_Search (Algorithm 1) systematically explores the space of possible executions. Each execution is characterized by the sequence of choices

Algorithm 1 MaceMC_Search

Input: Depth $increment$

$depth = 0$
repeat
  if $Sequences(depth)$ is empty **then**
    $depth = depth + increment$
  Reset system
  $seq =$ next sequence in $Sequences(depth)$
  MaceMC_Simulator(seq)
until STOPPING CONDITION

made to determine the node-event pair to be executed at each step. We iterate over all the sequences of choices of some fixed length and explore the states visited in the execution resulting from the sequence of choices. Consider the set of all executions bounded to a given depth $depth$. These executions form a tree by branching whenever one execution makes a different choice from another. To determine the order of executions, we simply perform a depth-first traversal over the tree formed by this depth bound. $Sequences(depth)$ returns a sequence of integers indicating which child to follow in the tree during the execution. It starts by returning a sequence of 0's, and each time it is called it increases the sequence, searching all possible sequences. For each sequence, MaceMC_Search initializes the system by resetting the values of all nodes' variables to their initial values and then calls the procedure MaceMC_Simulator to explore the states visited along the execution corresponding to the sequence. After searching all sequences of length $depth$, we repeat with sequences of increasing depth. We cannot search extreme system depths due to the exponential growth in state space. While they have not been necessary to date, optimizations such as multiple random walks or best-first search may enhance coverage over initial system states.

Algorithm 2 MaceMC_Simulator

Input: Sequence seq of integers

for $i = 0$ to d_{max} **do**
  $readyEvents =$ set of pending $\langle node, event\rangle$ pairs
  $eventnum =$ Toss($i, seq, |readyEvents|$)
  $\langle node, event\rangle = readyEvents[eventnum]$
  Simulate $event$ on $node$
  if p_s is violated **then**
    signal SAFETY VIOLATION
  if $i > depth$ **and** p_l is satisfied **then**
    return
signal SUSPECTED LIVENESS VIOLATION

Algorithm 2, MaceMC_Simulator, takes a sequence of integers as input and simulates the resulting execution using the sequence of choices corresponding to the integers. MaceMC_Simulator simulates an execution of up to d_{max} transitions (cf. §4.4 for setting d_{max}). At the i^{th} step, MaceMC_Simulator calls the procedure Toss with i, the sequence, and the number of ready events to determine pending node event pairs to execute, and then executes the handler for the chosen event on the chosen node to obtain the state reached after i transitions. If this state violates the given safety predicate, then MaceMC_Simulator reports the safety violation. If this state is beyond the search depth and satisfies the given liveness predicate, then the execution has not violated the liveness property and the algorithm returns. Only considering liveness for states beyond the search depth is important because otherwise a live state within the periphery would prevent us from finding liveness bugs that enter the dead state beyond the periphery. If the loop terminates after d_{max} steps, then we return the execution as a suspected liveness violation.

Combining Exhaustive Search and Random Walks The procedure Toss ensures that MaceMC_Search and MaceMC_Simulator together have the effect of exhaustively searching all executions of bounded depths and then performing random walks from the periphery of the states reached in the exhaustive search. Toss(i, seq, k) returns the i^{th} element of the sequence seq if i is less than $|seq|$ (the length of the sequence) or some random number between 0 and k otherwise. Thus, for the first $|seq|$ iterations, MaceMC_Simulator selects the $seq[i]^{th}$ element of the set of pending node event pairs, thereby ensuring that we exhaustively search the space of all executions of depth $|seq|$. Upon reaching the end of the supplied sequence, the execution corresponds to a random walk of length $d_{max} - |seq|$ performed from the periphery of the exhaustive search. By ensuring d_{max} is large enough (hundreds of thousands of transitions), we can give the system enough opportunity to reach a live state. If the execution never enters a live state despite this opportunity, we flag the execution as a suspected liveness violation.

3.2 Finding the Critical Transition

If MACEMC reaches the maximum random walk depth d_{max} without entering a live state, we have a suspected liveness violation. The execution meets one of two conditions:

*Condition 1 (*C1*):* The execution is a DS liveness violation, meaning the system will never recover. The execution should be brought to the attention of the programmer to locate and fix the error.

*Condition 2 (*C2*):* The execution does not reach any live states, but might still in the future. The execution should be brought to the attention of the programmer to determine whether to proceed by increasing d_{max} or by inspecting the execution for a bug.

Before discussing how we distinguish between the two cases, consider an execution that does enter a dead state (meets condition C1). The programmer now faces the daunting and time consuming task of wading through tens of thousands of events to isolate the protocol or implementation error that transitioned the system to a dead state. Recall that while the system may enter a transient state early, typically a much later critical transition finally pushes the system into a dead state. After attempting to find liveness errors manually when only the violating execution was available, we set out to develop an algorithm to automatically locate the critical transition. Importantly, this same procedure also heuristically identifies whether an execution meets C1 or C2.

Algorithm 3 FindCriticalTransition

Input: Execution *E* non-live from step d_{init} to d_{max}
Input: Number of Random Walks *k*
Output: (Critical Transition d_{crit}, Condition C1 or C2)
1: {Phase 1: Exponential Search}
2: **if not** Recovers(E, d_{init}, k) **then return** (d_{init},C2)
3: $d_{curr} = d_{init}$
4: **repeat**
5: $d_{prev} = d_{curr}$
6: $d_{curr} = 2 \times d_{curr}$
7: **if** $d_{curr} > d_{max}/2$ **then return** (d_{curr},C2)
8: **until not** Recovers(E, d_{curr}, k)
9: {Phase 2: Binary Search}
10: {d_{prev} is highest known recoverable}
11: {d_{curr} is lowest believed irrecoverable}
12: **loop**
13: **if** $(d_{prev} = d_{curr} - 1)$ **then return** (d_{curr},C1)
14: $d_{mid} = (d_{prev} + d_{curr})/2$
15: **if** Recovers(E, d_{mid}, k) **then** $d_{prev} = d_{mid}$
16: **else** $d_{curr} = d_{mid}$

Algorithm 3 shows our two-phase method for locating the critical transition. It takes as input the execution *E* from the initial random walk, which from step d_{init} onwards never reached a live state even after executing to the maximum depth d_{max}. The function Recovers(E, i, k) performs up to *k* random walks starting from the i^{th} state on the execution *E* to the depth d_{max} and returns `TRUE` if *any* of these walks hit a live state, indicating that the i^{th} state should be marked transient; and `FALSE` otherwise, indicating that the i^{th} state is dead. In the first phase, MACEMC doubles d_{curr} until Recovers indicates that d_{curr} is dead. d_{max} and the resulting d_{curr} place an upper bound on the critical transition, and the known live state d_{prev} serves as a lower bound. In the second phase, MACEMC performs a binary search using Recovers to find the critical transition as the *first* dead state d_{crit} between d_{prev} and d_{curr}. If we perform *k* random walks from each state along the execution, then the above procedure takes $O(k \cdot d_{max} \cdot \log d_{crit})$ time (Note that $d_{crit} \leq d_{max}$).

In addition to the full execution that left the system in a dead state and the critical transition d_{crit}, we also present to the programmer the event sequence that shares the longest common prefix with the DS liveness violation that ended in a live state. In our experience, the combination of knowing the critical transition and comparing it to a similar execution that achieves liveness is invaluable in finding the actual error.

Two interesting corner cases arise in the FindCriticalTransition algorithm. The first case occurs when Phase 1 cannot locate a dead state (indicated by $d_{curr} > d_{max}/2$ in line 7). In this case, we conclude that as the critical transition does not appear early enough, the system was not given enough opportunity to recover during the random walk. Thus, case C2 holds. The developer should raise d_{max} and repeat. If raising d_{max} does not resolve the problem, the developer should consider the possibility that this execution is a TS liveness violation. To help this analysis, MACEMC provides the set of live executions similar to the violating execution, but the developer must isolate the problem. In the second case, we find no live executions even when in the initial state (line 2); either the critical transition is at d_{init} (the initial state), or, more likely, we did not set d_{max} high enough. The programmer can typically determine with ease whether the system condition at d_{init} contains a bug. If not, once again we conclude that case C2 holds and raise d_{max} and repeat Algorithm 1.

4 Implementation Details

This section describes several subtle details in our MACEMC implementation. While we believe the techniques described in Section 3 could be applied to any state-exploration model checker capable of replaying executions, MACEMC operates on systems implemented using the MACE compiler and C++ language extensions [18]. MACE introduces syntax to structure each node as a state machine with atomic handlers corresponding to events such as message reception, timers firing, etc. MACE implementations consist of C++ code in appropriately identified code blocks describing system state variables and event handler methods; and the MACE compiler outputs C++ code ready to run across the Internet by generating classes and methods to handle event dispatch, serialization, timers, callbacks, etc. MACE implementations perform comparably or better than hand-tuned implementations. Leveraging MACE code frees us from the laborious task of modifying source code to isolate the execution of the system, e.g., to control network communication events, time, and other sources of potential input. Thus, using MACE-implemented systems dramatically improves the accessibility of model checking to the typical programmer.

4.1 Preparing the System

To model check a system, the user writes a driver application suitable for model checking that should initialize the system, perform desired system input events, and check high-level system progress with liveness properties. For example, to look for bugs in a file distribution protocol, the test driver could have one node supply the file, and the remaining nodes request the file. The liveness property would then require that all nodes have received the file and the file contents match. Or for a consensus protocol, a simulated driver could propose a different value from each node, and the liveness property would be that each node eventually chooses a value and that all chosen values match. The MACEMC application links with the simulated driver, the user's compiled MACE object files, and MACE libraries. MACEMC simulates a distributed environment to execute the system—loading different simulator-specific libraries for random number generation, timer scheduling, and message transport—to explore a variety of event orderings for a particular system state and input condition.

Non-determinism To exhaustively and correctly explore different event orderings of the system, we must ensure that the model checker controls all sources of non-determinism. So far, we have assumed that the scheduling of pending ⟨node, event⟩ pairs accounts for all non-determinism, but real systems often exhibit non-determinism *within* the event handlers themselves, due to, e.g., randomized algorithms and comparing timestamps. When being model checked, MACE systems automatically use the deterministic simulated random number generator provided by MACEMC and the support for simulated time, which we discuss below. Furthermore, we use special implementations of the MACE libraries that internally call Toss at every non-deterministic choice point. For example, the TCP transport service uses Toss to decide whether to break a socket connection, the UDP transport service uses Toss to determine which message to deliver (allowing out-of-order messages) and when to drop messages, and the application simulator uses Toss to determine whether to reset a node. Thus, by systematically exploring the *sequences* of return values of Toss (as described in MaceMC_Search in the previous section), MACEMC analyzes all different sequences of internal non-deterministic choices. Additionally, this allows MACEMC to deterministically replay executions for a given sequence of choices.

Time Time introduces non-determinism, resulting in executions that may not be replayable or, worse, impossible in practice. For example, a system may branch based on the relative value of timestamps (e.g., for message timeout). But if the model checker were to use actual values of time returned by `gettimeofday()`, this comparison might always be forced along one branch as the simulator fires events faster than a live execution. Thus, MACEMC must represent time abstractly enough to permit exhaustive exploration, yet concretely enough to only explore feasible executions. In addition, MACEMC requires that executions be deterministically replayable by supplying an identical sequence of chosen numbers for all non-deterministic operations, including calls to `gettimeofday`.

We observed that systems tend to use time to: (i) manage the passage of real time, e.g., to compare two timestamps when deciding whether a timeout should occur, or, (ii) export the equivalent of monotonically increasing sequence numbers, e.g., to uniquely order a single node's messages. Therefore, we address the problem of managing time by introducing two new MACE object primitives—`MaceTime` and `MonotoneTime`—to obtain and compare time values. When running across a real network, both objects are wrappers around `gettimeofday`. However, MACEMC treats every comparison between `MaceTime` objects as a call to Toss and implements `MonotoneTime` objects with counters. Developers concerned with negative clock adjustments (and more generally non-monotone `MonotoneTime` implementations) can strictly use `MaceTime` to avoid missing bugs, at the cost of extra states to explore. Compared to state of the art model checkers, this approach frees developers from manually replacing time-based non-determinism with calls to Toss, while limiting the amount of needless non-determinism.

4.2 Mitigating State Explosion

One stumbling block for model-checking systems is the exponential explosion of the state space as the search depth increases. MACEMC mitigates this problem using four techniques to find bugs deep in the search space.

1. Structured Transitions The event-driven, non-blocking nature of MACE code significantly simplifies the task of model-checking MACE implementations and improves its effectiveness. In the worst case, a model checker would have to check all possible orderings of the assembler instructions across nodes with pending events, which would make it impractical to explore more than a few hundred lines of code across a small number of nodes. Model checkers must develop techniques for identifying larger atomic steps. Some use manual marking, while others interpose communication primitives. Non-blocking, atomic event handlers in MACE allow us to use event-handler code blocks as the fundamental unit of execution. Once a given code block runs to completion, we return control to MACEMC. At this point, MACEMC checks for violations of any safety or liveness conditions based on global system state.

2. State Hashing When the code associated with a particular event handler completes without a violation, MACEMC calculates a hash of the resulting system state. This state consists of the concatenation of the values of all per-node state variables and the contents of all pending, system-wide events. The programmer may optionally annotate MACE code to ignore the value of state variables believed to not contribute meaningfully to the uniqueness of global system state, or to format the string representation into a canonical form to avoid unneeded state explosion (such as the order of elements in a set). MaceMC_Simulator checks the hash of a newly-entered state against all previous state hashes. When it finds a duplicate hash, MACEMC breaks out of the current execution and begins the next sequence. In our experience, this allows MACEMC to avoid long random walks for 50-90 percent of all executions, yielding speedups of 2-10.

3. Stateless Search MACEMC performs backtracking by re-executing the system with the sequence of choices used to reach an earlier state, similar to the approach taken by Verisoft [11]. For example, to backtrack from the system state characterized by the sequence $\langle 0, 4, 0 \rangle$ to a subsequent system state characterized by choosing the sequence $\langle 0, 4, 1 \rangle$, MACEMC reruns the system from its initial state, re-executing the event handlers that correspond to choosing events 0 and 4 before moving to a different portion of the state space by choosing the event associated with value 1. This approach is simple to implement and does not require storing all of the necessary state (stack, heap, registers) to restore the program to an intermediate state. However, it incurs additional CPU overhead to re-execute system states previously explored. We have found trading additional CPU for memory in this manner to be reasonable because CPU time has not proven to be a limitation in isolating bugs for MACEMC. However, the stateless approach is not fundamental to MACEMC—we are presently exploring hybrid approaches that involve storing some state such as sequences for best-first searching or state for checkpointing and restoring system states to save CPU time.

4. Prefix-based Search Searching from an initial global state suffers the drawback of not reaching significantly past initialization for the distributed systems we consider. Further, failures during the initial join phase do not have the opportunity to exercise code paths dealing with failures in normal operation because they simply look like an aborted join attempt (e.g., resulting from dropped messages) followed by a retry. To find violations in steady-state system operation, we run MACEMC to output a number of live executions of sufficient length, i.e., executions where all liveness conditions have been satisfied, all nodes have joined, and the system has entered steady-state operation. We then proceed as normal from one of these live prefixes with exhaustive searches for safety violations followed by random walks from the perimeter to isolate and verify liveness violations. We found the PASTRY bug described in the introduction using a prefix-based search.

4.3 Biasing Random Walks

We found that choosing among the set of all possible actions with equal probability had two undesirable consequences. First, the returned error paths had unlikely event sequences that obfuscated the real cause of the violation. For example, the system generated a sequence where the same timer fired seven times in a row with no intervening events, which would be unlikely in reality. Second, these unlikely sequences slowed system progress, requiring longer random walks to reach a live state. Setting d_{max} large enough to ensure that we had allowed enough time to reach live states slowed FindCriticalTransition by at least a factor of ten.

We therefore modified Toss to take a set of weights corresponding to the rough likelihood of each event occurring in practice. Toss returns an event chosen randomly with the corresponding probabilities. For example, we may prioritize application events higher than message arrivals, and message arrivals higher than timers firing. In this way, we *bias* the system to search event sequences in the random walk with the hope of reaching a live state sooner, if possible, and making the error paths easier to understand.

Biasing the random walks to common sequences may run counter to the intuition that model checkers should push the system into corner conditions difficult to predict or reason about. However, recall that we run random walks only after performing exhaustive searches to a certain depth. Thus, the states reached by the periphery of the exhaustive search encompass many of these tricky corner cases, and the system has already started on a path leading to—or has even entered—a dead state.

One downside to this approach is that the programmer must set the relative weights for different types of events. In our experience, however, every event has had a straightforward rough relative probability weighting. Further, the reductions in average depth before transitioning to a live state and the ease of understanding the violating executions returned by MACEMC have been worthwhile. If setting the weights proves challenging for a particular system, MACEMC can be run with unbiased random walks.

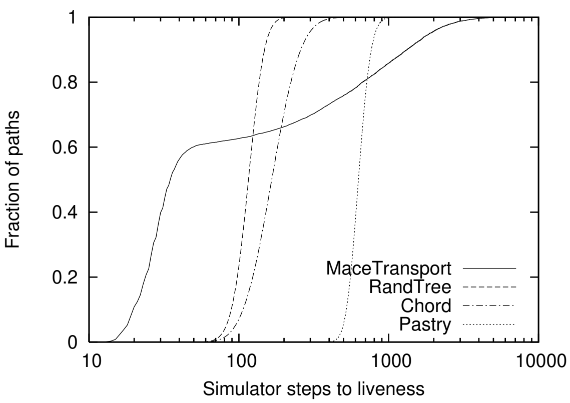

Figure 2: CDF of simulator steps to a live state at a search depth of 15.

4.4 Tuning MACEMC

In addition to event weights discussed above, MACEMC may be tuned by setting d_{max} (random walk depth), k (number of random walks), and a wide variety of knobs turning features on and off. Feature knobs include whether to test node failures, socket failures, UDP drops, UDP reordering, and the number of simulated nodes, and are generally easy to set based on the target test environment.

Setting k is a bit more complex. k represents the tradeoff between the time to complete the critical transition algorithm and the possibility that the reported critical transition is before the actual critical transition. This occurs when k random executions of d_{max} steps did not satisfy liveness, but some other path could have. We informally refer to this occurrence as "near dead". In our tests, we general use k between 20 and 60. At 60, we have not observed any prematurely reported critical transitions, while at 20 we occasionally observe the reported critical transition off by up to 2 steps. To tune k, the programmer considers the output critical transition. If it is not obvious why it is the critical transition, the programmer can increase k and re-run to refine the results.

Finally, we discuss how to set d_{max}. We ran MACEMC over four systems using random walks to sample the state space beyond an exhaustive search to 15 steps. Figure 2 plots the fraction of executions that reached the first live state at a given depth. What we observe is that in these four systems, since all sample executions reached a live state by 10,000 steps, a random execution that takes 80,000 steps to reach a live state would be a significant outlier, and likely somewhere along the execution it became trapped in a region of dead states. Setting d_{max} too low generally leads to the critical transition algorithm reporting condition C2, which is what we treat as the signal to increase d_{max}.

Figure 2 also illustrates that the depths required to initially reach a live state are much greater than what can be found with exhaustive search. MACEMC found only 60% of executions reached a live state for MACETRANSPORT after considering 50 steps (the edge of what can be exhaustively searched using state-of-the-art model checkers), less than 1% of executions for RANDTREE and CHORD, and none of the executions for PASTRY.

5 MACEMC Debugger

Although MACEMC flags violating executions and identifies the critical transition that likely led the system to a dead state, the developer must still understand the sequence of events to determine the root cause of the error. This process typically involves manually inspecting the log files and hand-drawing sketches of evolving system state. To simplify this process, we built MDB, our debugging tool with support for interactive execution, replay, log analysis, and visualization of system state across individual nodes and transitions. MDB is similar in function to other work in distributed debuggers such as the WiDS Checker [22] and Friday [10]. MDB allows the programmer to: (i) perform single step system execution both forward and backward, (ii) jump to a particular step, (iii) branch execution from a step to explore a different path, (iv) run to liveness, (v) select a specific node and step through events only for that node, (vi) list all the steps where a particular event occurred, (vii) filter the log using regular expressions, and (viii) *diff* the states between two steps or the same step across different executions by comparing against a second, similar log file.

MDB also generates event graphs that depict inter-node communication. It orders the graph by nodes on the x-axis and simulator steps on the y-axis. Each entry in the graph describes a simulated event, including the transition call stack and all message fields. Directional arrows represent message transmissions, and other visual cues highlight dropped messages, node failures, etc.

MDB recreates the system state by analyzing detailed log files produced by MACEMC. While searching for violations, MACEMC runs with all system logging disabled for maximum efficiency. Upon discovering a violation, MACEMC automatically replays the path with full logging. The resulting log consists of annotations: (i) written by the programmer, (ii) generated automatically by the MACE compiler marking the beginning and end of each transition, (iii) produced by the simulator runtime libraries, such as timer scheduling and message queuing and delivery, and (iv) generated by the simulator to track the progress of the run, including random number requests and results, the node simulated at each step, and the state of the entire system after each step. For our runs, logs can span millions of entries (hundreds to thousands of

$ **./mdb error.log**
(mdb 0) **j 5**
(mdb 5) **filediff live.log**
. . .
localaddress=2.0.0.1:10201
out=[
− *OutgoingConnection(1.0.0.1:10201, connection=ConnectionInfo(cwnd=**2**, packetsSent=**2**, acksReceived=**1**, packetsRetransmitted=0),*
− *inflight=[**6002** → MessageInfo(seq=**6002**, syn=**0**, retries=0, timeout=true)],*
− *rtbuf=[ ], sendbuf=[ ], curseq=6002, dupacks=0, last=**6001**)*
+ *OutgoingConnection(1.0.0.1:10201, connection=ConnectionInfo(cwnd=**1**, packetsSent=**1**, acksReceived=**0**, packetsRetransmitted=0),*
+ *inflight=[**6001** → MessageInfo(seq=**6001**, syn=**1**, retries=0, timeout=true)],*
+ *rtbuf=[ ], sendbuf=[**MessageInfo(seq=6002, syn=0, timer=0, retries=0, timeout=true)**], curseq=6002, dupacks=0, last=**0**)*
]
in=[ ]
−*timer<retransmissionTimer>([dest=1.0.0.1:10201, msg=MessageInfo(seq=**6002**, syn=**0**, retries=0, timeout=true)])*
+*timer<retransmissionTimer>([dest=1.0.0.1:10201, msg=MessageInfo(seq=**6001**, syn=**1**, retries=0, timeout=true)])*
. . .

Figure 3: MDB session. Lines with differences are shown in italics (− indicates the error log, + the live log), with differing text shown in bold. The receiver is IP address 1.0.0.1 and the sender is 2.0.0.1.

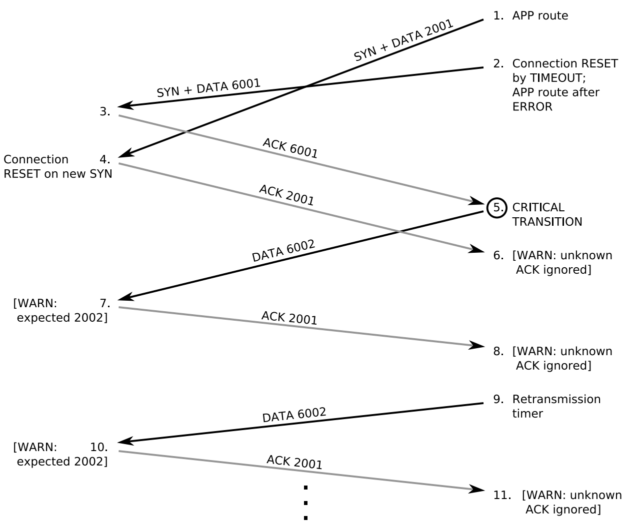

Figure 4: Automatically generated event graph for MACETRANSPORT liveness bug.

megabytes).

To demonstrate the utility of our debugging tools for diagnosing and fixing errors, we consider a case study with a bug in MACETRANSPORT: a reliable, in-order, message delivery transport with duplicate-suppression and TCP-friendly congestion-control built over UDP. Unlike TCP, MACETRANSPORT is fundamentally message- rather than stream-oriented, making it a better match for certain higher-level application semantics. As such, rather than using sequence numbers to denote byte offsets as with TCP, MACETRANSPORT assigns an incrementing sequence number to each packet. To obtain lower-latency communication, MACETRANSPORT avoids a three-way handshake to establish initial sequence numbers. A key high-level liveness property for MACETRANSPORT is that eventually every message should be acknowledged (unless the connection closes).

MACEMC found a violating execution of the "AllAcked" property in Table 1, where a sender attempts to send two messages to a receiver. Figure 4 shows a pictorial version of the event graphs automatically generated by MDB; the actual event graph is text-based for convenience and contains more detail. In Step 1, the sender sends a data packet with the SYN flag set and sequence number 2001. In Step 2, the retransmission timer causes the connection to close and MACETRANSPORT signals an error to the application. The application responds by attempting to resend the packet, causing MACETRANSPORT to open a new connection with sequence number 6001. At this point, both the old "SYN 2001" and the new "SYN 6001" packets are in flight. In Step 3, the network delivers the packet for the new 6001 connection, and the receiver replies by sending an "ACK 6001" message. In Step 4, the network delivers the out-of-order "SYN 2001" message, and the receiver responds by closing the connection on 6001, thinking it is stale, and opening a new incoming connection for 2001.

Unfortunately, in Step 5 (the critical transition) the sender receives the "ACK 6001." Believing the 6000-sequence connection to be established, the sender transmits "DATA 6002," at odds with the receiver's view. From here on, the execution states are dead as the receiver keeps ignoring the "DATA 6002" packet, sending ACKs for the 2001 connection instead, while the sender continues to retransmit the "DATA 6002" packet, believing it to be the sequence number for the established connection.

We illustrate a portion of an MDB session analyzing this bug in Figure 3. We load the error log in MDB, jump to the critical transition step (5), and *diff* the state with the live path with the longest shared prefix (output by MACEMC while searching for the critical transition (see §3.2)). The excerpt shows the state for the sender node. The key insight from this output is that in the live execution (lines indicated with $+$), the retransmission timer is scheduled with "SYN 6001," meaning that the packet could be retransmitted and the receiver could become resynchronized with the sender. Comparing the differences with the violating execution (lines indicated with $-$), where 6001 has been removed from the *inflight* map and timer because of the ACK, allows us to identify and fix the bug by attaching a monotonically increasing identifier in the SYN packets, implemented using a `MonotoneTime` object. Now, when the receiver gets the "SYN 2001" message out of order, it correctly concludes from the identifier that the message is stale and should be ignored, allowing acknowledgment of the "DATA 6002" message.

6 Experiences

We have used MACEMC to find safety and liveness bugs in a variety of systems implemented in MACE, including a reliable transport protocol, an overlay tree, PASTRY, and CHORD. With the exception of CHORD, we ran MACEMC over mature implementations manually debugged both in local- and wide-area settings. MACEMC found several subtle bugs in each system that caused violations of high-level liveness properties. All violations (save some found in CHORD, see below) were beyond the scope of existing software model checkers because the errors manifested themselves at depths far beyond what can be exhaustively searched. We used the debugging process with CHORD as control—we first performed manual debugging of a new implementation of CHORD and then employed MACEMC to compare the set of bugs found through manual and automated debugging.

Table 2 summarizes the bugs found with MACEMC to date. This includes 52 bugs found in four systems. Spanning the three mature systems, the 33 bugs across 1500 lines of MACE code correspond to one bug for every 50 lines of code. MACEMC actually checks the generated C++ code, corresponding to one bug for every 250 lines of code. In the only comparable check of a complex distributed system, CMC found approximately one bug for every 300 lines of code in three versions of the AODV routing protocol [25]. Interestingly, more than 50% of the bugs found by CMC were memory handling errors (22/40 according to Table 4 [25]) and all were safety violations. The fact that MACEMC finds nearly the same rate of errors while focusing on an entirely different class of liveness errors demonstrates the complementary nature of the bugs found by checking for liveness rather than safety violations. To demonstrate the nature and complexity of liveness violations we detail two representative violations below; we leave a detailed discussion of each bug we found to a technical report [17].

System	Bugs	Liveness	Safety	LOC
MaceTransport	11	5	6	585/3200
RandTree	17	12	5	309/2000
Pastry	5	5	0	621/3300
Chord	19	9	10	254/2200
Totals	52	31	21	

Table 2: Summary of bugs found for each system. LOC=Lines of code and reflects both the MACE code size and the generated C++ code size.

Typical MACEMC run times in our tests have been from less than a second to a few days. The median time for the search algorithm has been about 5 minutes. Typical critical-transition algorithm runtimes are from 1 minute to 3 hours, with the median time being about 9 minutes.

RANDTREE implements a random overlay tree with a maximum degree designed to be resilient to node failures and network partitions. This tree forms the backbone for a number of higher-level aggregation and gossip services including our implementations of Bullet [21] and RanSub [20]. We have run RANDTREE across emulated and real wide-area networks for three years, working out most of the initial protocol errors.

RANDTREE nodes send a "Join" message to a bootstrap node, who in turn forwards the request up the tree to the root. Each node then forwards the request randomly down the tree to find a node with available capacity to take on a new child. The new parent adds the requesting node to its child set and opens a TCP connection to the child. A "JoinReply" message from parent to child confirms the new relationship.

Property. A critical high-level liveness property for RANDTREE (and other overlay tree implementations) is that all nodes should eventually become part of a single spanning tree.

We use four separate MACE liveness properties to capture this intuition: (i) there are no loops when following $parent$ pointers, (ii) a node is either the root or has a parent, (iii) there is only one root (shown in Table 1), and (iv) each node N's parent maintains it as a child, and N's children believe N to be their parent.

Violation. MACEMC found a liveness violation where two nodes A, D have a node C in their child set, even though C's parent pointer refers to D. Along the violating execution, C initially tries to join the tree under B, which forwards the request to A. A accepts C as a child and sends it a "JoinReply" message. Before establishing the connection, C experiences a node reset, losing all state. A, however, now establishes the prior connection with the new C, which receives the "JoinReply' and ignores it (having been reinitialized). Node C then attempts to join the tree but this time is routed to D, who accepts C as a child. Node A assumes that if the TCP socket to C does not break, the child has received the "JoinReply" message and therefore does not perform any recovery. Thus, C forever remains in the child sets of A and D.

Bug. The critical transition for this execution is the step where C receives the "JoinReply" from A. MDB reveals that upon receiving the message, C ignores the message completely, without sending a "Remove" message to A. Along the longest live *alternate* path found from the state prior to the critical transition, we find that instead of receiving A's join reply message, C gets a request from the higher-level application asking it to join the overlay network, which causes C to transition into a "joining" mode from its previous "init" mode. In this alternate path, C subsequently receives A's "JoinReply" message, and correctly handles it by sending A a "Remove" message. Thus, we deduced that the bug was in C's ignoring of "JoinReply" messages when in the "init" mode. We fix the problem by ensuring that a "Remove" reply is sent in this mode as well.

CHORD specifies a key-based routing protocol [30]. CHORD structures an overlay in a ring such that nodes have pointers to their successor and predecessor in the key-space. To join the overlay a new node gets its predecessor and successor from another node. A node inserts itself in the ring by telling its successor to update its predecessor pointer, and a stabilize procedure ensures global successor and predecessor pointers are correct through each node probing its successor.

Property. We use a liveness property to specify that all nodes should eventually become part of a single ring (see Table 1). This minimal correctness condition guarantees that routing reach the correct node.

Violation. MACEMC found a liveness violation in the very first path it considered. This was not unexpected, given that CHORD had not been tested yet. However, the critical transition algorithm returned transition 0 and condition C2, implying that the algorithm could not determine if the path had run long enough to reach liveness.

Looking at the event graph, we saw the nodes finished their initial join quickly (step 11), and spent the remaining steps performing periodic recovery. This process suggested that the system as a whole was dead, since reaching a live state would probably not require tens of thousands of transitions when the initial join took only 11.

MDB showed us that mid-way through the execution, client0's successor pointer was client0 (implying that it believed it was in a ring of size 1), which caused the liveness predicate to fail. The other nodes' successor pointers correctly followed from client1 to client2 to client0. We believed the $stabilize$ procedure should correct this situation, expecting client2 to discover that client0 (its successor) was in a self-loop and correct the situation. Looking at this procedure in the event graph, we saw that there was indeed a probe from client2 to client0. However, client2 ignored the response to this probe. We next jumped to the transition in MDB corresponding to the probe response from the event graph. In fact, client0 reported that client2 was its predecessor, so client2 did not correct the error.

Starting at the initial state in MDB we stepped through client0's transitions, checking its state after each step to see when the error symptom occurs. After 5 steps, client0 receives a message that causes it to update its predecessor but *not* its successor, thus causing the bug.

Bug. This problem arose because we based our original implementation of CHORD on the original protocol [30], where a joining node explicitly notified its predecessor that it had joined. We then updated our implementation to the revised protocol [31], which eliminated this notification and specified that all routing state should be updated upon learning of a new node. However, while we removed the join notification in our revisions, we failed to implement the new requirements for updating routing state, which we overlooked because it concerned a seemingly unrelated piece of code. We fixed the bug by correctly implementing the new protocol description.

Overall, both our manual testing and model checking approaches found slightly different sets of bugs. On the one hand, manual testing found many of the correctness bugs and also fixed several performance issues (which cannot be found using MACEMC). Manual testing required that we spend at least half of our time trying to determine whether or not an error even occurred. A single application failure may have been caused by an artifact of the experiment, or simply the fact that the liveness properties had not yet been satisfied. Because of these complexities, identifying errors by hand took anywhere from 30 minutes to several hours per bug.

On the other hand, MACEMC did find some additional correctness bugs and moreover required less human time to locate the errors. MACEMC examines the state-snapshot across all nodes after each atomic event and reports only known bugs, thereby eliminating the guesswork of determining whether an error actually occurred. Furthermore, the model checker outputs which property failed and exactly how to reproduce the circumstances of the failure. MACEMC also produces a verbose log and event graph, and in the case of liveness violations, an alternate path which would have been successful. These features make it much easier to verify and identify bugs using MACEMC, without the hassle of conducting experiments that require running many hosts on a network. We spent only 10 minutes to an hour using MACEMC to find the same bugs that we painstakingly identified earlier with manual testing; and we found the new bugs (those not caught with manual testing) in only tens of minutes.

7 Related Work

Our work is related to several techniques for finding errors in software systems that fall under the broad umbrella of Model Checking.

Classical Model Checking. "Model Checking," i.e., checking a system described as a graph (or a Kripke structure) was a model of a temporal logic formula independently invented in [6, 27]. Advances like *Symmetry Reduction, Partial-Order Reduction*, and *Symbolic Model Checking* have enabled the practical analysis of hardware circuits [2, 23], cache-coherence and cryptographic protocols [9], and distributed systems and communications protocols [15], which introduced the idea of state-hashing used by MACEMC. However, the tools described above require the analyzed software to be specified in a tool-specific language, using the state graph of the system constructed either before or during the analysis. Thus, while they are excellent for quickly finding *specification* errors early in the design cycle, it is difficult to use them to verify the systems *implementations*. MACEMC by contrast tests the C++ implementation directly, finding bugs both in the design and the implementation.

Model Checking by Random Walks. West [32] proposed the idea of using random walks to analyze networking protocols whose state spaces were too large for exhaustive search. Sivaraj and Gopalakrishnan [29] propose a method for iterating exhaustive search and random walks to find bugs in cache-coherence protocols. Both of the above were applied to check safety properties in systems described using specialized languages yielding finite state systems. In contrast, MACEMC uses random walks to find liveness bugs by classifying states as dead or transient, and further, to pinpoint the critical transition.

Model Checking by Systematic Execution. Two model checkers that directly analyze implementations written in C and C++ are VERISOFT [11] and CMC [25]. VERISOFT views the entire system as several *processes* communicating through message queues, semaphores and shared variables *visible* to VERISOFT. It schedules these processes and traps calls that access shared resources. By choosing the process to execute at each such trap point, the scheduler can exhaustively explore all possible interleavings of the processes' executions. In addition, it performs stateless search and partial order reduction allowing it to find critical errors in a variety of complex programs. Unfortunately, when we used VERISOFT to model-check MACE services, it was unable to exploit the atomicity of MACE's transitions, and this combined with the stateless search meant that it was unable to exhaustively search to the depths required to find the bugs MACEMC found. A more recent approach, CMC [25], also directly executes the code and explores different executions by interposing at the scheduler level. CMC has found errors in implementations of network protocols [24] and file systems [34]. JAVAPATHFINDER [14] takes an approach similar to CMC for JAVA programs. Unlike VERISOFT, CMC, and JAVAPATHFINDER, MACEMC addresses the challenges of finding liveness violations in systems code and simplifying the task of isolating the cause of a violation.

Model Checking by Abstraction. A different approach to model checking software implementations is to first *abstract* them to obtain a finite-state model of the program, which is then explored exhaustively [3, 4, 7, 8, 12, 16] or up to a bounded depth using a SAT-solver [5, 33]. Of the above, only FEAVER and BANDERA can be used for liveness-checking of concurrent programs, and they require a user to manually specify how to abstract the program into a finite-state model.

Isolating Causes from Violations. Naik et al. [26] and Groce [13] propose ways to isolate the cause of a safety violation by computing the difference between a violating run and the closest non-violating one. MACEMC instead uses a combination of random walks and binary search to isolate the critical transition causing a liveness violation, and then uses a live path with a common prefix to help the programmer understand the root cause of the bug.

8 Conclusions

The most insidious bugs in complex distributed systems are those that occur after some unpredictable sequence of asynchronous interactions and failures. Such bugs are difficult to reproduce—let alone fix—and typically manifest themselves as executions where the system is unable to *ever* enter some desired state after an error occurs. In other words, these bugs correspond to violations of *liveness* properties that capture the designer's intention of how the system should behave in steady-state operation. Though prior software model checkers have dramatically improved our ability to find and eliminate errors, elusive bugs like the subtle error we found in PASTRY have been beyond their reach, as they only find violations of *safety* properties.

We have described techniques that enable software model checkers to heuristically isolate the complex bugs that cause liveness violations in systems implementations. A key insight behind our work is that many interesting liveness violations correspond to the system entering a *dead* state, from which recovery to the desired state is impossible. Though a safety property describing dead states exists mathematically, it is often too complex and implementation-specific for the programmer to specify without knowing the exact bug in the first place. Thus, we have found that the process of finding the errors that cause liveness violations often reveals previously unknown safety properties, which can be used to find and fix more errors. We have used MACEMC to find 31 liveness (and 21 safety) errors in MACE implementations of four complex distributed systems. We believe that our techniques—a combination of state-exploration, random walks, critical transition identification, and MDB—radically expand the scope of implementation model checkers to include liveness violations, thereby enabling programmers to isolate subtle errors in systems implementations.

Acknowledgements

We would like to thank our shepherd, Petros Maniatis, for his many insights and contributions to this paper, and our anonymous reviewers, for their valuable comments.

References

[1] Freepastry: an open-source implementation of pastry intended for deployment in the internet. `http://freepastry.rice.edu`, 2006.

[2] ALUR, R., HENZINGER, T., MANG, F., QADEER, S., RAJAMANI, S., AND TASIRAN, S. MOCHA: modularity in model checking. In *Computer-aided Verification (CAV)*, A. Hu and M. Vardi, Eds., Lecture Notes in Computer Science 1427. Springer-Verlag, 1998, pp. 521–525.

[3] BALL, T., AND RAJAMANI, S. The SLAM project: debugging system software via static analysis. In *Principles of Programming Languages (POPL)* (2002).

[4] CHEN, H., AND WAGNER, D. MOPS: an infrastructure for examining security properties of software. In *Computer and Communications Security (CCS)* (2002).

[5] CLARKE, E., KROENING, D., AND LERDA, F. A tool for checking ANSI-C programs. In *Tools and Algorithms for the Construction and Analysis of Systems (TACAS)* (2004), K. Jensen and A. Podelski, Eds., vol. 2988 of *Lecture Notes in Computer Science*, pp. 168–176.

[6] CLARKE, E. M., AND EMERSON, E. A. Synthesis of synchronization skeletons for branching time temporal logic. In *Logic of Programs* (1981), Lecture Notes in Computer Science 131.

[7] COOK, B., PODELSKI, A., AND RYBALCHENKO, A. Termination proofs for systems code. In *Programming Language Design and Implementation (PLDI)* (2006).

[8] CORBETT, J., DWYER, M., HATCLIFF, J., PASAREANU, C., ROBBY, LAUBACH, S., AND ZHENG, H. Bandera : Extracting finite-state models from Java source code. In *International Conference on Software Engineering (ICSE)* (2000).

[9] DILL, D., DREXLER, A., HU, A., AND YANG, C. H. Protocol verification as a hardware design aid. In *International Conference on Computer Design (ICCD)* (1992).

[10] GEELS, D., ALTEKAR, G., MANIATIS, P., ROSCOE, T., AND STOICA, I. Friday: Global comprehension for distributed replay. In *Networked Systems Design and Implementation (NSDI)* (2007).

[11] GODEFROID, P. Model checking for programming languages using Verisoft. In *Principles of Programming Languages (POPL)* (1997).

[12] GRAF, S., AND SAÏDI, H. Construction of abstract state graphs with PVS. In *Computer-aided Verification (CAV)*, Lecture Notes in Computer Science 1254. 1997, pp. 72–83.

[13] GROCE, A., AND VISSER, W. What went wrong: Explaining counterexamples. In *Spin Model Checking and Software Verification (SPIN)* (2003).

[14] HAVELUND, K., AND PRESSBURGER, T. Model checking Java programs using Java Pathfinder. *Software Tools for Technology Transfer (STTT) 2(4)* (2000), 72–84.

[15] HOLZMANN, G. The Spin model checker. *Transactions on Software Engineering 23*, 5 (1997), 279–295.

[16] HOLZMANN, G. Logic verification of ANSI-C code with SPIN. In *Spin Model Checking and Software Verification (SPIN)* (2000), Lecture Notes in Computer Science 1885.

[17] KILLIAN, C., ANDERSON, J., JHALA, R., AND VAHDAT, A. Life, death, and the critical transition: Finding liveness bugs in systems code. Tech. rep., University of California, San Diego. `http://mace.ucsd.edu/papers/MaceMC_TR.pdf`.

[18] KILLIAN, C., ANDERSON, J. W., BRAUD, R., JHALA, R., AND VAHDAT, A. Mace: Language support for building distributed systems. In *Programming Languages Design and Implementation (PLDI)* (2007).

[19] KINDLER, E. Safety and liveness properties: A survey. *EATCS-Bulletin*, 53 (1994).

[20] KOSTIĆ, D., RODRIGUEZ, A., ALBRECHT, J., BHIRUD, A., AND VAHDAT, A. Using Random Subsets to Build Scalable Network Services. In *USENIX Symposium on Internet Technologies and Systems (USITS)* (2003).

[21] KOSTIĆ, D., RODRIGUEZ, A., ALBRECHT, J., AND VAHDAT, A. Bullet: High bandwidth data dissemination using an overlay mesh. In *Symposium on Operating Systems Principles (SOSP)* (2003).

[22] LUI, X., LIN, W., PAN, A., AND ZHANG, Z. Wids checker: Combating bugs in distributed systems. In *Networked Systems Design and Implementation (NSDI)* (2007).

[23] MCMILLAN, K. L. A methodology for hardware verification using compositional model checking. *Science of Computer Programming 37*, (1–3) (2000), 279–309.

[24] MUSUVATHI, M., AND ENGLER, D. R. Model checking large network protocol implementations. In *Networked Systems Design and Implementation (NSDI)* (2004).

[25] MUSUVATHI, M., PARK, D., CHOU, A., ENGLER, D., AND DILL, D. CMC: A pragmatic approach to model checking real code. In *Operating Systems Design and Implementation (OSDI)* (2002).

[26] NAIK, M., BALL, T., AND RAJAMANI, S. From symptom to cause: Localizing errors in counterexample traces. In *Principles of Programming Languages (POPL)* (2003).

[27] QUEILLE, J., AND SIFAKIS, J. Specification and verification of concurrent systems in CESAR. In *International Symposium on Programming*, M. Dezani-Ciancaglini and U. Montanari, Eds., Lecture Notes in Computer Science 137. Springer-Verlag, 1981.

[28] ROWSTRON, A., AND DRUSCHEL, P. Pastry: Scalable, distributed object location and routing for large-scale peer-to-peer systems. In *Middleware* (2001).

[29] SIVARAJ, H., AND GOPALAKRISHNAN, G. Random walk based heuristic algorithms for distributed memory model checking. *Electr. Notes Theor. Comput. Sci. 89*, 1 (2003).

[30] STOICA, I., MORRIS, R., KARGER, D., KAASHOEK, F., AND BALAKRISHNAN, H. Chord: A scalable peer to peer lookup service for internet applications. In *ACM Special Interest Group on Data Communication (SIGCOMM)* (2001).

[31] STOICA, I., MORRIS, R., LIBEN-NOWELL, D., KARGER, D. R., KAASHOEK, M. F., DABEK, F., AND BALAKRISHNAN, H. Chord: a scalable peer-to-peer lookup protocol for internet applications. *Transactions on Networking 11*, 1 (2003), 17–32.

[32] WEST, C. H. Protocol validation by random state exploration. In *6th IFIP WG 6.1 International Workshop on Protocol Specification, Testing, and Verification* (1986).

[33] XIE, Y., AND AIKEN, A. Scalable error detection using boolean satisfiability. In *Principles of Programming Languages (POPL)* (2005).

[34] YANG, J., TWOHEY, P., ENGLER, D. R., AND MUSUVATHI, M. Using model checking to find serious file system errors. In *Operating Systems Design and Implementation (OSDI)* (2004).