Model-based Development of Safety Critical Software: Opportunities and Challenges

John McDermid, FREng
Professor of Software Engineering, University of York
Director Rolls-Royce Systems & Software Engineering UTC
Director BAE Systems Dependable Computing Systems Centre
Non-Executive Director High Integrity Solutions (HIS)
Overview

- Objectives of model-based development
  - Comparisons with other areas
  - Safety critical software development

- Opportunities
  - Time and Money

- Challenges
  - Functionality
  - Change
  - Non-functional properties
  - Integration

- Conclusions
Model-Based Development

- Objectives in “traditional engineering”
  - Reduce risks, costs and timescales of developments
    - e.g. do bird strike tests only once
  - For example in aerospace and automotive industries

- Example of Rolls-Royce engine development
  - Extensive use of finite-element analysis
  - Mechanical properties of design
  - Aero-thermal design

- Mechanical design very advanced
  - Prediction of failure behaviour
  - Prediction of impact damage
  - Enables one-off tests validating the model
Cobbler’s Children

- In software development, little use of computer models
  - Extensive and expensive manual activity
Objectives for SCS

- Safety critical software has a good safety record

- Safety critical software is expensive
  - Circa 1 kLoC per person year, but much variation

- Sources of costs
  - Low level verification
    - Circa 25% of cost in unit/module test
  - Rework
    - Producing software to flight standard three times is not uncommon
  - Erroneous requirements
    - Perhaps 40% of post unit test errors for simple systems
    - As high as 85% for complex ones, e.g. F22

- Save time and money without reducing product integrity
Safety Critical Software is Growing

In Service Date

Code Size kLoC

- 1980
- 1987
- 1993
- 1999
- 2004
- 2014

Ada Europe - 6
Opportunities: Time and Money

- Code generation enables reduction of cost and time
  - Move from V model to Y
  - Early validation, automated analysis, greater abstraction …

![Diagram showing the transition from V to Y model]
Software Architecture

- Architecture is a “high level” design model
  - System components and interconnections

- Software architecture very broad and should cover
  - Functionality and interfaces
  - Data definition, data flow and information flow
  - Moding and scheduling
  - Timing and performance
  - Mapping to hardware
  - Failure behaviour and safety properties …

- Objective to have a rich model enabling
  - Validation and verification against (safety) requirements
  - Prediction of key implementation properties, with confidence
But Current Models are Very Low Level

Model is functional, doesn’t address timing, failure …

Need more expressive power, and more abstraction …
But there is a bigger problem ...
Analysis of Architectural Models

To avoid GIGO, analysis needs to address

- **Verification**
  - Does it meet the requirements?

- **Validation**
  - Is it consistent and complete (both internally and externally)?
  - Is it feasible (given the hardware resources)?
  - Does the model meet derived safety requirements (DSRs)?
  - Are there potentially unsafe deviations from design intent?

**Approaches**

- **Review**
- Safety analyses, e.g. HAZOP
- Automated analysis of specifications
  - Illustrate using extensions to Matlab/Simulink/Stateflow (MSS)
Illustrative Example

- Engine thrust reverser control
  - Reverses air flow to decelerate aircraft
  - Achieved by moving “Bucket Doors”
**Example of Automated Analysis**

- Example of aero-engine thrust reverser control
  - Aircraft deceleration using bucket doors
  - Hazard if used in flight or asymmetrically, or at too high thrust
  - Specified using state machines (Stateflow in MSS)
  - DSRs on safe operation and recovery, e.g. interlocks

- Analysis via extraction of the model, DSRs and formal proof
  - Completeness, internal/external consistency, meets DSRs …

NB Software “unsafe” if its view of the world differs from reality
Example of DSR and Analysis

- Analysis for validation, and verification against DSRs
  - Automated analysis approach
    - Healthiness checks, e.g. determinism
    - Annotations to define DSRs, linked to state machine
    - Assumptions which model behaviour of embedding system/physics
    - Formal analysis to check DSR holds
    - A counterexample is given if the check fails

- Checks reduce chance of GIGO due to model errors
The Challenge of Change

Change is inevitable

Benjamin Disraeli, 1867

- Can reduce the likelihood of change
  - Verification and validation, e.g. as illustrated

- Can reduce the impact of change
  - Automated verification and validation
  - Design to accommodate change
    - Product lines, strong similarity between products
    - Produce configurable assets for product line
    - Select and configure for particular products
    - Save time, reduce risk of error and enforced change
    - Embed in models, making them configurable
Example: Engine Starting

[Diagram of engine starting process with labeled nodes and arrows]

Ada Europe - 16
Changing between Product Line Members

Before:
- Target
  - AdjustDrives
  - Configurations
  - SAV
  - Library
  - libAdjustDrives

After:
- Target
  - AdjustDrives
  - Configurations
  - noSAV
  - Library
  - libAdjustDrives
Top-Level Model – Change Localised
Adjust Drives – No SAV
Product Line Management

● Benefits
  ■ Encodes product line ideas in tools used by design engineers
  ■ Can produce checks to ensure sound configuration
  ■ Can verify and validate components independently
  ■ Save time, money and reduces risk
    ◆ Controlled reuse

● Limitations
  ■ Quite complex to encode in current tools
    ◆ In MSS some ugly “mechanics” to realise variability
    ◆ Hard to ensure consistent change to models held by multiple tools
  ■ Difficult to reduce/remove need for re-verification
  ■ Limited help with unpredicted changes
  ■ Doesn’t directly address non-functional properties
Non-Functional Properties

- Non-functional is an awful term
  - Aspects of behaviour, not just “ideal functionality”

- Range of properties of interest
  - Some, e.g. timing, can be represented as attributes
  - Others, e.g. fault management, require new/modified functions

- Timing
  - Can articulate requirements for software
    - Deadlines, jitter, etc.
  - Annotate models with WCET, etc. (estimates or actuals)
  - Undertake analysis or synthesise schedules

- Consider fault accommodation
Fault Management Code

- Development generally a manual process
  - Costly, may be more than half system code
  - Error prone, and likely to change

- Alternatively, automate configuration
  - Provide configuration for existing product-line components
  - Select software components based on data on
    - Hardware failure modes (FMEAs)
    - Configuration rules (fragments of Markov models)
  - Code production by reuse, not generation
    - Change handled through selection of different code templates

- Traceable behaviour
  - From choice of component back to requirements
**Software Layering**

- **System functions**
- **Drivers for devices**
  - Validated sensor data and actuator control
- **Fault management**
  - To produce “trusted” data for application
    - NB hazard if software view of world differs from reality
- **Abstraction from processing hardware**
  - Operating system

---

- Application Requirements from Platform Level
- Hardware Abstraction Layer (HAL)
  - Isolation from details of Computing Hardware
- Failure Management for embedding system sensors and actuators
  - Isolation from details of Sensors and Actuators
- Sensed Values
- Validated Values
Fault Management Logic

- Fault-accommodation requirements in Markov model
  - Can despatch (use) system “carrying” failures
    - Despatch analysis based on Markov model
    - Evaluate probability of being in non-dispatchable state, e.g. only one failure from hazard
    - Link between safety/availability process and software design
  - Auto-generation ensures software and analysis in step
    - Reuse pre-verified fault-accommodation modules

- May use four valued logic
  - Working, undetected, detected, and confirmed
  - Table illustrates “logical and” (\[.\])
  - Used for analysis

<table>
<thead>
<tr>
<th>.</th>
<th>w</th>
<th>u</th>
<th>d</th>
<th>c</th>
</tr>
</thead>
<tbody>
<tr>
<td>w</td>
<td>w</td>
<td>u</td>
<td>d</td>
<td>c</td>
</tr>
<tr>
<td>u</td>
<td>u</td>
<td>u</td>
<td>d</td>
<td>c</td>
</tr>
<tr>
<td>d</td>
<td>d</td>
<td>d</td>
<td>d</td>
<td>c</td>
</tr>
<tr>
<td>c</td>
<td>c</td>
<td>c</td>
<td>c</td>
<td>c</td>
</tr>
</tbody>
</table>
Example Implementation
Deriving Safety Analyses

- By adding failure assumptions to models, possible to generate safety analyses
  - Complements work on fault management
    - Derive safety models used for certification
  - Several alternative approaches
    - Needs semantic model for failures and propagation
  - Several challenges
    - Scale, intelligibility of output, trust in tools
  - Requires integration
Integration

● Need (at least)
  ■ Notational integration
  ■ Method/process integration (development and safety processes)
  ■ Toolset integration

● Notations
  ■ Expressive enough to cover all properties of interest
  ■ A “single” notation, or related views

● Architecture Analysis and Definition Language (AADL)
  ■ Developed out of work by Honeywell and US Army
  ■ Good concept, with growing support
    ◆ Notation, tools and SAE standard
  ■ Potential for timing / reliability / safety analysis
Process and Toolset Integration

- Most tools are quite specialised
  - Do some things well
  - Don’t address all relevant issues, e.g. don’t model all of the architectural properties, and are unlikely to address all

- Need to set up
  - Process models, to link activities
  - Data models, to link notations and to provide traceability
  - Tool infrastructure that realises links including impact analysis

HIS V-Model
Conclusions

● Model-based development important for future safety critical software developments
  ■ Believe this will become the norm, in time

● So, is this the end for program level analysis?

● No
  ■ Currently, program level toolsets, e.g. SPARK Examiner better developed than modelling tools – for safety critical software
  ■ Much code generation will be linking pre-defined code modules
    ◆ These modules need to be developed and verified
    ◆ Continued challenges in compositional verification

● Model based development will shift balance …
Ada Joint Program Office

awards

Ada Validation Certificate # 890531N1.10097
to

York Software Engineering Limited

for successfully validating

York Ada Compiler Environment (ACE) Release 4

Tested Configuration

Host(s): Intergraph Inter Pro 340 (under UNIX System V.3)

Target(s): Same as Host

ACVC Version: 1.10

National Computing Centre, U.K.
Ada Validation Facility

26 June 1989
Date of Issue

01 December 1990
Expiration Date