MODIST: Transparent Model Checking of Unmodified Distributed Systems.

Yang et al. NSDI 2009

What is model checking?

- *Abstract a program, explore its states. Find errors in the abstraction, especially edge cases.*
- What is a (good) model? (Expressed in some language: mathematics, traces, Promela, C++, English..)
    - Model what you want to check. Abstracts away detail. For example, internal program logic or certain nodes in the system (if we don't care about those behaviours). Yet, must reflect the underlying program.
    - Scope: You want to retain the detail/interesting behaviour. Must draw a line between useful and useless behaviour.
    - Feasibility: Should be easy enough to implement by a developer.
    - *Property checking: Model should have enough information to check a property.*
    - Utility: Should correspond to some "real world" system (something that you care about)
    - Ali: "Entities and relations between entities". Choose best entities and relations!
- What is a specification/property? (Intent)
- Property is what you want to check. For example, an assertion of a correctness property.
    - Safety v. Liveness properties: https://www.cs.cornell.edu/fbs/publications/DefLiveness.pdf
    - Specification, or property should reflect… end-goal of the system? Legal and illegal behaviour. Reflects assumptions made in the system. Intention of the system designer.
- What is model checking? (Search process)

What is model checking?

- Engineering: Model checking ~ Programs as
- Sciences: Scientific method (validation) ~ Physics models
- MC is a type of validation (there are others: formal verification)
  - Using it to validate models
- When we model check
  - We check for existence of illegal states by *searching*
  - “Traverse the search space” using some systematic search process.
  - *Exhaustive search : exploration of model “states” && check if those “states” are legal or not (relative to property)*

Modist ***MC***

- How is Modist a model checker?
- Model: graph of process states, with action/event transitions between states.
  - State = uniquely defined by sequence of actions (implicit entity)
- Property: global assertions on consistent snapshots (D3S), fail stop detection (crashes), *weak* liveness detection (processes getting stuck ~ revisiting the same state)
- Search
  - Record the events that happened. To reach same state, re-do the sequence of events.

Modist choices

- Implicit states good match for transparency. No rigid structure (blackbox input software), therefore no explicit state defined.
    - Unlike Mace, which requires explicit state notation (from dev), there is no such requirement in Modist.
- Blackbox => all that Modist observes are events at the sys call interface.
- Also, snapshotting process memory is expensive!

Modist v. Verdi v. Mace PL

- (Reports from breakouts)
- If you're building from scratch, then use Mace! If you have formal semantics, then use Verdi! If you have an existing system, use Modist! Modist for "now", other approaches for "future" systems.
- Developer effort variation: Modist — lowest effort, while for others the specific language/OS requirement
- Modist has no explicit state v. Explicit state in Verdi and Mace.
  - *Verdi (and Mace?) may have bugs … in the model*
  - Verdi there is a distinction between model and implementation
  - In Mace and Modist there is no distinction between model and implementation: any bug you find, you can "easily" reproduce and verify (no false positives)
- *Do people model check model checkers? Interesting idea… phd "project"?*
  - For Modist MC correctness less important, since you can verify/reproduce bugs with replay. And there is no certification of "no bugs". MC process as bug finder (no verification)
  - But Modist correctness.. hard!
- *Ultimately try to decrease dev effort*: Verdi (VST idea),
  - Verdi (VST) *transforms automagically* system into a more fault tolerant system
  - Mace (benefit from structure => other tools become possible)
  - Modist (blackbox bug finder)
- Model checking ~ *intelligent* fuzzer? Events as "inputs"
- Verdi hardest to use — have to know Coq, but strongest guarantees! (Assuming trusted computing base is trust)

Modist, the artifact

- OS system call API: great place for interposition/shim layering — failure injection (thin waist for applications)
- Finn said (intelligent) fuzzing, but *WHY must* Modist capture some of the OS semantics. Why can't modest behave "irrationally" at the shim layer? (i.e., why is "dumb fuzzing" a bad idea?)
    - Random fuzzing mostly exercises input validation code (finds bugs in code that did not properly validate the input).
    - *"Must play ball mostly, and drop it 10 steps in"* ~ "progress normally before doing weird things"
    - But, why conform to OS semantics?
        - Avoid false positives
        - Control non-determinism
        - Don't perturb the system under test
- Transparency has a cost -> shim layer should not be visible (to app, nor OS). It needs to respect whatever assumptions the app is making about OS behaviour (OS has a spec for its behaviour). Invisible ~ indistinguishable from the OS (shim layer has to respect OS semantics).
    - Otherwise crashes will appear frequently, but will be mostly false positives (not real bugs => won't be able to reproduce them, because OS behaviour would be diff. From shim layer behaviour)
- The shim could emulate a different operating system/API semantics.
    - Useful if I wanted to change my API and observe impact of change on applications

Modist

- Unmodified systems, not entirely true: Modist introduces code that changes the system (D3S assertions are compiled into the binary, static analysis of time also changes the system)
-

Modist eval

- Standard for MC: use existing systems, find some bugs
- Randomness is an issue: hard to guarantee determinism. *Big difficulty for real tools.*
- *Surprising?* Many bugs in deployed systems that are widely used.
  - Many of these seem like they should have been caught by testing (e.g., MPS back and forth liveness bug)
- Systems with ~172KLOC (BerkeleyDB).
  - Showing off the scalability of the interposition approach
- Paths and states ~ good measures?
  - They don't tell us where/when they find the bugs (maybe within first 1m of exec)
  - Am I exploring the "right" paths/states?
- Importance of search heuristics — randomness is important!

Next: Dapper

- Distributed tracing paper (our first “Google” paper).

- Not the first tracing paper, but the first industry take. Influential in modern distributing tracing systems

- Tracing is more practical than model checking, verification, exotic languages.. is it the right (only?) way to reason about complex deployed distributed system?