Fault Propagation and Transformation: A Safety Analysis

Malcolm Wallace
Safety is not the same as correctness.

- Correctness = theorem-proving, model-checking, abstract interpretation, etc.
- Safety = analysis of product + context, response in the presence of failure (internal or external)

Safety is not compositional.

Examples of safety techniques:

- Multiple (identical) redundancy
- Multiple implementation methods (non-identical)
Motivation

'Design-in' safety, at the architectural level
- Identify hazards
- Identify failures that could trigger a hazard
- Demonstrate that failures are caught or mitigated
- Certify the whole system

Issues:
- Huge cost (analysis is labour-intensive)
- Non-modular (safety of system, not components)
- Incremental certification?
- Impact of late design changes?
- Product-line families?
Contribution

- Define a modular, compositional, notation and analysis for failure behaviours.
  - Tackle a small portion of the problem first.

- Calculate the system failure behaviour automatically from the composition of its parts.
  - Reduce the cost of prediction
  - Both in initial design, and in estimating the impact of a change.
A simple real-time architecture
Real-Time Networks

- IMU
- Actuators
- Inertial navigation
- Body motion
- Mode
- Mode events

server (hardware input)
server (hardware output)
software activity (thread)
signal (blocking read and write)
pool (destructive write)
channel (blocking write)
### RTN protocols

<table>
<thead>
<tr>
<th>Non-destructive write</th>
<th>Destructive read (blocking)</th>
<th>Non-destructive read (non-blocking)</th>
</tr>
</thead>
<tbody>
<tr>
<td>(blocking)</td>
<td><img src="image" alt="Signal" /></td>
<td><img src="image" alt="Constant" /></td>
</tr>
<tr>
<td></td>
<td><img src="image" alt="Channel" /></td>
<td><img src="image" alt="Pool" /></td>
</tr>
<tr>
<td>Destructive write</td>
<td></td>
<td></td>
</tr>
<tr>
<td>(non-blocking)</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Faults

Identify all 'interesting' fault types (following HAZOP/SHARD guidelines):

- **value faults:**
  - stale value
  - detectably incorrect value
  - undetectably incorrect value

- **timing faults:**
  - early
  - late

- **message sequence faults:**
  - omission
  - commission
Propagation and Transformation

How does a component deal with a failure on input?

* → late
early → *
 omission → omission
 late → stale value

(source, sink, propagate, or transform)
Fault Propagation and Transformation

- An FPTC expression is a collection of clauses.
- Each clause expresses one transformation behaviour.
- Combination of clauses gives full Boolean expressivity.
- Notation can easily be extended to describe sets of faults.

\[
egin{align*}
* & \rightarrow \text{Value} \\
* & \rightarrow \text{Omission} \\
(*,*,\text{Value}) & \rightarrow * \\
(*,\text{Value},*) & \rightarrow * \\
(\text{Value},*,*) & \rightarrow * \\
\text{Late} & \rightarrow \text{Value} \\
\text{Early} & \rightarrow * \\
\text{Omission} & \rightarrow \text{Value} \\
\text{Commission} & \rightarrow * \\
\text{Omission} & \rightarrow \text{Late} \\
\text{Commission} & \rightarrow \text{Late}
\end{align*}
\]
Behaviour of protocols

<table>
<thead>
<tr>
<th>Signal</th>
<th>Channel</th>
<th>Pool</th>
</tr>
</thead>
<tbody>
<tr>
<td>omission -&gt; late</td>
<td></td>
<td></td>
</tr>
<tr>
<td>omission -&gt; late</td>
<td></td>
<td></td>
</tr>
<tr>
<td>early -&gt; *</td>
<td></td>
<td></td>
</tr>
<tr>
<td>* -&gt; late</td>
<td></td>
<td></td>
</tr>
<tr>
<td>late -&gt; stale value</td>
<td></td>
<td></td>
</tr>
<tr>
<td>omission -&gt; stale value</td>
<td></td>
<td></td>
</tr>
<tr>
<td>early -&gt; *</td>
<td></td>
<td></td>
</tr>
<tr>
<td>commission -&gt; *</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Algorithm

- Architecture is a graph
  - Nodes have propagation/transformation expressions
  - Edges simply connect nodes
- Regard faults as tokens introduced by error sources
- Push the tokens round the graph network
  - Use FPTC expressions to remove, propagate, or transform them
- Collect sets of all possible tokens on each edge
- Calculate the fixpoint (i.e. maximal fault sets)
Algorithm

\[ \varnothing \left[ \begin{array}{c} * \rightarrow f \\ \end{array} \right] \_ = \{f\} \]

\[ \varnothing \left[ \begin{array}{c} f \rightarrow * \\ \end{array} \right] f' = \emptyset \]

\[ \varnothing \left[ \begin{array}{c} f \rightarrow g \\ \end{array} \right] f' = \{g\}, \text{ if } f == f' \]
\[ \emptyset, \text{ otherwise} \]
stabilise(graph) = fix (\( \lambda g \rightarrow \text{forall } c::\text{Component} \in g . \text{evaluate}(c) \)) graph

fix(f) x \mid x == f(x) \rightarrow x
| otherwise \rightarrow fix(f)f(x)

evaluate(c) = outputTokenSet(c) \cup \cup \{ \text{forall } t \in \text{transformClause}(c), \text{forall } f \in \text{inputTokenSet}(c) . \varphi[[t]]f \}
Algorithm properties

- Terminating
- Confluent
- Stable – no 'flip-flops'
BVRAAM launch software design
BVRAAM – mode controller
BVRAAM – aircraft comms

- Aircraft comms diagram:
  - Aircraft
  - Target position
  - Aircraft messages in
  - Aircraft messages out
  - Clocks clk2, clk3
  - Read aircraft messages
  - Transfer alignment
  - Aircraft manoeuvre
  - Initial position
  - Aircraft INS data
  - Manage BIT
  - BIT command
  - Write aircraft messages
  - Status reporting
  - Missile status summary
  - Status reports
  - Missile ident

- The diagram illustrates the process flow for BVRAAM, focusing on aircraft communications and alignment with INS data.
Propagation Analysis

- Put together:
  - faults potentially injected by individual components (hardware or software)
  - the network of connections
  - transformation of faults through components

- To get:
  - potential faults expressed at outputs

- Determine where to place fault accommodation code
Design of the Notation

- **Default clauses**
  - *→late, omission→late, value→value
  - What happens in response to e.g. Early?

- **Multiple connections**
  - Implicit multiplexing?
  - Tupling?
    - late→(value,*,late)
  - Interaction with default clauses?
  - Exponential growth in input clauses
    - 6 failure modes, 5 inputs = $7^5 = 16807$ patterns
    - Permit variables, wildcards, sets
Pattern language design

- Variables and wildcards:
  - (value, _) → value
  - (late, f) → f

- Sets to amalgamate similar clauses:
  - (f, g) → (*, f), (f, g) → (*, g)
  - (f, g) → (*, {f, g})

- Overlapping patterns:
  - (*, late) → (*, *)
  - (*, f) → (*, f)

- Complex overlaps:
  - (X, Y, _) → A, (_, _, Z) → B
Conclusions

- Architectural description must be modular
  - Identify components, connections, dependencies

- As one aspect of safety, we are interested in failure behaviours, rather than correct behaviour
  - Express response to failures on input as potentially different failures on output
  - Manual analysis of components in all possible contexts

- Automate the calculation of the whole-system response
  - Changes to the model only require a cheap re-run of the algorithm
  - Full impact of change can be assessed
Quality of Service analysis

- A similar technique could be used to analyse QoS issues
- E.g. instead of tracking value faults, track value quality: \{lo, med, hi\}
- Or maybe SIL levels
- Combination of multiple sources increases confidence in data quality (Med, Med, Lo) -> Hi