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 ($0.5 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

1. Create model
2. Check model
3. Implementation
4. 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

Diagram:

- **User1**
  - Start
  - Require_R!
  - Request Granted
  - Do stuff()
  - Release_R!

- **Manager**
  - R_free
  - Require_R?
  - Release_R?
  - R_in_use
Uppaal Sequence Diagram View
2 Users, 1 Resource
2 Users, 1 Resource Sequence
2 Users, 1 Resource
Different Sequences
Global State View (conceptual)

User0
User1
Manager

start
R_free

start
R_in_use

start
R_in_use

start
R_in_use

start
R_in_use

start
R_free

start
R_in_use

start
R_in_use

start
R_in_use

start
R_free

start
R_in_use

start
R_in_use

start
R_free

User0

User1
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” $P$
- $\lozenge P$  “Eventually” $P$

- $\lozenge \square P$  Eventually henceforth $P$
- $\square \lozenge P$  Henceforth eventually $P$
- $\lozenge \lozenge \lozenge P$  Eventually henceforth eventually $P$
  etc.
General Global State View

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

- Two levels of nesting only

- $A[ \ ] \varphi$ means $\varphi$ 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:

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

  ! is “not” \quad \land is “and”
Only one user using at a time

User0
User1
Manager

start
start
start
start
start
start

granted
start
start
start
start

R_free
R_free
R_in_use
R_in_use
R_in_use
R_in_use

User0
User1

start
start
start
start

R_free
R_free
R_free
R_free

R_in_use
R_in_use
R_in_use
R_in_use

granted
granted
granted
granted

R_in_use
R_in_use
R_in_use
R_in_use

R_free
R_free
R_free
R_free

Uppaal Verifying Property 1

Overview

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

Query

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

Comment

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

Status

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

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.requestGranted && User1.requestGranted)

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

\[ A[] \land (User0.request\_granted \land User1.request\_granted) \]
\[ A[] \land \neg \text{deadlock} \]
\[ E\leftrightarrow User0.request\_granted \]
\[ E\leftrightarrow User1.request\_granted \]
\[ User0.\text{start} \rightarrow User0.request\_granted \]
\[ User1.\text{start} \rightarrow User1.request\_granted \]

Query

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

Comment

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

Status

Property is satisfied.
User1.\text{start} \rightarrow User1.request\_granted
Property is satisfied.
What can’t be proved in this model?

A<> 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

<table>
<thead>
<tr>
<th>Property</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>A[] (! (User0.request_granted \&amp;\&amp; User1.request_granted))</code></td>
</tr>
<tr>
<td><code>A[] ! deadlock</code></td>
</tr>
<tr>
<td><code>E&lt;&gt; User0.request_granted</code></td>
</tr>
<tr>
<td><code>E&lt;&gt; User1.request_granted</code></td>
</tr>
<tr>
<td><code>User0.start --&gt; User0.request_granted</code></td>
</tr>
<tr>
<td><code>User1.start --&gt; User1.request_granted</code></td>
</tr>
<tr>
<td><code>A&lt;&gt; User0.request_granted</code></td>
</tr>
</tbody>
</table>

### Query

<table>
<thead>
<tr>
<th>Property</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>A&lt;&gt; User0.request_granted</code></td>
</tr>
</tbody>
</table>

### Comment

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

### Status

<table>
<thead>
<tr>
<th>Property</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>A[] ! deadlock</code></td>
</tr>
<tr>
<td><code>Property is satisfied.</code></td>
</tr>
<tr>
<td><code>E&lt;&gt; User0.request_granted</code></td>
</tr>
<tr>
<td><code>Property is satisfied.</code></td>
</tr>
<tr>
<td><code>E&lt;&gt; User1.request_granted</code></td>
</tr>
<tr>
<td><code>Property is satisfied.</code></td>
</tr>
<tr>
<td><code>User0.start --&gt; User0.request_granted</code></td>
</tr>
<tr>
<td><code>Property is satisfied.</code></td>
</tr>
<tr>
<td><code>User1.start --&gt; User1.request_granted</code></td>
</tr>
<tr>
<td><code>Property is satisfied.</code></td>
</tr>
<tr>
<td><code>A&lt;&gt; User0.request_granted</code></td>
</tr>
<tr>
<td><code>Property is not satisfied.</code></td>
</tr>
</tbody>
</table>
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

[Diagram of two user processes showing mutual waiting for resources R and S]
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 Automatonon 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)
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

```
start_A!
clk = 0
clk >= 10
start_B!
```

- **clock set**

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

Model

Check model

Results

Generate code

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.,
  [www.uppaal.com](http://www.uppaal.com)

- **SPIN**: Gerard Holzmann,
  [www.spinroot.com](http://www.spinroot.com)

- **CMU**: Ed Clarke,