Formal Methods for the Certification of Auto-generated Flight Code

Ewen Denney
Robust Software Engineering
NASA Ames Research Center
California, USA
Bugs in Space

• Mars Climate Orbiter (1998)
  – Unit problem in GN&C software
  – Crashed into Mars

• Mars Polar Lander (1998)
  – Inconsistencies in GN&C landing model
  – Premature engine shut-off
  – Crashed into Mars

• Mars Exploration Rover (2004)
  – Spirit shut down unexpectedly for 10 days
  – Flash ROM overload caused reboot
  – Parameter permitted unlimited consumption of system memory as flash memory was exhausted
  – Lost $4M science a day
It’s not just now …

• Gemini 5 (1965)
  – Missed landing point by 100 miles
  – GN&C didn’t model rotation of earth around sun

• Apollo 11 (1969)
  – Software reboot during descent to lunar surface
  – Forced manual landing of lunar lander
It’s not just NASA …

- Missed landing point by 275 miles
- “glitch in the craft’s guidance software”

Ariane 5 (1996)
- Bad 64 to 16 bit conversion led to overflow in GN&C software
- Veered off trajectory
- Self-destructed

Cryosat (2005)
- ESA Earth Explorer Mission to measure polar ice
- Launcher fell into ocean when fuel ran out
- Due to “software glitch” in control system: software failed to send command for 2nd stage separation (but “rocket is ok”)
It’s not just NASA …

- Missed landing point by 275 miles
- “glitch in the craft’s guidance software”

Ariane 5 (1996)
- Bad 64 to 16 bit conversion led to overflow in GN&C software
- Veered off trajectory
- Self-destructed

Cryosat (2005)
- ESA Earth Explorer Mission to measure polar ice
- Launcher fell into ocean when fuel ran out
- Due to “software glitch” in control system: software failed to send command for 2nd stage separation (but “rocket is ok”)
It’s getting worse …

Software for International Space Station (ISS) now estimated at 6.5 MLOC!
Future missions call for vastly increased levels of intelligence.

**Automated planning**
- On-board decision-making
  - Spacecraft operations
  - System management
- Schedule generation
  - Crew, equipment, systems

**Informed Logistics**
- Modeling of failure mechanisms
- Prognostics
- Troubleshooting assistance
- Maintenance planning
- End-of-life decisions

**Real-Time Systems Health Management**
- Distributed sensing for structural health
- Fault detection, isolation, and recovery
- Failure prediction and mitigation
- Crew and operator interfaces

**Adaptive Control**
- Improving safety and control performance beyond human ability
- Control in situations of failure or component degradation
- Operability in unknown or changing environments
The million $ question

Given increasingly
– complex systems
– compressed schedules
– safety-critical software,
how can we develop software which is
– reliable,
– sustainable, and
– certifiable?

Software certification: demonstrating that software meets its requirements and a given level of safety, either:
– through following a specified process (process-oriented certification), or
– through providing evidence that the level of safety is met (product-oriented certification)
Model-based Development

Earth Orbit OPS: GN&C SUBSYSTEM

Modeling:
Simulink (control), Stateflow (executive), Embedded Matlab (everywhere)

Code generation: Real-Time Workshop
Dear ..., 

If you are using R14SP3 Simulink code generation products, please review the following information. If you are not using R14SP3 versions of MathWorks products, please disregard this message.

We have identified bugs in R14SP3 Simulink® code generation products, which in rare instances generate incorrect code that is not easily detected. These bugs have been fixed in subsequent releases: R2006a, R2006b, or the upcoming R2007a release.

To prevent impact from these bugs, R14SP3 code generation software users should take the following actions:

*Review Related Bug Reports with Potential Workarounds*
You can find the documented issues and potential workarounds through the following links (login required):


Frequent updates - bug reports, work-arounds and fixes
Qualification

• A code generator is *qualified*
  – with respect to a given standard
  – for a given project
  
  if there is sufficient evidence about the generator itself so that V&V need not be carried out on the generated code to certify it

• Must be done for every project, version
  – can obtain *verification credit*

• Generators are rarely qualified
  – ASCET-SE (IEC 61508), SCADE, VAPS (DO-178B)

• Qualifying code generators is (almost) infeasible!
Commercial code generators are black boxes

- Not qualified so need to analyze generated code

- Historically buggy: despite extensive heritage, rare bugs still remain

- Cannot detect many bugs at model level or via simulation

- Math intensive code requires powerful analysis techniques
Product-oriented certification

• Augment code generator to generate certificates together with code (aka. the “verifying compiler” approach)
• No need to qualify/re-qualify code generator
• Code certificates:
  – proof of a specific safety property
  – can be independently verified
  – require only a small trusted infrastructure
  – process is completely automated
• Support engineers doing software assurance
  – generate safety documentation for human analysts
Assurance strategies for autocoding

- Documentation
  - *explain* the code synthesis and certification process
  - increases transparency and trust in process
- Traceability
  - *link* elements of generation process
  - mandated by NASA standards
- Proof
  - mathematical *proof* is gold standard
  - difficult to achieve and interpret without automation
  - show incrementally for individual properties

Use code generator plug-in to automate this
⇒ minimal impact to existing process
Technical approach

- Combine generator with certification plug-in: AutoCert
- Generate certificates which can be verified independently (IV&V)
- Based on formal logic
  - Range of safety properties
  - Pattern-based approach to inferring annotations
  - Fully automated
  - Can be used to generate explanations
  - Small set of trusted components
Language-specific safety properties

Language-specific safety properties to check specific constructs of the target language (C)

• Memory safety: array bounds
  – Buffer overflows often lead to unsafe programs

• Variable initialization before use
  – Un-initialized variables can cause unpredictable/unrepeatable effects
  – Compilers only check for initialization of scalars
  – e.g.: RTW Bug: uninit variables in DEMUX blocks
Domain-specific safety properties

- invariants
  - matrix symmetry: “covariance matrix P always symmetric”
  - quaternion, probability vector norms: “must add up to 1”
  - coordinate systems, units: “must use consistently”
  - arithmetic saturation/variable ranges: “actuator rate < 0.1”

- system-specific properties
  - “are all sensor data used?”

- block properties
  - “all values of x in interpolation table disjoint and increasing”

Simulink control models often “math-heavy”
• Small kernel of untrusted components
  
  - patterns and annotations *untrusted*
Covariance matrices (PM, PP) in a Kalman filter must remain symmetric during update.

- Individual matrix operations in the generated code must be checked:
  - $\mathbf{x} = \mathbf{R} + \mathbf{H}^T \mathbf{PMH}^T$ is symmetric

Annotations are required.

Analysis tool automatically generates annotations based upon idiomatic code patterns.

$K = \mathbf{PMH}^T \text{inv}(\mathbf{R} + \mathbf{H} \mathbf{PMH}^T)$;

$\mathbf{PP} = (\mathbf{I} - K \mathbf{H}) \mathbf{PM} (\mathbf{I} - K \mathbf{H})^T + K \mathbf{R} K^T$;
Vector norms

- Intuitively:  
  **Vectors must be normalized**

- Show preservation of norm by update operations

- Domain-specific requirement

- Requires code annotations
• Traceability:
  “the ability to link requirements back to rationales and forward to corresponding design artifacts, code, and test cases”
• Traceability:
  “the ability to link requirements back to rationales and forward to corresponding design artifacts, code, and proofs”
• “why is this line of code safe?”
  line of code → verification conditions
• “where does this condition come from?”
  verification condition → lines of code
Autocode safety reports

- Verification says *that* the code is safe
- Explanation says *why* the code is safe
- Use code analysis to generate *safety report*: explain how code complies with safety properties
  “the variable rtb_GetVeci is in the coordinate frame Earth-Centric Inertial because it is defined by applying the ECEF to ECI transformation to the variable … which is in turn…”
  ⇒ support code reviews
- Trace to relevant code fragments and model
Summary

• Formal basis
  – Safety requirements in first-order logic
  – Semantics in VCG
  – Prove VCs with ATP

• Tool
  – Tight integration with development tool suite
  – Trace code and model to verification artifacts
  – Trusted architecture

• Usage
  – Incremental approach
  – Generates safety documentation
  – Supports independent V&V