Verification of Intelligent Controllers using Model Checking

Charles Pecheur, UC Louvain

(formerly RIACS / NASA Ames)
Embedded Controllers

• Everywhere
  – more and more so
• Dependability is critical
  – human risks
  – material risks
  – economic risks
• Logic (vs. physical) part is increasing
Process Control

- Partially observable process (hidden state $x$, estimated by $\hat{x}$)

- **observability**: infer $x$ from $y$ (and $u$)

- **commandability**: impose $x$ through $u$

- **control theory**: $x =$ physical quantities, differentiable
  $\rightarrow$ linear models, PDI controllers

- **logic processes**: $x =$ states, modes, failures, discrete
  $\rightarrow$ state machines, programmable automata
Verification of Control Systems

• **Monitors and commands** a process
  – in particular, **failure diagnosis** and **recovery**

• **Complex**
  – multiple controllers, asynchronism, coupling
  – race conditions, feature interaction

• **Software**
  – powerful and flexible but not linear, not continuous

• **How to Validate?**
  – including "**diagnosability**" and "**recoverability**" from failures?
## Reliability: Hardware vs Software

<table>
<thead>
<tr>
<th>Hardware</th>
<th>Software</th>
</tr>
</thead>
<tbody>
<tr>
<td>physical variability</td>
<td>identical copies</td>
</tr>
<tr>
<td>failures due to wear, environment</td>
<td>design flaws</td>
</tr>
<tr>
<td>reliability through redundancy</td>
<td>copies of the same code have the same bugs</td>
</tr>
<tr>
<td>reliability varies in time</td>
<td>reliability depends on execution, not on time</td>
</tr>
<tr>
<td>progressive degradation</td>
<td>abrupt degradation</td>
</tr>
</tbody>
</table>
Autonomy (at NASA)

Autonomous spacecraft = on-board intelligence (AI)

- **Goal:** Unattended operation in an unpredictable environment
- **Approach:** *model-based reasoning*
- **Pros:** smaller mission control crews, no communication delays/blackouts
- **Cons:** *Verification and Validation ???*
  Much more complex, huge state space
- **Better verification is critical for adoption**
Model-Based Autonomy

• Based on AI technology
• Generic reasoning engine + application-specific model
• Model describes (normal and faulty) behaviour of the process
• Engine selects control actions "on-the-fly" based on the model
  – ... rather than pre-coded decision rules
  – better able to respond to unanticipated situations
Livingstone

- **Model-based diagnosis system** from NASA Ames
  - i.e. an advanced state estimator
- Uses a discrete, qualitative model to reason about faults
  => naturally amenable to formal analysis

*Courtesy Autonomous Systems Group, NASA Ames*
A Simple Livingstone Model

Goal: determine modes from observations
Generates and tracks candidates

<table>
<thead>
<tr>
<th>breaker</th>
<th>bulb</th>
<th>meter</th>
<th>rank</th>
</tr>
</thead>
<tbody>
<tr>
<td>off⁰</td>
<td>ok⁰</td>
<td>ok⁰</td>
<td>0</td>
</tr>
<tr>
<td>off⁰</td>
<td>ok⁰</td>
<td>blown¹</td>
<td>1</td>
</tr>
<tr>
<td>on⁰</td>
<td>dead⁴</td>
<td>short⁴</td>
<td>8</td>
</tr>
</tbody>
</table>
Verify Model-Based Control?

Of course, but what exactly?

- The model?
- The engine?
- The whole controller?

All of the above!
Verification of the Model

- This is the "application code"
  - where the development effort (and bugs) are
- Abstract, concise, amenable to formal analysis
  - this is another benefit of model-based approaches
  - ... or model-based design in general
Model Checking

- **Model checking** = (ideally) exhaustive exploration of the (finite) state space of a system
  - $\approx$ exhaustive testing with loop / join detection

“Valve is closed when Tank is empty”

AG (tank=empty $\Rightarrow$ valve=closed)
Symbolic Model Checking

- **Symbolic** model checking =
  - compute sets of states,
  - using symbolic representations,
  - that can be efficiently encoded and computed.

- Can handle very large state spaces \((10^{50+})\), or even infinite domains (continuous time and variables)

- Example: **SMV/NuSMV** (Carnegie Mellon/IRST)
  - finite state using boolean encoding (BDD, SAT)
Livingstone-to-SMV Translator

Joint work with Reid Simmons (Carnegie Mellon)

- A translator that converts Livingstone models, specs, traces to/from SMV (in Java)
  - SMV: symbolic model checker (both BDD and SAT-based) allows exhaustive analysis of very large state spaces ($10^{50+}$)
- Hides away SMV, offers a model checker for Livingstone
- Enriched specification syntax (vs. SMV's core temporal logic)
- Graphical interface, integration in Livingstone development tools
Verification of Diagnosis Models

- **Coding Errors**
  - e.g. Consistency, well-defined transitions, ...
  - Generic
  - Compare to Lint for C

- **Model Correctness**
  - Expected properties of modeled system
  - e.g. flow conservation, operational scenarios, ...
  - Application-specific

- **Diagnosability**
  - Are faults detectable/diagnosable?
    - Given available sensors
    - In all/specific operational situations (dynamic)
Diagnosability

- **Diagnosis**: estimate the hidden state $x$ (incl. failures) given observable commands $u$ and sensors $y$.
- **Diagnosability**: Can (a smart enough) Diagnoser always tell when Process comes to a bad state?
- **Property of the Process** (not the Diagnoser)
  - even for non-model-based diagnosers
  - but analysis needs a (process) model
Verification of Diagnosability

- **Intuition**: bad is diagnosable if and only if there is no pair of trajectories, one reaching a bad state, the other reaching a good state, with identical observations.
  - or some generalization of that: (context, two different faults, ...)
- **Principle**:
  - consider two concurrent copies $x_1, x_2$ of the process, with coupled inputs $u$ and outputs $y$
  - check for reachability of $(\text{good}(x_1) \&\& \text{bad}(x_2))$
- Back to a classical (symbolic) model checking problem!
- Supported by Livingstone-to-SMV translator
In-Situ Propellant Production

- Use atmosphere from Mars to make fuel for return flight.
- Livingstone controller developed at NASA KSC.
- Components are tanks, reactors, valves, sensors...
- Exposed improper flow modeling.
- Latest model is $10^{50}$ states.
X-34 / PITEX

- Propulsion IVHM Technology Experiment (ARC, GRC)
- Livingstone applied to propulsion feed system of space vehicle
- Livingstone model is $4 \cdot 10^{33}$ states
PITEX Diagnosability Error

with Roberto Cavada (IRST, NuSMV developer)

• "Diagnosis can decide whether the venting valve VR01 is closed or stuck open (assuming no other failures)"

INVAR !test.multibroken() & twin(!test.broken())
VERIFY INVARIANT !(test.vr01.mode=stuckOpen & twin(test.vr01.valvePosition=closed))

• Results show a pair of traces with same observations, one leading to VR01 stuck open, the other to VR01 closed. Application specialists fixed their model.
Verification of the Controller

- good model + good engine $\neq$ good controller
  - Heuristics in engine, simplifications in model
- System-level verification
  - Controller as black (or grey) box
  - Need a model of the environment (test harness)
  - Applicable to others than model-based
Livingstone PathFinder

with Tony Lindsey (QSS @ ARC)

- An advanced testing/simulation framework for Livingstone applications
  - Executes the **Real Livingstone Program** in a simulated environment (testbed)
  - **Instrument** the code to be able to **backtrack** between alternate paths
- **Scenarios** = non-deterministic test cases (defined in custom language)
- **Modular** architecture with generic APIs (in Java)
  - allows different diagnosers, simulators (can use Livingstone), search algorithms (depth-first, breadth-first, heuristic, random, ...)
- See TACAS'04 paper
Verification of the Engine

- A (technically complex) computer program
  - Use traditional software verification approaches
  - Maybe full-blown proof on core algorithms
- Generic, re-used across applications
  - More likely to be stable and trustable
  - Like compilers, interpreters, virtual machines, etc
... and Verification of Software

• There is more to it than reasoning engines!
  – Device drivers, OS, navigation, communication, ...
  – real-time, concurrent, reactive, interrupts, priorities, ...

• All traditional good practices apply
  – Sound software engineering practices (requirements, design, modelling, documentation, reviews, testing, configuration management, ...)
  – Advanced software verification techniques (monitoring, static analysis, model checking, proofs)
The Program Verification Spectrum

(adapted from John Rushby)
Software Failure Example 1

Ariane 501 (1996)

- **cause**: fixpoint arithmetic overflow in guidance system
- **effect**: rocket and payload destroyed, program delayed
- **solution**: static analysis to detect potential runtime errors
  - This was the driving target for developing PolySpace
Software Failure Example 2

Mars Climate Orbiter (1999)

• **cause**: US/metric unit incompatibility between components

• **effect**: incorrect orbit insertion trajectory, probe crashed (and public embarrassment)

• **solution**: strong type checking, rigorous design practices
Software Failure Example 3

Remote Agent Experiment (1999)

- **cause**: missing critical section in concurrent program
- **effect**: race condition and deadlock in flight
  - in supervised experiment, no mission damage
- **solution**: model checking
  - a similar bug was found before flight using SPIN on another part of the code
Human Factors

• Adapt technology to its users
  – use their paradigms/languages (translation)
  – integrate in their tools and environments
  – vision: verification tools as advanced debuggers

• Technology maturation
  – From something that works to something that is usable
  – Lots of work and time
  – Polish the code but also documentation, training, etc

• Space mission adoption
  – Space missions take very conservative attitude w.r.t. new technologies (for good reason)
  – No-one wants to be the first adopter
  – Usefulness of technology validation missions
Conclusions

• Verification of control software
  – Particularity: control loop, observability/commandability
    • In particular, failure diagnosability and recoverability
• Verification of model-based controllers
  – Needs advanced verification (because of large state space)
  – Facilitates advanced verification (thanks to model)
• Model checking
  – Applicable to these problems
  – esp. symbolic model checking, esp. to model-based
  – Delicate precision/scalability trade-off
• Verification of software
  – All other principles still apply
Perspectives

• Key ideas:
  – model-based analysis (model checking)
  – partial observability

• Extensions
  – from discrete to continuous, real-time, hybrid models
  – from fault diagnosis to planning

• Connections
  – with classical risk analysis (fault trees, FMEA)
  – with man-machine interface issues (observability!)
  – with epistemic logics (diagnoser as knowledge agent)

• Keep in touch with reality
  – scalability, relevance to practical needs, tools, integration
References

• On this talk:

• See also
  – http://www.info.ucl.ac.be/~pecheur/publi/