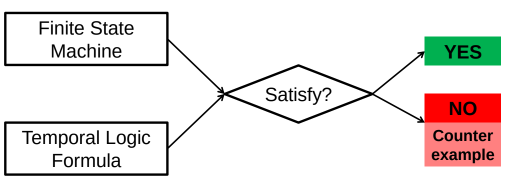
CSE P503: Principles of Software Engineering

David Notkin
Spring 2009

Tonight's agenda

- Model checking motivation, technical introduction, checking specifications
- Interlude: discussion about "When Should a Process Be Art" by Hall and Johnson (March 2009 *Harvard Business Review*)
- SLAM/SDV
- May 21st – what to do?
- Bounded model checking: very brief intro to alloy
- One-minute paper (email to me by close of business tomorrow): Key point? Open question? Mid-course correction?

Model checking



- What are finite state machines? Temporal logic formulae? What can they represent? What does "satisfy" mean? How does "satisfy" work? Why should we care?
- Tonight: some low-level details, jumping to high-level approaches – fill in the glue if you want on your own (I can help)

ACM 2007 Turing Award Citation

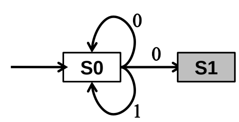
In 1981, Edmund M. Clarke and E. Allen Emerson, working in the USA, and Joseph Sifakis working independently in France, authored seminal papers that founded what has become the highly successful field of Model Checking. This verification technology provides an algorithmic means of determining whether an abstract model--representing, for example, a hardware or software design--satisfies a formal specification expressed as a temporal logic formula. Moreover, if the property does not hold, the method identifies a counterexample execution that shows the source of the problem. The progression of Model Checking to the point where it can be successfully used for complex systems has required the development of sophisticated means of coping with what is known as the state explosion problem. Great strides have been made on this problem over the past 27 years by what is now a very large international research community. As a result many major hardware and software companies are now using Model Checking in practice. Examples of its use include the verification of VLSI circuits, communication protocols, software device drivers, real-time embedded systems, and security algorithms. …

Finite state machines (FSMs)

Mathematically: a 5-tuple

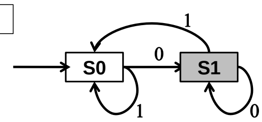
1. Finite non-empty alphabet
2. Finite non-empty set of states
3. A single start state
4. A state-transition mapping that takes a state and a symbol and returns
   – a new state (for deterministic FSMs) or
   – a new set of states (for non-deterministic FSMs)
5. A possibly empty set of final states

Trivial example



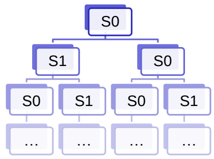
What do they do?
(Both do the same thing)

What's the difference?



A computation tree

- Represent all possible paths with a computation tree – even when infinite, structure is constrained because of finite states
- Model checking answers questions about this tree structure
- Kinds of queries
  - Does every accepting input include a **0**? A **1**?
  - Does any accepting input include a **0**? A **1**?
  - Does every accepting input that has a **1** have a **1** in the remaining input?
  - …



- The computation tree is generated from the state machine
- The temporal logic formula queries the computation tree

Two Approaches to Model Checking

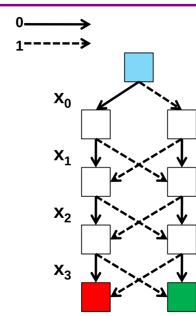
- Explicit – represent all states
  - Use conventional state-space search
  - Reduce state space by folding equivalent states together
- Symbolic – represent sets of states using boolean formulae
  - Reduce huge state spaces by considering large sets of states simultaneously
  - Convert state machines, logic formulae, etc. to boolean representations
  - Perform state space exploration using boolean operators to perform set operations

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

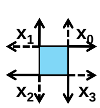
Binary Decision Diagrams (BDDs)

- 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: $\mathbf{x_0}$ and $\mathbf{\neg x_0}$
- On the right is an example of a BDD for odd (even) parity of 4-bit numbers



What would odd parity look like if…

- …the bits in the BDD were ordered in reverse? $\mathbf{x_3x_2x_1x_0}$
- …the bits were unordered? (Not a BDD)



Groups of 3-4

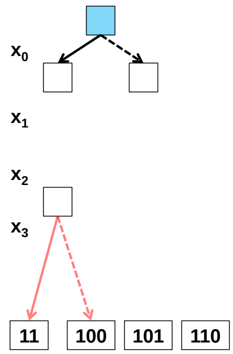
Does the bit order affect the size?

- Not for parity…

Groups of 3-4
count off as A, B, C, D

- Group A: $\mathbf{x_0x_1x_2x_3}$ – compute $\mathbf{x_1x_0 + x_3x_2}$
- Group B: $\mathbf{x_0x_1x_2x_3}$ – compute $\mathbf{x_2x_0 + x_3x_1}$
- Group C: $\mathbf{x_0x_1x_2x_3}$ – compute $\mathbf{x_1x_0 * x_3x_2}$
- Group D: $\mathbf{x_0x_1x_2x_3}$ – compute $\mathbf{x_2x_0 * x_3x_1}$

x_3x_1 / x_2x_0	00	01	10	11
00	00	01	10	11
01	01	10	11	100
10	10	11	100	101
11	11	100	101	110



Efficiency

- 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

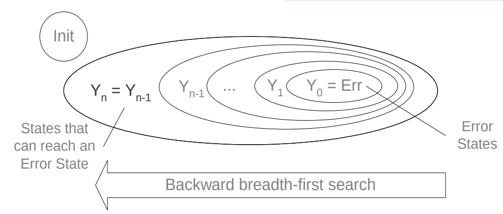
Burch et al.
Couder et al.

- Define boolean state variables
  - e.g., define **X** = $\mathbf{x_{n-1}}, \mathbf{x_{n-2}}, \ldots, \mathbf{x_0}$ 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')**

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

Check software specification

- 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

- "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

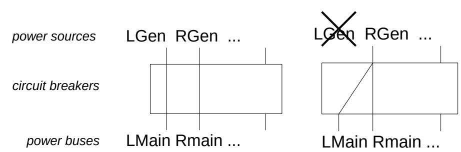
- Most costly bugs in specification
- Use analyzable formal specification
  - State-machine specifications
  - Intuitive to domain experts like aircraft engineers
  - Statecharts [Harel 87], RSML [Leveson et al. 94], SCR [Parnas et al.], etc.

Case Study 1: TCAS II

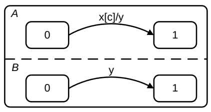
- Traffic Alert and Collision Avoidance System
  - Reduce mid-air collisions: warn pilots of traffic and issue resolution advisories
  - "One of the most complex systems on commercial aircraft."
- 400-page specification reverse-engineered from pseudo-code: written in RSML by Leveson et al., based on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




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


Analyses and Results

- 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

- 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

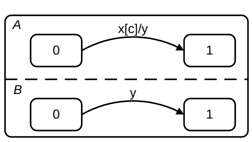
A single failure can cause a bus to lose power

1. Power-up sequence; normal operation
2. A circuit breaker fails
3. Other circuit breakers reconfigured to maintain power
4. User changes some inputs
5. The first circuit breaker recovers
6. User turns off a generator
7. A bus loses power

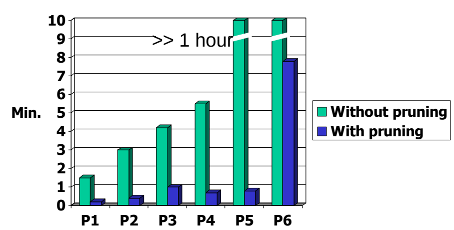
This error does not exist in onboard system

Mutual Exclusion of Transitions

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



Overall Effects on TCAS II



Initial EPD Analyses Failed

- 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

Some Lessons Learned

- 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

Interlude

Recent email from a colleague

"Hall and Johnson, in March 09 Harvard Business Review, write an interesting article that (1) questions undue commitment to variance-reducing 'scientific"' processes and (2) insists on the importance of 'artistic' processes – process definitions that expressly provide 'artists' with room to use judgment.

"They list 'software development' as a domain in which artistic processes are important. They see artists, in business, as typically being *supported* by surrounding scientific processes. For example, the neurosurgeon is an artist, but the nurses and prep folks will do best following strict protocols. [It's] a pretty interesting article, with implications for how we think about the role of process formalization and enforcement in software development."

Follow-up email from colleagues

"Seems like this all reduces to economics - taking risk in order to achieve a benefit. Reducing variance is valuable when the restriction is eliminating low-value cases off a known high-value path. Artistic variance is valuable when no given path is known to be high-value or maybe the space of options isn't even known – it needs to be discovered.

"What I find interesting is the idea of low-variance support for high-variance activities. Still, it's about value in a high-dimensional space: the neurosurgeon isn't going to learn much by exploring the part of the solution space that involves dirty scalpels. So the surgeon's exploration of the solution space needs to be constrained to dimensions of probable value. This article seems to propose a rationale for decomposing the problem and the team into low- and high-variance roles. Cognitively, this seems to make some sense."

And more... "When to go 'artistic'"

- "when one suspects a higher value alternative might exist - better than the known high-value one
- when all the knowledge about the values of paths originate from a homogeneous set of sources
- when an assumption behind all the analytics seem deserving of mistrust
- when the environment … seems poised to change but has not yet
- when all the known value paths have been explored too many times
- when it's hard to know the value or when you suspect you aren't computing value correctly (enough)"
- … <some bullets elided>

"That is, I believe there are times to go artistic even when there might be suitably high value paths in front of you."

So, whaddya think?

SLAM and SDV

- 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

- 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
- Oh, and tech transfer – beyond the scope of lecture (but not of the wiki)

Evaluation and examples

- 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

- 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

- 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

- 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)

- 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

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

Overview of process

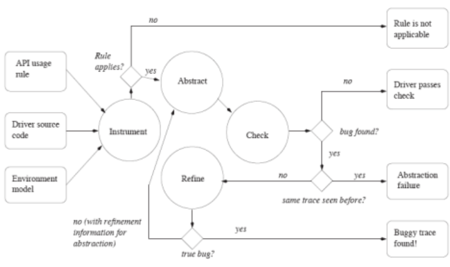


Figure 5: SDV's analysis engine's architecture

A hot topic: *many* efforts including…

- 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."

Thursday May 21?

- I'm in Vancouver at the 2009 International Conference on Software Engineering (http://www.cs.uoregon.edu/events/icse2009/home/)
  - Co-chairing the Doctoral Symposium
  - New Ideas and Emerging Research paper/poster
    - M. Nita and D. Notkin. White-Box Approaches for Improved Testing and Analysis of Configurable Software Systems
  - Research paper
    - M. Kim and D. Notkin. Discovering and Representing Systematic Code Changes
- So, what do we do?

Possibilities include…

- Rescheduling
- Cancelling
- Guest lecture (hard, since many are also in Vancouver)
  - I have an excellent 1.5 hour Michael Jackson talk available, though
- Class presentations on your state-of-the-art research papers
- Other ideas?

Bounded model checking

- The TCAS/EPD work avoided most abstraction by starting with finite state specifications
- SLAM/SDV and other model checkers that work on source code must abstract the program to get to a finite state model
- Bounded model checking instead accepts an infinite state machine along with a formula to check – and then truncates the search space
  - Guaranteed to find errors within the bound
  - Errors outside the bound are not found
  - Small scope hypothesis: a high proportion of bugs can be found by testing a program for all test inputs within some small scope

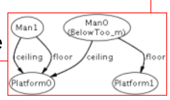
Alloy: Daniel Jackson @ MIT

- A bounded model checker/tool
- “Electrifies” formal descriptions
- Example models include caches, file stores, security constraints (JVM), file system synchronization, railway safety, peer-to-peer protocols, etc.

Silly example

```
sig Platform {}        there are “Platform” things
sig Man {ceiling, floor: Platform}
        each Man has a ceiling and a floor Platform
pred Above [m, n: Man] {m.floor = n.ceiling}
        Man m is “above” Man n if m's floor is n's   ceiling
fact {all m: Man | some n: Man | Above[n,m] }
        "One Man's Ceiling Is Another Man's Floor"
assert BelowToo {
   all m: Man | some n: Man | Above [m,n] }
        "One Man's Floor Is Another Man's Ceiling"?
check BelowToo for 2
        counterexample with 2 or less platforms and men?
```

Alloy finds a counterexample



One-minute paper

- Add directly to the wiki or email to me by close of business tomorrow (or fill in a note card tonight if you want to be anonymous)
  - Key point?
  - Open question?
  - Mid-course correction?
- Yes, we’ll do these most weeks