Failure Modelling in Software Architecture Design for Safety

Weihang Wu
Tim Kelly
Presented by George Despotou

High Integrity Systems Engineering Group
Department of Computer Science
Outline

- **Motivation**
  - The role of feedback in architecting dependable systems
  - The need for compositional and automated safety analysis
  - The value of CSP
  - The relationship between system modelling and failure modelling

- **CSP Failure Modelling Approach**
  - The process view
  - Architecture transformation
  - Failure modelling
  - Causal analysis
  - Use of CSP tools

- **Summary**
  - Initial results
  - Ongoing work
Motivation 1

- **Architectural Feedback on Safety**
  - Evaluate the impact of architectural decisions on safety (safety tactics)
    - How to select or identify proper scenarios for evaluation
    - Protection mechanisms themselves may fail
  - Validate existing safety requirements
  - Elicit new safety requirements to subsequent refinement process
  - Analyse safety implications on software-hardware mapping
  - Predict both normal and failure behaviours of the system

- **Software Safety Analysis of Architectures**
  - An underlying formal model
  - Compositional reasoning
    - Compositional features of architectures must be acknowledged
  - Expressive power
    - Common failure scenarios such as sequential failures, cascading failures, and common-cause failures
  - Automation support
Value of CSP

- Mathematical language devised to solve concurrency problems
  - Freedom of deadlocks and livelocks
- Formal specification of systems behaviours
  - In terms of patterns of event sequences or component interactions
  - Architectural description language – Wright
- Compositional reasoning is an integral part of the language
- Explicit notation for specifying nondeterminism
  - Arise from the abstraction techniques or incomplete knowledge
  - Identify alternative failure flows in an unconstrained manner
- Two important tools available
  - Animator (ProBE) and model checker (FDR2)
- Recent work on timed and probabilistic extensions

System Modelling and Failure Modelling

- System modelling: only normative events are observable
  - Failure events are implicitly seen as anti-occurrences of normative events
- Failure modelling: all failure events are explicitly observable
  - Normative events are only modelled if necessary
- System modelling languages such as CSP can be extended to model failure behaviours
Failure Modelling Approach 1

The Process View

- Establish a correspondence between failure behaviours of a system and its underlying software architecture
  - Architectural building blocks
    - Components and connectors, safety-related architectural decisions, architectural views
  - CSP building blocks
    - Processes, channels (events)
- We treat architectural design as an iterative and incremental development process
Failure Modelling Approach 2

- Architectural Transformation
  - TMR system example

UML-RT class diagram for TMR style

UML-RT collaboration diagram for TMR system

CSP model

\[
P_1 = \text{PROCESS } [[\text{input} \leftarrow \text{in1}, \text{output} \leftarrow \text{out1}]] \\
P_2 = \text{PROCESS } [[\text{input} \leftarrow \text{in2}, \text{output} \leftarrow \text{out2}]] \\
P_3 = \text{PROCESS } [[\text{input} \leftarrow \text{in3}, \text{output} \leftarrow \text{out3}]] \\
V_{T1} = \text{VOTE } [[\text{sender} \leftarrow \text{out1}, \text{receiver} \leftarrow \text{input1}]] \\
V_{T2} = \text{VOTE } [[\text{sender} \leftarrow \text{out2}, \text{receiver} \leftarrow \text{input2}]] \\
V_{T3} = \text{VOTE } [[\text{sender} \leftarrow \text{out3}, \text{receiver} \leftarrow \text{input3}]] \\
V_1 = \text{VOTER } [[\text{result} \leftarrow \text{output}]]
\]
Failure Modelling Approach 3

**CSP Failure Modelling**
- Identification of failure events
  - Identify failure modes by guidewords such as SHARD/HAZOP
  - Failure model allocation/injection to the CSP system model
- Expressive power
  - CSP support the definition of multi-part events by infix dot
    - All events must have one part describing normal or failure conditions such as sensor.failed, processor.working
  - Failure flows can be captured by CSP sequencing and recursion operators
  - Combination of failure flows can be modelled by the introduction of deterministic or nondeterministic choice
    - Depend on the degree of knowledge

---

```
-- Crash failure
CPU_CH = cpu.failure.omission -> CPU_CH

-- Transient timing failures
CPU_TF = cpu.failure.timing -> CPU_TF
  cpu.ok -> CPU_TF

-- Transient value failures
CPU_VF = cpu.failure.value -> CPU_VF
  [] cpu.ok -> CPU_VF

-- Corruption failures
CPU_CRT = CPU_TF [] CPU_VF
```
Failure Modelling Approach 4

- Failure Modelling
  - Two basic forms of failure flows
    - Failure propagation
      - Include failure transformation and stopping by protection mechanisms
    - Failure generation
      - The cause of failure stimulus has been hidden by model view
      - The cause may arise from its enclosing components or its underlying hardware platform
  - Interaction between these two forms
    - Inconsistency may arise: e.g., a timing failure arrives at the input of component C, whilst C itself generates an value failure
    - Proper form of arbitration is needed
  - Failures of protection mechanisms
    - The ways to handle failures are obvious
    - But what if these mechanisms fail?
      - What happen if a watchdog timer fails?
    - The answer may depend on internal detailed design or implementation
    - Worst case assumption
      - Specify the occurrences of all possible failure outputs introduced by nondeterministic choice
Compositional Failure Modelling

- CSP composition rule
  - Handshaking synchronisation
  - Processes to be composed require synchronised events

- Failure implications on synchronisation
  - Synchronisation point represents the means to failure propagation across component boundaries
  - Unsynchronised failure events are free to occur only within the component boundary
    - E.g., internally generated failure events

- Composition of components within one view
  - Define failure behaviours of elementary components
  - Compose all elementary processes using CSP parallel composition operators
    - \( TMR_{CCVIEW} = ((P1 [[out1]] VT1) ||| (P2 [[out2]] VT2) ||| (P3 [[out3]] VT3)) [[input1, input2, input3]] V1 \)

- Composition of views
  - Require synchronisation points between views
    - Mapping between them needs to be defined before composition
    - E.g., C&C view and hardware architecture view cannot be composed directly without the allocation view
Failure Modelling Approach 6

- Causal Analysis
  - CSP view of causality
    - Temporal ordering and handshaking synchronisation
      - Trace model
    - Necessary condition of causality
  - Conclude causal relationships based on trace models
    - By changing the states of event sequences
      - Borrowed from Philosophy domain: there is a causal connection between A and B if and only if we can change B by changing A
    - Similar to the tenet of accident analysis techniques such as Why-Because Analysis

- The steps
  - Isolate the initiating event
  - Treat CSP external choice notation as logical disjunction
  - Treat CSP sequential notation as logical conjunction
  - Treat normal events as non-occurrence of failure events

\[
\text{occur(output.failure.V)} = (\text{occur(a.ok)} \land \text{occur(b.fail)}) \lor \\
(\text{occur(a.fail)} \land \text{occur(b.ok)}) \lor \\
(\text{occur(a.fail)} \land \text{occur(b.fail)}) \lor \\
\text{occur(a.fail)} \lor \text{occur(b.fail)}
\]
Failure Modelling Approach 7

- Use of CSP Tools
  - ProBE
    - Validate intended failure behaviour
  - FDR2
    - Verify the consistency of a failure view
    - Refinement checking between views
      - E.g., allocation failure view refines the C&C view
      - assert TMR_CCVIEW [T= TMR_ALLOCVIEW \ ICpu
    - Generate failure scenarios by counterexamples
      - Failure scenarios of interest are the ones related to system-level failures
      - Specify safety properties that exclude undesired system events
      - Perform trace refinement against safety properties
      - FDR2 provides batch interface for direct control on counterexample generation

```plaintext
ISafeSys = diff(Events, {output.failure.V})
-- anything but value failures of output allowed
SAFESPEC = [] x : ISafeSys @ x -> SAFESPEC
assert SAFESPEC [T= TMR_CCVIEW
```
Summary

- **Small-Scale Examples**
  - Architectural documentation by UML-RT
  - Two architectural views
    - C&C and allocation views
  - Uniprocessor hardware platform

- **Findings**
  - The choice of architectural representations/descriptions is not important to our method
    - Provided that the corresponding transformation rules are well defined
  - Architecture description is not necessarily complete
  - A hardware/system architecture view must be provided
    - This view can be derived by the allocation view or hardware architecture design

- **Ongoing Work**
  - Generating CSP codes from annotated architecture models
    - Architecture annotation
      - UML 2
    - CSP code generation
  - Probabilistic failure modelling