CSE503: SOFTWARE ENGINEERING

MODEL CHECKING

David Notkin
Spring 2011

Two Approaches to Model Checking

Reprise

- Explicit – represent all states
  - Use conventional state-space search
  - Reduce state space by folding equivalent states together
- Symbolic – represent sets of states using boolean formulae
  - Reduce huge state spaces by considering large sets of states simultaneously – to the first order, this is the meeting of BDDs (binary decision diagrams) and model checking (more later)
  - Convert state machines, logic formulae, etc. to boolean representations
  - Perform state space exploration using boolean operators to perform set operations
  - SAT solvers are often at the base of symbolic model checking

Example temporal logic properties

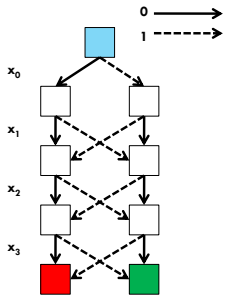
- Error states not reached (invariant)
  - **AG ¬Err**
- Eventually ack for each request (liveness)
  - **AG (Req → AF Ack)**
- Always possible to restart machine (possibility)
  - **AG EF Restart**

Representing sets

- Symbolic model checking needs to represent large sets of states concisely – for example, all even numbers between 0 and 127
  - Explicit representation
    - 0, 2, 4, 6, 8, 10, 12, 14, 16, 18, 20, 22, 24, 26, 28, 30, 32, 34, 36, 38, 40, 42, 44, 46, 48, 50, 52, 54, 56, 58, 60, 62, 64, 66, 68, 70, 72, 74, 76, 78, 80, 82, 84, 86, 88, 90, 92, 94, 96, 98, 100, 102, 104, 106, 108, 110, 112, 114, 116, 118, 120, 122, 124, 126
  - Implicit (symbolic) representation
    - $\neg x_0$ (x_0: least significant bit)
  - The size of the explicit representation grows with the bound, but not so for the implicit representation (in many cases)
- Need efficient boolean representation

Binary Decision Diagrams (BDDs)

- The original and most common representation is binary decision diagrams (BDDs) [Bryant 86]
- These are directed acyclic graphs evaluated as binary decision trees
- For the trivial example, these are trivial BDDs: x_0 **and** $\neg x_0$
- On the right is an example of a BDD for odd (even) parity of 4-bit numbers



What would odd parity look like if…

6

- …the bits in the BDD were ordered in reverse?
  $x_3x_2x_1x_0$
- Bit order $x_0x_1x_2x_3$ – compute BDD for $x_1x_0 + x_3x_2$
- Bit order $x_0x_1x_2x_3$ – compute BDD for $x_2x_0 + x_3x_1$
- Bit order $x_0x_1x_2x_3$ – compute BDD for $x_1x_0 * x_3x_2$
- Bit order $x_0x_1x_2x_3$ – compute BDD for $x_2x_0 * x_3x_1$
- **Take 5-10 minutes with 1-2 others to work these out**

Efficiency

7

- BDD size is often small in practice
- Some large hardware circuits can be handled
- Some well-known limitations: e.g., exponential size for a > bc
- Few theoretical results known
- Performance unpredictable
- When BDDs are manageable in size, model checking is generally efficient

Symbolic Model Checking

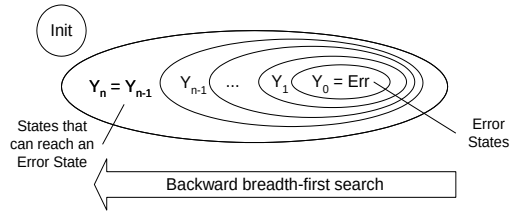
8

- Define boolean state variables
  - e.g., define $X = x_{n-1}, x_{n-2}, \ldots, x_0$ for an n-bit integer.
- A state set becomes a boolean function **S(X)**
  - the formulae for even numbers, odd parity, etc.
- Set operations ($\cap$,$\cup$) become boolean operations ($\wedge$,$\vee$)
- Transition relation: **R(X,X)**
- Compute predecessors using boolean operations: **Pre(S)** = $\exists X'.\ S(X') \wedge R(X,X')$
- In other words, turn everything into boolean algebra and represent the states – and the temporal formulae – as BDDs

Invariant Checking as Set Manipulation

- Compute $Y_i+1 = Pre(Y_i) \cup Y_i$
- Check if $Y_n \cap Init = \varnothing$

Can the initial state ever reach an error state?



Recap

- Check finite state machines vs. temporal logic formulae: yes or no with counterexample
- Symbolic model checking represents everything as BDDs and converts set operations over the state space to boolean operations over sets of states
- Need state machines, efficient BDDs, temporal logic formulae, etc.

Many FSM variations

- Deterministic and non-deterministic
- Mealy and Moore machines
- Transformers and acceptors
- Hierarchical state machines
  - Statecharts
  - RMSL
- The good news is that these are all theoretically equivalent representations
- That leaves the size of the state space as a key issue to address: in practice, state spaces have sufficient structure to be managed even when they are huge

Another key issue: abstraction

- Programs are not generally finite-state
  - Classic trivial example: recognizing nested parentheses requires unbounded state space (and it can be worse than this)
- So to use model checking we need to acquire a useful finite-state model
- Roughly two choices
  - Directly find a useful finite-state model
  - Produce a useful finite-state model from a non-finite-state model – and understand clearly what is and is not lost in that abstraction process
  - Door #3: bounded model checking

Check software specification

13

- Motivation: circa 1998-2000 – work here at UW CSE
- How to increase confidence in correctness of safety-critical software?
- Existing techniques useful with limitations: inspection, syntactic checking, simulation/testing, and theorem proving
- Symbolic model checking successful for industrial hardware
  - Effective also for software?
  - Many people's conjecture: No

Experts Said

14

- "The time and space complexity of [symbolic model checking] is affected…by the regularity of specification. Software requirements specifications lack this necessary regular structure…" [Heimdahl & Leveson 96]
- "[Symbolic model checking] works well for hardware designs with regular logical structures…However, it is less likely to achieve similar reductions in software specifications whose logical structures are less regular." [Cheung & Kramer 99]
- "…[symbolic model checkers] are often able to exploit the regularity…in many hardware designs. Because software typically lacks this regularity, [symbolic] model checking seems much less helpful for software verification." [Emerson 97]

Consider Safety-Critical Software

15

- Most costly bugs in specification
- Use analyzable formal specification
  - State-machine specifications
  - Intuitive to domain experts like aircraft engineers
  - Statecharts [Harel 87], RSML [Leveson et al. 94], SCR [Parnas et al.], etc.

Why is specification promising?

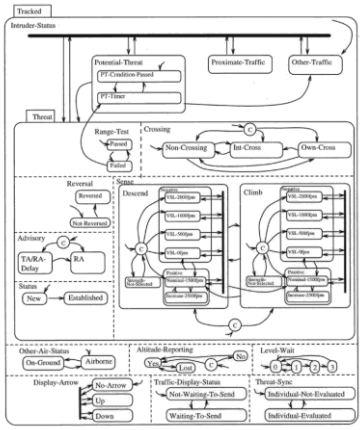
16

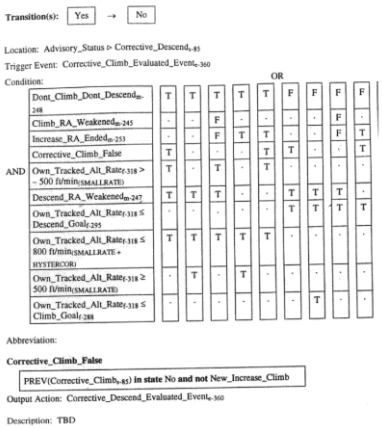
	Hardware	Spec	Multi-threaded Code
Data	Simple	Simple (except arithmetic)	Often complex
States	Finite	Finite (except arithmetic)	Possibly infinite
Concurrency	Synchronous	Synchronous	Asynchronous

Case Study 1: TCAS II

17

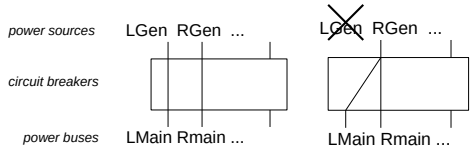
- Traffic Alert and Collision Avoidance System
  - Reduce mid-air collisions: warn pilots of traffic and issue resolution advisories
  - "One of the most complex systems on commercial aircraft."
- 400-page specification reverse-engineered from pseudo-code: written in RSML by Leveson et al., based on statecharts





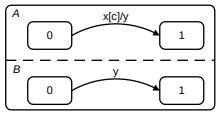
Case Study 2: EPD System

- Electrical Power Distribution system used on Boeing 777
- Distribute power from sources to buses via circuit breakers
  - Tolerate failures in power sources and circuit breakers
- Prototype specification in statecharts
- Analysis joint with Jones and Warner of Boeing



Translation to SMV

21




```
VAR
  A: {0,1};
  x: boolean;
  y: boolean;
ASSIGN
  init (A):= 0;
  next (A):= case
    A=0 & x & c : 1;
    1 : A;
    esac;
  ...
```


Deterministic or not?

22

```
V_254a := MS = TA_RA | MS = TA_only | MS =3 | MS = 4 |
                       MS = 5 | MS = 6 | MS = 7;
V_254b := ASL = 2 | ASL = 3 | ASL = 4 | ASL = 5 |
          ASL = 6 | ASL = 7;
T_254 := (ASL = 2 & V_254a) | (ASL = 2 & MS = TA_only) |
         (V_254b & LG = 2 & V524a);
V_257a := LG = 5 | LG = 6 | LG = 7 | LG = none;
V_257b := MS = TA_RA | MS = 5 || MS = 6 | MS = 7;
V_257c := MS = TA_RA | MS = TA_only | MS = 3 | MS = 4 |
          MS = 5 | MS = 6 | MS = 7;
V_257d := ASL = 5 | ASL = 6 | ASL = 7;
T_257 := (ASL = 5 | V_257a | V_257b) |
         (ASL = 5 & MS = TA_only) |
         (ASL = 5& LG = 2 & V_257c) |
         (V_257d & LG = 5 & V_257b) |
         (V_257d & V_257a & MS = 5);
```


Analyses and Results

23

- Used and modified SMV [McMillan 93]

	TCAS II	EPD System
State space	230 bits, 10^{60} states	90 bits, 10^{27} states
Prior verification	inspection, static analysis	simulation
Problems we found	inconsistent outputs, safety violations, etc.	violations of fault tolerance

Some Formulae Checked

24

- TCAS II
  - Descent inhibition: AG (Alt < 1000 → ¬Descend)
  - Output agreement: AG ¬(GoalRate ≥ 0 ∧ Descend)
- EPD system
  - AG (NoFailures → (LMain ∧ RMain ∧ LBackup ∧ RBackup))
  - AG (AtMostOneFailure → (LMain ∧ RMain))
  - AG (AtMostTwoFailures → (LBackup ∨ RBackup))
- Where do these come from?

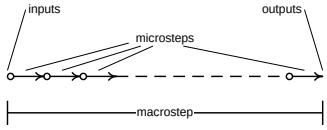
One example (EPD) counterexample

25

- A single failure can cause a bus to lose power
  - Power-up sequence; normal operation
  - A circuit breaker fails
  - Other circuit breakers reconfigured to maintain power
  - User changes some inputs
  - The first circuit breaker recovers
  - User turns off a generator
  - A bus loses power

This error does not exist in onboard system

Synchrony hypothesis

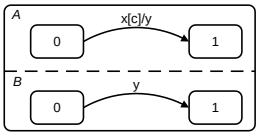


- No new inputs within macrostep
- Macrostep encoded as a sequence of transitions
- Statecharts, Esterel [Berry & Gonthier 92], Lustre [Halbwachs et al. 92], etc.

Mutual Exclusion of Transitions

27

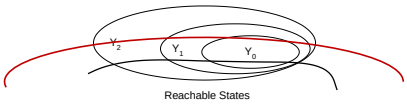
- Many "concurrent" transitions are sequential
  - Determine using static analysis
- Use this to prune backward search



- Generally unclear whether forward or backward search is better
- Forward search
  - Often good for low-level hardware.
  - But always bad for us; large BDDs

A Disadvantage of Backward Search

- Visiting unreachable states



- Find invariants that are small as BDDs and effective in reducing main BDD sizes
- Often from the domain itself

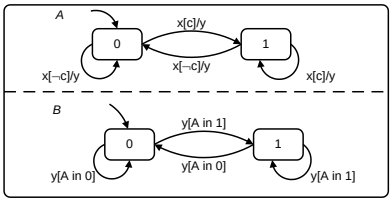
Initial EPD Analyses Failed

29

- Even though it has fewer states than TCAS II
- Main difference in synchronization
  - TCAS used "oblivious" synchronization –every external event took the same number of state transitions
  - EPD used "non-oblivious" synchronization
- Solution: convert non-oblivious to oblivious and maintain (most) properties

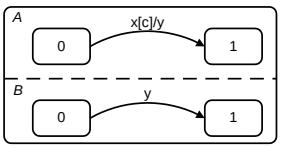
	TCAS II	EPD System
State space	230 bits, 10^{60} states	90 bits, 10^{27} states

Oblivious Synchronization (TCAS II)



- **y** signals completion of machine A
  - Macrostep length: 2
  - **x → y → stable**

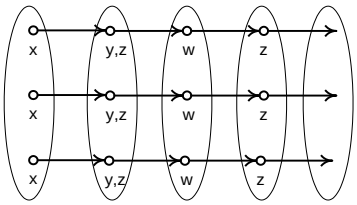
Non-Oblivious Synchronization (EPD)



- **y** signals state change in machine A
- Macrostep length: 1 or 2
  - **x → y → stable**
  - **x → stable**

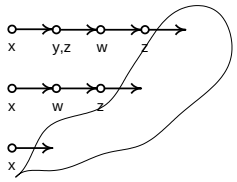
Oblivious Synchronization

- Backward search yields small BDDs
- Event sequence always identical: every macro-step has the same length

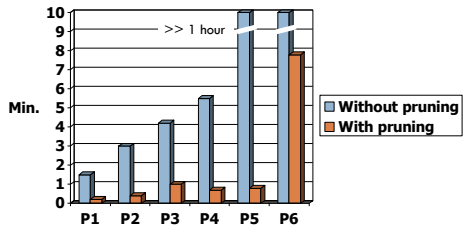


Non-Oblivious Synchronization

- Backward search leads to (much) larger BDDs
- Basic solution: Automatic semantics-preserving transformation
  - Add stuttering states to make every macro-step of equal length
  - Preserve most properties, e.g., invariants and eventualities. [Lamport 83, Browne et al. 89]
- Increase # states and # state variables
- Increase # iterations to reach fixed points



Overall Effects on TCAS II



Some Lessons Learned

35

- Focus on restricted models that people care about
- Exploit high-level knowledge to improve analysis
  - Synchronization, environmental assumptions, etc.
  - In addition to low-level BDD tricks
- Combine static analysis and symbolic model checking
- Help understand system behaviors
  - In addition to verification/falsification

SLAM and SDV

36

- Technically interesting: how to effectively use model checking to establish useful properties of an important class of C programs
- Sociologically interesting: what it takes to transfer technology – it's an ecosystem of sorts
  - A much broader view of the ecosystem of creating major high-tech industries can be found in Innovation in Information Technology, The National Academies Press, 2003 (http://www.nap.edu/catalog.php?record_id=10795)

Basic story

37

- Third-party device drivers caused a disproportionate number of "blue screens" for Windows – costly in time and effort, as well as in reputation for Microsoft
- Are major causes of the device driver errors checkable automatically even though arbitrary C code isn't fully checkable: infinite paths, aliasing, …
- Found an abstraction of drivers and properties to check that allowed a combination of model checking and symbolic execution to identify major classes of errors in practice

Evaluation and examples

38

- Applied SDV to 126 WDM drivers (storage, USB, 1394-interface, mouse, keyboard, …)
  - Well tested, code reviewed by experts, in use for years, 26 were open source
  - 48 to 130,000 LOC, average of 12KLOC
- An initial study reported 206 defects: investigation of 65, including working with the code owners, classified 53 as true errors and 12 as false errors

- In a path a driver marked an I/O request packet pending with a kernel API, but didn't mark it in a related data structure
- A driver's dispatch routine returned `STATUS PENDING` but declared the I/O request packet as completed with `IoCompleteRequest`
- A driver called `IoStartNextPacket` from within `StartIo`, which could lead to recursion exceeding the stack space
- Early in the execution a device driver called an API that can raise the interrupt request level of the thread, and then (much later) called another kernel API that should not be called when the interrupt request level is raised (because it touches paged data)
- `IoCompleteRequest` was called while holding a spinlock, which could cause deadlock
- …

Abstraction for SDV

39

- Focused goal: check that device drivers make proper use of the driver API – not to check that the drivers do the right thing (or even anything useful)
- Automatically abstracts the C code of a device driver
  - Guarantees that any API usage rule violation in the original code also appears in the abstraction
- Then check the abstraction – which is smaller and more focused than the original code

Boolean predicate abstraction

40

- Translate to a representation that has all of C's control flow constructs but only boolean variables that in turn track the state of relevant boolean expressions in the C code
- These relevant expressions are selected based on predefined API usage rules constructed for device drivers
- Consider a driver with 100 KLOC and complicated data structures and checking for an API usage rule intended to verify proper usage of a specific spinlock
- Abstract to a program that tracks, at each line of code, the state of the spin lock as either locked or unlocked
- This leads to a boolean program with around 200,000 states, which is manageable by model checking

API usage rules

41

- A state machine with two components
  - a static set of state variables (a C `struct`)
  - a set of events and state transitions
- On right: rule for the proper usage of spin locks
  - one state variable
  - two events on which state transitions happen – returns of calls to `acquire` and `release`

```
state { enum {Unlocked, Locked}
        state = Unlocked;
} watch KeAcquireSpinLock.$1;
KeAcquireSpinLock.return [guard $1] {
  if ( state == Locked ) {
    error;
  } else {
    state = Locked;
  }
}
KeReleaseSpinLock.return [guard $1] {
  if ( state == Unlocked ) {
    error;
  } else {
    state = Unlocked;
  }
}
```

Overall process (beyond abstraction)

42

- Given a boolean program with an error state, check whether or not the error state is reachable – BDD-based model-checking
- If the checker identifies an error path that is a feasible execution path in the original C, then report an error
- If the path is not feasible then refine the boolean program to eliminate the false path
- Use symbolic execution and a theorem prover to find a set of predicates that eliminates the false error path

Overview of process

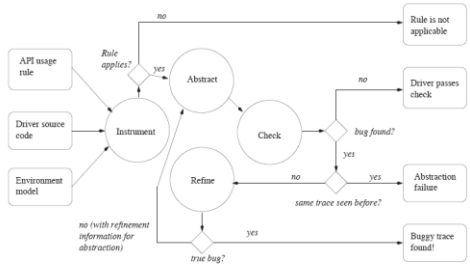


Figure from "Thorough Static Analysis of Device Drivers" (Ball et al. EuroSys 06)

A hot topic: many efforts including…

44

- BLAST: Berkeley Lazy Abstraction Software Verification Tool (http://mtc.epfl.ch/software-tools/blast/)
  - "The goal … is to be able to check that software satisfies behavioral properties of the interfaces it uses. [It] uses counterexample-driven automatic abstraction refinement to construct an abstract model which is model checked for safety properties. The abstraction is constructed on-the-fly, and only to the required precision."
- VeriSoft (http://cm.bell-labs.com/who/god/verisoft/)
  - "… automatically searches for coordination problems (deadlocks, etc.) and assertion violations in a software system by generating, controlling, and observing the possible executions and interactions of all its components."
- Java PathFinder (http://javapathfinder.sourceforge.net/)
  - "[It] is a Java Virtual Machine that is used as an explicit state software model checker, systematically exploring all potential execution paths of a program to find violations of properties like deadlocks or unhandled exceptions. … [A] model checker has to employ flexible heuristics and state abstractions. JPF is unique in terms of its configurability and extensibility, and hence is a good platform to explore new ways to improve scalability."

Coming soon…

45

- Model checking has really taken off in some dimensions
- In particular, there has been a lot of work connecting automated test generation and model checking (along with symbolic evaluation, etc.)
- We'll come back to this after we do an overview of some key software testing basics