Intro to Model-Checking

Robert Keller
CS 156 Spring 2011
System Design Steps

• Establish *requirements*.

• Create *specification* for a system that meets the requirements.

• Design and implement ("code") the system.

• Establish that the implementation correctly obeys the specification.
Software Errors can be Expensive

- European Ariane 5 spacecraft ($.5 Billion loss)
- NASA Mars Climate Orbiter ($165 Million loss)
- Canadian Therac 25 (5 deaths)
Relevance to HMC?

• It can’t happen here?

• How many Clinic projects have we done with
  • Electus Technology?
  • Medtronic/MiniMed?
  • the FAA?
Who cares about this?

- Intel
- NASA
- IBM
- Rockwell
- TPC
- DOE
- Voters

Systems in all areas are increasingly software-centric.
The application PowerPoint quit unexpectedly.

Mac OS X and other applications are not affected.

Click Reopen to open the application again. Click Report to see more details or send a report to Apple.
Approaches to System Correctness

• Testing/Simulation
• Formal verification
• Model-checking
Testing/Simulation

- Can only establish existence, not absence, of errors, unless system can be tested exhaustively.

- Laborious and time-consuming.
Formal Verification

- Requires sophisticated axiomatic framework.
- Requires powerful theorem-proving system.
- Users writing specifications must be well-versed in logic.
- Requires creativity, in the form of intermediate correctness specifications.
Model Checking

+ More thorough than simulation.

+ Does not require axiomatic framework.

+ Doesn’t require as much logic as verification.
  
- System must be *finite-state*, however
+ certain types of errors can be detected even in infinite-state systems.

+ Software tools keep getting better.
Finite-state systems of interest

- Communication protocols
- Telephones
- Alarm systems
- File-opening protocols in software
- GUI protocols
- Business protocols
Model-Checking Steps

- Implementation
  - Create model
  - Check model
- Model
- Results
Automation Possibilities

• **Create Model:**
  • Sometimes manual
  • Can be automated, depending on implementation language (NASA Ames: Java Pathfinder)
  • Semi-automated via appropriate tools

• **Check Model:**
  • Usually automated
Modeling Language

• A language of some kind is needed as input for any automated task.

• Various modeling languages and companion checkers exist:
  • SPIN (Holzmann: Bell Labs, now JPL)
  • Uppaal (Uppsala, Sweden + Aalborg, Norway)
  • Murϕ (Stanford)
Uppaal Modeling Language

- Based on “timed automata” (finite-state automata + timing info)

- Model representations:
  - Graphical
  - Textual
  - XML
Basic Single-Resource Example

• Users make use of a single shared resource, call it R.

• Only one user can use R at a time.

• Protocol:
  • Users must request the resource to use it.
  • Users must release the resource when done.
User Cycle

start

require_R!

request_granted

do_stuff()

release_R!
Resource-Manager Cycle
Rendezvous Paradigm
Uppaal Simulation View
Uppaal Sequence Diagram View
2 Users, 1 Resource
2 Users, 1 Resource Sequence

[Diagram showing a sequence of states and transitions for two users and a manager, with states such as "start", "request Granted", "R_free", etc.]
2 Users, 1 Resource
Different Sequences

Diagram:

Left:
- User0: start → request_granted → - → start
- User1: start → request_granted → R_in_use → -
- Manager: R_free → require_R → request_granted → R_free

Right:
- User0: start → request_granted → R_in_use → -
- User1: start → request_granted → R_in_use → -
- Manager: R_free → require_R → request_granted → R_in_use → release_R → R_free
General Global State View

Initial state
Path Analyzed by One Simulation

Initial state
Paths Analyzed by One Model-Check
e.g. Steiner, et al., 2004

“The resulting models have billions or even trillions of reachable states, yet the symbolic model checker of SAL is able to examine these in a few tens of minutes (for billions of states) or hours (for trillions).”
Expressing Checkable Properties

- Temporal Logic (TL) is an extension of predicate logic (PL) expressing properties of behavioral sequences.

- PL concerned with logical properties of a single state:
  \[ x > y + 5 \]

- TL concerned with properties of a sequence of states:
  “after \( y > 0 \), \( x > y + 5 \) is possible”
Examples of Temporal Operators

- $\square P$  
  "Henceforth" $P$

- $\Diamond P$  
  "Eventually" $P$

- $\square \Diamond P$  
  Eventually henceforth $P$

- $\square \Diamond P$  
  Henceforth eventually $P$

- $\square \Diamond \Diamond P$  
  Eventually henceforth eventually $P$

etc.
General Global State View

Initial state
Red (Henceforth Red)
Red (Eventually Red)
Red
Red
Temporal Operators in Uppaal

- Two levels of nesting only
- $A[\ ] P$ means $P$ is true in all reachable states
Correctness Property 1

- Only one user should use the resource at a time.

- In Uppaal's TL specification language:

  $A[] \neg (\text{User0.request\_granted} \land \text{User1.request\_granted})$

  $\neg$ is "not"     \quad $\land$ is "and"
Only one user using at a time

User0
User1
Manager

start
start
start
start
start
start

R_free
R_free
R_free
R_free
R_free
R_free

start
start
start
start
start
start

R_in_use
R_in_use
R_in_use
R_in_use
R_in_use
R_in_use

start
start
start
start
start
start

R_in_use
R_in_use
R_in_use
R_in_use
R_in_use
R_in_use

start
start
start
start
start
start

R_free
R_free
R_free
R_free
R_free
R_free

granted
granted
granted
granted
granted
granted

_user
_user
_user
_user
_user
_user


Uppaal Verifying Property 1

Overview

A[] ! (User0.request_granted && User1.request_granted)

Query

A[] ! (User0.request_granted && User1.request_granted)

Comment

The request can be granted to only one user at a time.

Status

A[] ! (User0.request_granted && User1.request_granted)

Property is satisfied.
Correctness Property 2

- No deadlock
- In Uppaal’s TL specification language:

  \[ A[] !\text{deadlock} \]

Here deadlock means "there is no way of leaving the state".
Uppaal Verifying Property 2

Overview

A[] ! (User0.request_granted && User1.request_granted)

A[] !deadlock

Query

A[] !deadlock

Comment

There is no deadlock.

Status

Property is satisfied.
A[] !deadlock
Property is satisfied.
More on Uppaal Specification

- $E<> P$ means there exists a path in which $P$ becomes true.

- $E<> \text{User0.granted}$
- $E<> \text{User1.granted}$
More Exacting Requirement

- $P \rightarrow Q$ means whenever $P$ is true, there is a path in which $Q$ becomes true subsequently.

- User0.start $\rightarrow$ User0.granted
- User1.start $\rightarrow$ User1.granted
Uppaal Proving

Overview

A[] ! (User0.request_granted & User1.request_granted)
A[] !deadlock
E<> User0.request_granted
E<> User1.request_granted
User0.start --> User0.request_granted
User1.start --> User1.request_granted

Query

User1.start --> User1.request_granted

Comment

When User1 wants to use the resource, eventually it will be granted.

Status

Property is satisfied.
User1.start --> User1.request_granted
Property is satisfied.
What can’t be proved in this model?

\[ A \leftrightarrow \text{User0.request\_granted} \]

(From every state, eventually the user’s request will be granted.)

Two reasons:

- A user is not forced to move.
- One user could be “starved” by the other.
Forcing Motion

• States can be marked “urgent”, meaning cannot dwell in this state.

• The system must move from that state if possible.
User with all states urgent
Uppaal Showing Property Failure

Overview

\[ A[]! (\text{User0.request\_granted } \&\& \text{User1.request\_granted}) \]

\[ A[]! \text{deadlock} \]

\[ E<> \text{User0.request\_granted} \]

\[ E<> \text{User1.request\_granted} \]

\[ \text{User0.start} \rightarrow \text{User0.request\_granted} \]

\[ \text{User1.start} \rightarrow \text{User1.request\_granted} \]

\[ A<> \text{User0.request\_granted} \]

Query

\[ A<> \text{User0.request\_granted} \]

Comment

When User1 wants to use the resource, eventually it will be granted.

Status

\[ A[]! \text{deadlock} \]

\text{Property is satisfied.}

\[ E<> \text{User0.request\_granted} \]

\text{Property is satisfied.}

\[ E<> \text{User1.request\_granted} \]

\text{Property is satisfied.}

\[ \text{User0.start} \rightarrow \text{User0.request\_granted} \]

\text{Property is satisfied.}

\[ \text{User1.start} \rightarrow \text{User1.request\_granted} \]

\text{Property is satisfied.}

\[ A<> \text{User0.request\_granted} \]

\text{Property is not satisfied.}
Certain Types of Failures (Deadlock, starvation, ...) can be exhibited explicitly
Starvation can be avoided

• with a more complicated system structure

• e.g. conscience: after using a resource, a user must give the other user a chance.
Deadlock Example

• 2 resources, 2 users
• Uppaal constructs the shortest sequence to deadlock state
Deadlock Example

![Deadlock Diagram]
Shortest Sequence to Deadlock
Constructed by Uppaal

Overview

A[] ! (User0.requests Granted && User1.requests Granted)
A[] ! deadlock
E<> User0.requests Granted
E<> User1.requests Granted
User0.start --> User0.requests Granted
User1.start --> User1.requests Granted
A<> User0.requests Granted
How to establish properties involving infinite sequences?

• Certain temporal properties are exemplified by “Buchi automata”:
  • Finite-state automata
  • A special notion of acceptance for infinite sequences of states of the model.

• Such automata can be “crossed” with a system and the result analyzed by state reachability to establish whether or not the property holds.
Properties of Infinite Sequences

• red, red, red, red, ... ■ red

• !red, !red, ..., red, red, red, ... ◆ ■ red

• !red, red, !red, red, !red, red, ... ■ ◆ red
Buchi Automaton for Red

An infinite sequence is accepted iff the red state is entered infinitely-often.
Buchi Automaton for ♦ ■ Red

true

red (non-deterministic)

red (!red)
Buchi Automaton for \( \blacklozenge \) \( \blacklozenge \) Red

An infinite sequence is accepted iff the red state is entered infinitely-often.
Real-Timing

• Uppaal is somewhat novel in providing for real-time.

• Based on timed automata.

• Clock variables can be introduced.

• Time is continuous!

• Time inequalities analyzed symbolically.
Real-Time Examples

• Start B no sooner than 10 seconds after A starts
Real-Time Examples

- Start B no later than 10 seconds after A starts

The invariant is a form of implicit control.
Techniques Combined

• Start B between 9 and 10 seconds after A starts
Downside on Methodology: Is there a better way?

- Implementation
  - Create model
    - slow, expensive
  - Check model
    - Model
    - possible transcription errors
    - may be discouraging
- Results
  - somewhat laborious
Cleaner Way

- Model
  - Check model
  - Generate code
- Results
- Implementation
Code Generation Efforts

- Larsen, Yi, et al. generated C code from Uppaal, e.g. for real-time control of Aibo robot and a “production cell” model.
Complementary Tools & Related Research Areas

- Static analysis tools analyze code symbolically, stopping short of being full verification.

- Theorem provers can be combined with M-C.

- UML (if formal semantics can be devised).
Conclusions

• Illustrated the idea of model-checking.
• Demonstrated temporal logic for specifications.
• Exemplified with Uppaal.
• Contrasted to other approaches.
• Discussed representation of timing.
• Future approaches likely to invert the order (model, then code).
References

• Uppaal: Kim Larsen, et al.,
  http://www.uppaal.com

• SPIN: Gerard Holzmann,
  http://www.spinroot.com/

• CMU: Ed Clarke,
  http://www.cs.cmu.edu/~Emodelcheck/