Fault Tree Analysis of UML Designs

Dr. Christopher Harper
Director, Avian Technologies Ltd.
www.avian-technologies.co.uk
cjharper@avian-technologies.co.uk

Alan Parkinson
Director, AGP Micro Ltd.
www.agpmicro.co.uk
alan@agpmicro.co.uk
Contents

• The Engineering Problem
• Failure Analysis using Fault Trees
• UML Semantics
• Fault Tree Analysis of UML
• Model-driven Safety Validation
• Current work
Critical systems require dependability assurance.

Dependability assurance requires a lot of extra work (analysis, review, testing).

The extra work can introduce late design changes into the system.

Late design changes are extremely expensive!
Train Brake Control System: Context Use Case Diagram

- Emergency Brake Button
- Driver Service Brake Handle
- Emergency Braking
- Wheel Slide Protection
- Service Braking
- <<extend>>
- <<include>>
- Brake Cylinder Pressure
- Door Control
- Brake Management
- Door Control Relay
- Tacho Sensors
- Fire Alarm Signal
- Low Pressure Sensor
- Brakes
- Brakes
- Brakes
- Brakes
- Brakes
Door Control: Class Design

Class Model

Door Control

Tacho Interface

WheelTacho
+ Speed getAxleSpeed()

<<uses>>

Type definition:
Speed: real[0..500]

<<enumeration>>
AlarmType
- Active
- Inactive

DoorController
+ void ControlDoors()

FireAlarmSignal
- AlarmType FireAlarmState
  + bool IsFireAlarmActive()
Train Brake System: Interaction Diagram

- DoorController
- WheelTacho
- FireAlarmSignal
- Door Control Relay

- ControlDoors()
- getAxleSpeed()
- Speed AxleSpeed
- IsFireAlarmActive()
- bool AlarmActive

[if]
- AlarmActive = true & AxleSpeed < 5

[else]

- OpenDoors
- LockDoors

© 2007 Avian Technologies Ltd
UML 2.0+ Semantics (1)

- UML semantic model has two parts, the semantics architecture and the causality model [Selic]

Semantics Architecture:

<table>
<thead>
<tr>
<th>Activities</th>
<th>State Machines</th>
<th>Interactions</th>
</tr>
</thead>
</table>

**Behavioural Base**
- behaviours
- behavioural classifiers

**Actions**
- Defines semantics of individual actions
- Communication, Read/Write, Computation, Structural Actions

**Inter-object Behaviour Base**
- Defines how structural entities communicate
- Sync/async calls, operations/receptions, events, triggers

**Intra-object Behaviour Base**
- Defines how structural entities behave internally
- Activities, structured activities, nodes, edges, flows

**Structural Foundations**
- no disembodied behaviour in UML
- all behaviour generated by actions of structural elements
- pure values, variables, objects, links, messages, composites/aggregates

© 2007 Avian Technologies Ltd
• Basic unit of behaviour is the action

• Actions are specified as:
  – Output Pins = f(Input Pins, Structural Elements)

• Pins are the parameters of class operation signatures

• **Causality Model** [Selic]

  • Active objects have a thread
    → time of response not dictated by calling object
    → run-to-completion semantics
    → messages arriving between activations are buffered

  • Passive objects run only when called
    → responds regardless of whether a previous call has been completed (requires mutex to protect, if necessary)
Semantic Variation Points

- Not all the semantics required to specify correct operation is defined within the UML standard, BUT…

- ‘placeholders’ are left for the gaps – Variation Points

- Variation points must be completed by suppliers of executive software systems, e.g. RTOS, Comms

- Care must be taken! Models may have different behaviour under different variation points. Assumptions about semantics must be recorded.

- But, given these caveats, we believe that the semantics of UML is (just) sufficient to support more advanced model-based analysis, (e.g. failure analysis)
Failure Modelling

- Failure analysis is only complete with respect to a particular failure model, which must be a prior assumption at the start of any analysis.

- Failure model used in this methodology is drawn from techniques developed by David Pumfrey at the University of York [Pumfrey].

- Six basic software failure types:
  
<table>
<thead>
<tr>
<th>Service Type</th>
<th>Failures</th>
</tr>
</thead>
<tbody>
<tr>
<td>Provision Failures</td>
<td>Omission, Commission</td>
</tr>
<tr>
<td>Timing Failures</td>
<td>Early, Late</td>
</tr>
<tr>
<td>Value Failures</td>
<td>Coarse (implausible value), Subtle (plausible value)</td>
</tr>
</tbody>
</table>

- Failure model used extensively by UoY in conjunction with aerospace industry (e.g. Airbus, BAE Systems) and by Avian Technologies on contract work in the rail sector.
UML Interpretation of Failure Model

- The standard failure types must be *interpreted* in UML terms:

<table>
<thead>
<tr>
<th>UML Model Element Type</th>
<th>Service Provision Error</th>
<th>Service Timing Error</th>
<th>Service Value Error</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Omission</td>
<td>Early</td>
<td>Coarse (Implausible)</td>
</tr>
<tr>
<td></td>
<td>Commission</td>
<td>Late</td>
<td>Subtle (Plausible)</td>
</tr>
<tr>
<td>Synchronous Operation Call</td>
<td>Call Action associated with message is not executed at the intended point in the behaviour</td>
<td>Call Action associated with this message executed instead the intended one</td>
<td>N/A (GeneralOrdering error?)</td>
</tr>
<tr>
<td>Combined Fragment : Loop</td>
<td>Loop body not executed when intended:</td>
<td>Loop body executed when not intended:</td>
<td>As commission</td>
</tr>
<tr>
<td></td>
<td>- \text{minint} parameter value error (is too low, or is zero)</td>
<td>- \text{minint} parameter too high</td>
<td>As omission</td>
</tr>
<tr>
<td></td>
<td>- \text{maxint} parameter value error (too low)</td>
<td>- \text{maxint} parameter too high (or infinite)</td>
<td>N/A</td>
</tr>
<tr>
<td></td>
<td>- guard condition incorrectly false</td>
<td>- guard condition incorrectly true</td>
<td>N/A</td>
</tr>
</tbody>
</table>

© 2007 Avian Technologies Ltd
A fragmentary fault tree template is defined for every type of failure of every type of model element in a class of UML behaviour diagram [cf. Leveson et al]. Each fragment relates the failure type of the associated UML Action (at its output pins) to a set of failures of its input pins.

The causal logic of each fragment is specific to the type of action. Its structure may depend on semantic variation points – in this case the relevant variation must be selected as the analysis unfolds.

Pattern-based methodology (cf. OO design patterns, safety case patterns [Habli & Kelly])
The template-driven approach ensures that a consistent interpretation is taken of the causal relationships between input and output pins of actions.

A UML diagram can be scanned automatically to obtain the overall structure of its associated fault tree.

The end result is a Fault Tree that is logically consistent with, and systematically derivable from, its associated UML Behaviour Diagram.

The procedure can be (partially) mechanized, thereby accelerating the analysis process and improving its consistency (but a human analyst must still decide which template fits into each slot).
The Counterfactual Nature of Failure Analysis

IF this system failure occurs...

THEN these are the software errors that would have to have happened...

and these are the basic faults that would have to have happened to cause error A ...

- Doors open while train is moving (>= 5kph)
- OpenDoors signal sent when not intended
- AxleSpeed < 5 when not intended
- Tacho sensor input reads < 5kph when not intended
- Wheel slip causes low speed measurement
- Tacho sensor fault
- getAxleSpeed() called when not intended
- ControlDoors() called when not intended
- ControlDoors() calls getAxleSpeed() when not intended
- AlarmActive = true when not intended

© 2007 Avian Technologies Ltd
The Counterfactual Nature of Failure Analysis (2)

BUT:
if the basic faults don’t happen …

…this system failure can’t happen!

SO:
If we test the software for the existence of the basic faults…
…we can determine whether the software will cause the system-level hazard.

This is the real value of performing a software FTA.
Synthesis of Safety Validation Tests from Fault Trees

A fault tree corresponds with a set of DNF (sum-of-products) statements, which are derived from the paths from the top condition to the base events. Two statements are shown below, derived from the paths shown by the white trace through the tree:

1. \((\text{Tacho}\_\text{fault} \land \text{FireAlarmState}=\text{Active})\)
2. \((\text{Wheel}\_\text{slip} \land \text{FireAlarmState}=\text{Active})\)

These sets of events are, in effect, test patterns that cause the potential hazard to occur at the output. They can be applied as test cases to verify this fact.

Fault trees can also reveal test cases for other operations/classes/modules that require assessment, because they are sources of sensitive base events causing the failure condition (e.g. red dotted path).

- Test cases can be derived easily from fault trees, because fault trees specify a set of weakest pre-conditions for the occurrence of a hazard (the top condition of the tree) [McDermid&Clarke].
- The inverse or negation of each pre-condition forms a verification condition, or could be re-written as a design inspection criterion.
- UML is well-suited to this because it includes the language OCL, which is intended for the specification of design pre-condition and post-condition information.
Synthesis of Design Corrections from Fault Trees

- Fault trees can be used to drive design changes by exploring the tree for vulnerable points (where paths from top events to base events consist only of OR-gates) and correcting the design so as to (re)introduce AND gates into the new fault tree for the corrected design. [Illustrated by changes marked in green.]
- This can also be done by the creation and enforcement of design contracts (pre-conditions or post-conditions to code execution) because of the relationship between fault trees and weakest preconditions.

**AxleSpeed < 5 when not intended**

- Validated Tacho sensor input reads < 5kph when not intended
- getAxleSpeed() called when not intended
- ControlDoors() called when not intended
- Tacho semaphore available when not intended

**Tacho #1 sensor fault**
- Axle #1 slip causes low speed measurement
- Tacho #1 sensor fault

**Tacho #2 sensor fault**
- Axle #2 slip causes low speed measurement
- Tacho #2 sensor fault

NOTE: Assigning the maximum value to the axle speed is a safe-side response to the validity error.
HiCASE – A New Software Tool for Safety Assurance of SysML/UML Designs

Avian Technologies and AGP Micro are working on a joint venture to develop HiCASE - a software tool for safety assurance and engineering of UML (and eventually SysML) system designs:

- **Objectives**
  - Reduce costs of developing high integrity systems/software by improving integration of safety processes with design processes
  - Semi-automatic generation of safety analyses directly from UML/SysML models
  - Automatic generation of safety validation tests/checklists from causal analyses (fault trees)
  - Automatic generation of safety properties as class contracts for design-by-contract processes and formal verification (e.g. SPARK)

- **Production Schedule (approximate)**
  - Product launch early summer 2007
  - Functionality upgrades/enhancements at least once per year (for subscription holders)

- **Product Versions**
  - Professional Edition: basic UML model scanning and safety analysis capabilities only
  - Enterprise Edition: full range of functions
  - Certification Pack: additional documentation for DefStan00-56 qualification (all SILs)
References