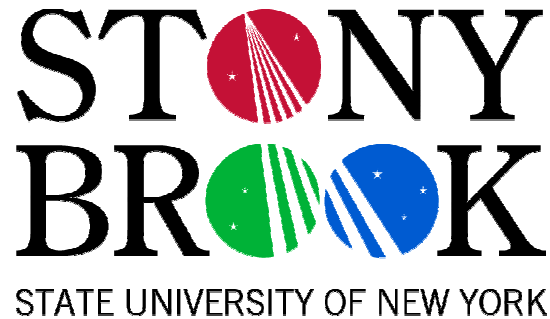


Software Model Checking: Where It Is, and Where It's Headed

Scott D. Stoller



Outline

- Introduction to Model Checking (MC)
- Software MC Success Stories
- Research Directions
 - ◆ Partial-order reduction
 - ◆ Heuristic search
 - ◆ MC + symbolic execution
 - ◆ Concrete (little abstraction) MC of Java, C, machine code
 - ◆ Heap abstractions
 - ◆ Environment modeling

Introduction to Model Checking

- **Model Checking (MC)**: systematic exploration of the possible behaviors of a system to determine whether the system satisfies a specified property.
- If property is **not satisfied**, the model checker provides a **counter-example**: a **path** in the state space that **violates** the property.
- **Abstraction**: **approximations** that reduce the cost of MC.
- **Abstraction** may cause **false alarms** (spurious counterexamples).
- **Unsound abstractions** may cause **missed errors**.
- MC is **path-sensitive**. Traditional **static analysis** isn't.

Explicit-State MC and Symbolic MC

- **Explicit-State MC**: states are manipulated **individually**.
- **Symbolic MC**: sets of states are represented by **logical formulas**. The set of successors of a set Φ of states is computed by manipulating Φ and the formula representing the system's transition relation.
- Symbolic MC using **OBDDs** (an efficient representation of boolean formulas) is dominant in **hardware verification**.
- **OBDDs** are **not** as widely used in **software verification**.
 - ◆ Hard to combine with partial-order reduction.
 - ◆ Harder to model dynamic memory allocation.
 - ◆ Use symbolic execution and constraints instead.

Outline

- Introduction to Model Checking (MC)
- **Software MC Success Stories**
- Research Directions
 - ◆ Partial-order reduction
 - ◆ Heuristic search
 - ◆ MC + symbolic execution
 - ◆ Concrete (little abstraction) MC of Java, C, machine code
 - ◆ Heap abstractions
 - ◆ Environment modeling

Success Story for MC with Abstraction: Static Driver Verifier (formerly SLAM)

- [Ball, Rajamani, et al., 2000-2005]
- Applied to all drivers developed by Microsoft. Released in Windows Device-Driver Development Kit, 2005.
- **Predicate abstraction** [Graf and Saidi 1997]: The data state of a C program **CP** is abstracted by the values of a set of **predicates**; e.g., start with predicates used in conditionals or in the property.
- This produces a **Boolean program BP**, with the usual control structures (loops, procedure calls, etc.), non-deterministic choice (to model "don't know"), and Boolean variables. Use program analysis and theorem prover to compute **BP**.

SDV: Model Checking of Boolean Programs

- Compute reachable data states at each point in BP.
- **Explicit+symbolic state representation:** explicit for program counter, symbolic (OBDDs) for sets of reachable data states.
- **Exploit scoping** of variables.
- **Procedure summaries:** The result of analyzing a procedure called in abstract state s_0 (captures global vars and args) is a set of resulting states $\{t_0, t_1, \dots\}$ (each captures global vars and return value). Add $(s_0, \{t_0, t_1, \dots\})$ to the procedure summary for later re-use.
 - ◆ This optimization **avoids redundant computation**.
 - ◆ It also **ensures termination** on recursive programs.

SDV: Counter-Example Guided Abstraction Refinement (CEGAR)

- BP overapproximates the possible behaviors of the C program, because the predicates capture limited info, and the theorem prover may diverge.
- If MCer says “true”, BP and CP satisfy the property.
- If MCer produces a counter-example, test its feasibility as follows. Use symbolic execution of CP to construct a predicate Φ that is satisfiable iff the counter-example is feasible. If Φ is satisfiable, CP does not satisfy the property. Otherwise, add predicates in theorem prover's proof that Φ is not satisfiable to the predicate abstraction, and repeat.

SDV: Killer Application

- Device drivers are a killer app for software MC.
- Faulty device drivers can easily crash your computer.
- Most device drivers are a manageable size (< 60 KLOC).
- It took man-years to write:
 - ◆ Formal specification of correct usage of Windows Device Driver API
 - ◆ Environment model (test harness) representing the Windows kernel
- But that effort is amortized over tens of thousands of device drivers.

Success Story for MC Without Abstraction: C Model Checker (CMC)

- [Musuvathi, Engler, Yang, Twohey, Park, Chou, Dill, 2002-now]
- Designed for **reactive systems**: supply an input, wait until system quiesces, record the resulting state.
- **State** = everything reachable from specified global vars (no call stack!)
 - ◆ Perform **heap canonicalization** while traversing heap
- **Explicit-state** MCer with traditional optimizations:
 - ◆ **Sub-structure sharing** for states on the DFS stack
 - ◆ **Hash compaction**: store 4-8 byte hashes (also called fingerprints) of visited states, instead of the states.

CMC: Heuristic, Properties Checked

- An **unsound abstraction (heuristic)**: ignore "less interesting" parts of the state when hashing, to avoid exploring similar states.
 - ◆ **Example**: When checking filesystems, hash only the state of the filesystems, not other parts of the heap or thread stacks.
- Check for general C **programming errors** (dangling pointers, array out-of-bounds, memory leaks, etc.) and **application-specific properties**.

CMC: Applications: Protocols

- 3 implementations of AODV (ad-hoc on-demand distance vector) routing protocol [OSDI 2002], about 7 KLOC each
 - ◆ Found 34 bugs
- Linux TCP [NSDI 2004]: 50KLOC, plus rest of kernel.
 - ◆ Found 4 bugs.
 - ◆ State vector: 200 KB.
 - ◆ 55% code coverage
 - ◆ 92% protocol coverage (code coverage in their reference implementation of TCP: a state machine implemented in 0.5 KLOC).

CMC: Applications: Filesystems

- Linux Filesystems: ext3, JFS, and ReiserFS [OSDI 2004].
- Found critical errors in all three, most patched within a day.
Found 32 bugs total.
- Each entry on stack is 1-3 MB.
- Non-determinism is used for:
 - ◆ Arguments of system calls
 - ◆ Branches dependent on environment variables and time
(this avoids need for constraints)
 - ◆ Success of memory allocation.

Outline

- Introduction to Model Checking (MC)
- Software MC Success Stories
- **Research Directions**
 - ◆ Partial-order reduction
 - ◆ Heuristic search
 - ◆ MC + symbolic execution
 - ◆ Concrete (little abstraction) MC of Java, C, machine code
 - ◆ Heap abstractions
 - ◆ Environment modeling

Partial-Order (Commutativity) Reductions

- An interesting state may be reachable by **many paths** that differ only in the order of operations that **commute** with each other.
- **Goal of POR:** Explore only one of those paths. This avoids exploring and storing intermediate states on the other paths.
- **Note:** SDV and CMC mostly ignore concurrency.
- **Traditional PORs** are **ineffective** for **shared-variable** concurrent programs, because all reads and writes to shared variables are classified as **non-commuting**.

Lock-Based Partial-Order Reductions

- In many programs, most shared variables are **protected** by **locks**: a thread must hold that lock when accessing the variable.
- Accesses to a lock-protected variable **commute** with concurrent transitions of other threads, because those transitions cannot be accesses to that variable.
- PORs specialized to **exploit locking** (mutual exclusion): [Stoller 2000], [Flanagan and Qadeer 2003], and [Dwyer, Hatcliff, et al., 2004]. Used in JPF, Bogor, Zing, etc.
- **Very effective** in programs that use locks.
- [Qadeer, Rajamani, and Rehof 2004]: lock-based POR and procedure summarization.

Heuristic Search: Property, Program Structure

- **Search algorithms:** A*, beam search, best-first (greedy) search, genetic algorithms
- **Property-based heuristics** [Leue, Edelkamp, et al., 2001]:
 - ◆ distance in control-flow graph (CFG) to an assertion
 - ◆ distance in property automaton to closest error state.
- **Program-based heuristics** [Groce and Visser, 2004]:
 - ◆ **Branch count:** prefer uncovered branches, otherwise non-branch instructions, otherwise infrequently taken branches.
 - ◆ **Choose-free:** avoid transitions that perform non-deterministic choice introduced by abstraction.

Heuristic Search: State Change

- Favor transitions that cause **larger state changes** (relative to initial state) or cause variables to take on **less frequented values**
 - ◆ Used in DIDUCE [Hangal and Lam 2002] and CMC [Musuvathi+, 2002].

Heuristic Search: Concurrency

- **Most blocked** (for deadlocks): favor transitions that cause a thread to block
- **Lock-order**: favor execution of threads involved in lock-order conflicts (acquiring locks in different orders) found during runtime monitoring [Havelund 2000].
- **Races** [Havelund 2000]: favor execution of threads involved in races found during runtime monitoring.
- **Synchronization coverage**: try to reach each synchronization statement once in a state where it blocks and once in a state where it does not block [Bron, Farchi, Magid, Nir, and Ur 2005].

Heuristic Search: Concurrency

- **Context-bounded MC:** favor executions with fewer context switches.
 - ◆ Specifically, explore all schedules with 1 context switch, then with 2 context switches, etc.
 - ◆ Especially useful with state-less search [Qadeer and Rehof 2005]

MC + Symbolic Execution

- **Symbolic execution (SymEx):** a static analysis in which inputs are represented by **symbols**, and computed values are represented by **expressions** and **constraints**.
- **Example:** `f(x,y) { int z=x; while (z>0) { z = z-y; ... } ... }`
 - ◆ **Sequence of states:** $z=X$. $z=X-Y$. $z=X-2Y$
 - ◆ **And constraints**, e.g., after 1 iteration, $X-Y > 0$ in loop body, $X-Y \leq 0$ at the point after the loop.
 - ◆ **Backtrack** if the accumulated constraints, called the **path condition**, are not satisfiable.
- **Unsound abstraction:** **bound** the length of explored executions, to ensure termination.

Testcase Generation Using MC + SymEx in JPF [Visser+ 2004]

- **Goal:** Find all non-isomorphic valid inputs up to a given size, by MC + SymEx of **Java code** for each method's **precondition**.
- **Case Study:** Red-black trees.
- Use MC + SymEx to create symbolic states representing these inputs.
- **Non-reference values** are handled as on previous slide.
- When a **symbolic reference** is used, materialize it to a known value, using **non-deterministic choice** between existing references (instances) of that type and a new one.
- Use **constraint solver** to materialize symbolic values in the resulting symbolic states.

Testcase Generation using MC + SymEx in XRT

- [Grieskamp, Tillmann, and Schulte, 2005]
- XRT is an **explicit-state MC** that supports **symbolic execution** with constraints.
- **Input language:** CIL, the intermediate language of Microsoft's CLR.
- Intended for **testcase generation** in unit testing: symbolically explore all feasible paths in the model (specification), and use constraint solver to create high-coverage test suite from the resulting path conditions.
- XRT is the **next generation** of **Spec#**.
- Developers will be able to write models in any language that compiles to CIL (e.g., C# or VB).

Bounded Verification of JML Specifications Using MC + SymEx [Robby et al., 2005]

- JML specifications are mainly **pre-conditions** and **post-conditions** for methods.
- **Start** in a symbolic state with the method **pre-condition asserted** as a constraint.
- **Symbolically execute** the method. Use **non-determinism** to materialize symbolic references.
- **Check** whether the **post-condition** is satisfied in the symbolic states at method exit points.
 - ◆ Instead of materializing the symbolic states into testcases.
- More efficient with **stronger pre-conditions**.

Symbolic + Concrete Execution in DART: Directed Automated Random Testing

- [Godefroid, Klarlund, Sen 2005] [Cadar and Engler 2005]
- Goal: generate test suites that cover of all feasible execution paths up to a given length in a program.
- Start with a symbolic and a random concrete input.
- Run the program, with concrete and symbolic execution, accumulating the path condition $\varphi_1 \wedge \dots \wedge \varphi_n$.
- Use constraint solver to find an input that satisfies $\varphi_1 \wedge \dots \wedge \varphi_{\{n-1\}} \wedge \neg\varphi_n$, or if we already explored the corresponding path, φ_1 and ... and $\neg\varphi_{\{n-1\}}$
- Select random values for unconstrained inputs. Repeat.
- **Case Study:** oSIP, a multi-media protocol. 30KLOC.

Concrete MC of Java

- **Concrete MC:** little use of **sound** abstractions.
 - ◆ Use (well chosen) **unsound** abstractions.
 - ◆ **Motivation:** Fewer **false alarms**. Much easier to apply to complex systems.
 - ◆ Very effective for **defect detection**.
- **Java Path Finder (JPF)** [Visser+, 2000+]
 - ◆ Explicit-state MC, in the SPIN [Holzmann] tradition, based on a JVM that can perform checkpointing and efficient backtracking.
 - ◆ Can handle on the order of 10 KLOC (hence the focus on testcase generation, etc.).

Concrete MC of Java: Bandera, Bogor [Dwyer, Hatcliff, ..., 2000+]

- **Bandera**: a toolset for software MC. Property specification language, program analyses and transformations (new **slicing** algorithms, data abstraction, ...), path exploration tool, etc.
 - ◆ Translates Java to intermediate representation to model checker input language. Last step implemented for multiple model checkers.
- **Bogor**: model checker, similar in concept to JPF, with its own extendable input language.
- **Case studies**: Java Grande benchmarks, Siena publish-subscribe middleware.

Concrete MC of C: VeriSoft [Godefroid 1997+]

- **Goal:** MC single-threaded multi-process systems, implemented in C, up to bounded execution depth.
- Intercept **non-deterministic operations** (scheduling decisions, calls to Verisoft.random in **test harness**). Systematically try **all possibilities**.
- VeriSoft stores **no states!** It stores **transitions** on a search stack. Backtracking is implemented by **restart+replay**.
 - ◆ Use POR to reduce redundant exploration of states.
- Applied successfully to **large telecom systems**.

Concrete MC of C: C Model Checker (CMC)

- [Musuvathi, Engler, et al., 2002+]
- Discussed earlier

Concrete MC of C: C Bounded Model Checker (CBMC)

- [Kroening, Clarke, et al., 2003+]
- Translate C program into a Boolean formula Φ representing all of the executions up to given length bound, by unwinding the transition relation and introducing fresh variables for each intermediate state.
- Use SAT solver to try to satisfy $\Phi \wedge \neg\text{correct}$.
- A satisfying assignment is a counterexample to **correct**.
- SAT solvers can handle formulas with millions of vars, tens of millions of clauses in CNF.
- CBMC verified equivalence of **Verilog implementations** and **C specifications** of DES and a simple CPU.

Concrete MC of Machine Language: CHESS [Qadeer & Rehof, 2005]

- **Why** MC machine language? C statements are not atomic.
 - ◆ Interrupt-driven embedded software.
 - ◆ Weak memory models
- Translate machine instructions into calls to functions in an **x86 simulator** written in **Zing**.
 - ◆ **Zing** [Rajamani+, 2003+]: explicit-state MC for a small OO language. Supports POR, procedure summaries, software transactions, context-bounded MC, stateful and state-less search.
- Applications so far are compiled from about 3 KLOC of C.

Concrete MC of Machine Language: Estes [Mercer and Jones, 2005]

- Use **debugger** to compute **transitions** of the program.
- **Load** process state from MC into GDB,
set desired **breakpoint**,
let GDB **execute** to the breakpoint,
extract the state from GDB.
- Cycle-accurate simulation
- Easily allows different granularity for different transitions.
- Applications: interrupt-driven embedded software.
- Applications so far are a few hundred lines of C + ASM.

Heap Abstraction

- Predicate abstraction + CEGAR works well for model checking properties that do not depend on details of heap.
Recall: Abstractions are defined by sets of nullary predicates (implicitly parameterized by the current state).
 - ◆ Example: $px(): x > 0$, $py(): *y.next \neq \text{NULL}$.
- Effective automatic abstraction for heap-intensive programs/properties is still a challenge.
- TVLA [Reps, Sagiv, et al., 2000+] is a framework for verification of heap-intensive properties. Abstractions are defined by sets of predicates with any arity.
 - ◆ Example: $pn(v1, v2): *v1.next == v2$
where $v1, v2$ range over Node.

Heap Abstraction

- An **abstract heap** (shape graph) contains:
 - ◆ **individual nodes** (each representing one object),
 - ◆ **summary nodes** (each representing one or more objects)
 - ◆ **truth values** (true, false, unknown) for the **predicates**, with the nodes of the abstract heap as arguments
- **Transfer functions** describe effect of program statements on abstract heap.
- TVLA defines **core predicates** + transfer functions for **lists**.
- **Application-specific** predicates + transfer functions also needed. Good progress on computing them automatically.
- **Deep analysis** of small programs. **Not yet scalable**.

Environment Modeling

- Model checking a component requires
 - ◆ **Driver**: model of components that call it
 - ◆ **Stubs**: models of components (e.g., libraries) it calls
- Writing them with appropriate abstraction can be **difficult**.
- **Static Driver Verifier** project invested an enormous amount of time in environment modeling.
- **Verification of TCP using CMC**: "One of the surprising results was that it was easier to run the entire Linux kernel in ... CMC ... than extract out TCP in a stand-alone version." Their stubs led to too many false alarms.
- **MC of Java programs**: Modeling Java API is an obstacle.

Outline

- Introduction to Model Checking (MC)
- Software MC Success Stories
- Research Directions
 - ◆ Partial-order reduction
 - ◆ Heuristic search
 - ◆ MC + symbolic execution
 - ◆ Concrete (little abstraction) MC of Java, C, machine code
 - ◆ Heap abstractions
 - ◆ Environment modeling