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

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

There has been substantial recent progress in employing model checking to find bugs in unmodified systems implementations [1,2]. Model checking can improve code quality by finding bugs violating system-level correctness properties by systematically analyzing carefully constructed executions of the system. Existing software model checkers find violations of *safety* conditions. For example, a developer may specify that a pointer should never be dereferenced after it is freed or that a node's routing table should never accept advertised routes that contain the node itself to avoid loops. Unfortunately, finding bugs through safety properties has two problems. First, the programmer must already know the set nuances of the system which could cause it to fail (if you don't know accepting an advertised route containing the local node is a problem, you won't check that it doesn't happen). Second, problems exist where the system does not do anything *bad* but still never succeeds at its task.

We contend that for complex systems the *desirable* behaviors of the system may be specified more easily than identifying everything that could go wrong. Of course, specifying both desirable conditions and safety assertions is valuable; however, current model checkers do not have any mechanism for verifying whether desirable system properties can be achieved. Examples of such properties include: i) a reliable transport eventually delivers all messages even in the face of network losses and delays, ii) all nodes eventually join an overlay, and, iii) nodes in a distributed tree system eventually form (and re-form in the face of errors) a single spanning tree. These global *liveness* requirements specify that, in the limit, the system should achieve some particular condition.

Unfortunately, identifying liveness violations poses a much greater challenge than finding safety violations. For safety properties, simply finding a path that violates the given condition proves the violation, whereas liveness properties are not violated by a partial execution path not satisfying it, but rather that a path will not *eventually* satisfy it. For instance, flagging a liveness violation in a temporarily partitioned network is premature. Rather, the model checker must verify that the system will never recover. This can either be because is has entered a *dead* state where it becomes impossible to *ever* recover to a live state, or simply that it chooses a path which never recovers (but could at any point branch and recover). Model checkers that operate over finite-state specifications locate liveness violations by finding a cyclic path without any live states. This approach is infeasible when checking real systems (that have infinite state spaces) because such repeating sequences typically occur beyond the horizon that can be exhaustively searched (a few dozen system events) and because complex system interactions (e.g., a timer unrelated to an error firing periodically) means that a repeating sequence may not be found even if a search to sufficient depth were possible.

This poster presents a model checker and set of algorithms capable of finding both safety and liveness violations in systems implementations. To find liveness violations, we first perform an exhaustive search to some maximum depth (limited by CPU time, memory, or both). From this periphery we then execute strategically chosen random walks to sample the space beyond that which can be exhaustively searched to locate potential liveness violations. From this initial state we perform additional sampling to ensure a low rate of false positives in the set of violations reported to the developer.

Next, we address the difficult task of finding the actual system error that causes a particular liveness violation. Unfortunately, the shortest program execution prefix which is not a live path very likely has little to do with the error; rather, a much later operation usually causes the execution to become dead, after which recovery becomes impossible. To limit the tedium of wading through hundreds of steps to find the error, we developed an algorithm to isolate the *critical transition*—the point after which the system can never recover—likely to be the source of the error, an interactive debugging tool, MDB that allows forward and backward stepping through global events and per node state inspection, and automatically generated event graphs to visualize system behavior.

We have built a fully functional model checker, MACEMC, for analyzing safety and liveness properties of unmodified, complex systems implementations. We have run MACEMC on a number of systems, including an implementation of PASTRY, CHORD, a reliable transport protocol, and an overlay tree. We have used MACEMC to find around 50 bugs across these systems. After developing MDB and event graph visualization, we reduced the human time required to find and fix liveness violations from a few hours to 10-20 minutes. In addition to the poster, we plan to demonstrate MDB and the event graph generator, as well as the last-nail technique.

References

[1] GODEFROID, P. Model checking for programming languages using Verisoft. In *POPL 97: Principles of Programming Languages* (1997), ACM, pp. 174–186.

[2] MUSUVATHI, M., PARK, D., CHOU, A., ENGLER, D., AND DILL, D. CMC: A pragmatic approach to model checking real code. In *OSDI 02: Operating Systems Design and Implementation* (2002), ACM.