22nd IEEE International Conference on
Emerging Technologies and Factory Automation

ETFA'2017

Software Patterns for Fault Injection in CPS Engineering

Nicolas Navet, Ivan Cibrario Bertolotti, and Tingting Hu

UNIVERSITÉ DU LUXEMBOURG

Motivation

Model-Driven Engineering (MDE) and Domain-Specific Languages (DSL) have been widely acknowledged as two key technologies to meet the **software productivity** challenge and develop **trustworthy** systems, in particular for Cyber Physical Systems (CPS).

- CPS are subject to **dependability** constraints.
- **Fault injection** is an effective technique to assess the dependability of a system. Fault injection can be implemented either in hardware (HIFI) or **software** (SWIFI).
- However, it is time consuming and requires extensive know-how to implement software fault injection correctly and to determine the verification coverage of an experiment
- Instead, **software patterns** are able to capture important practice in a form that makes the practice accessible. When combined with MDE, it allows to detect errors in the early design phase.

Objectives

- Propose a set of **software patterns** that implement fault injection with modeling languages or language extensions, such as StateFlow, **CPAL** or Mbeddr that natively support Finite State Machines (FSMs).
- Seamlessly integrate the verification activity by means of fault injection and simulation within the **design flow** and to fully, or partially **automate** it.

The Cyber Physical Action Language

- A representative domain-specific language for embedded systems designed for MDE.
- Offers high-level abstractions to express domain-specific properties or patterns of behaviors well suited to embedded systems with timing and dependability constraints, and for CPS at large.
- Verification in CPAL can be achieved by means of schedulability analysis, timing accurate simulation, and runtime observation.
- It is not only a modelling and design language but also an **implementation** language as CPAL models can be executed directly on the target platforms with a real-time interpretation engine.

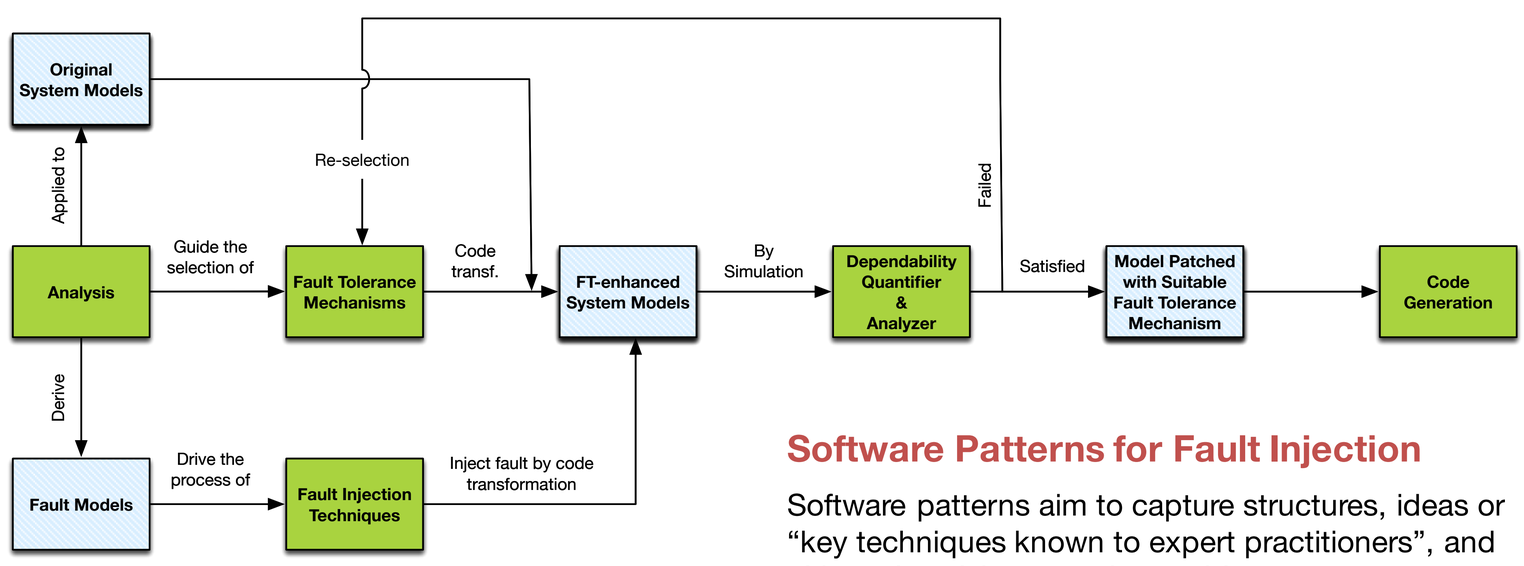
```
processdef P (params) {
   common {
      code
   }

   state Warning {
      code
   }
   on (cond) {code} to Alarm_Mode;
   after (time) if (cond) to Normal_Mode;

   finally {
      code
   }
}

process P: inst [period, offset][cond] (args);

@cpal: time: inst {
        annotation code
}
```

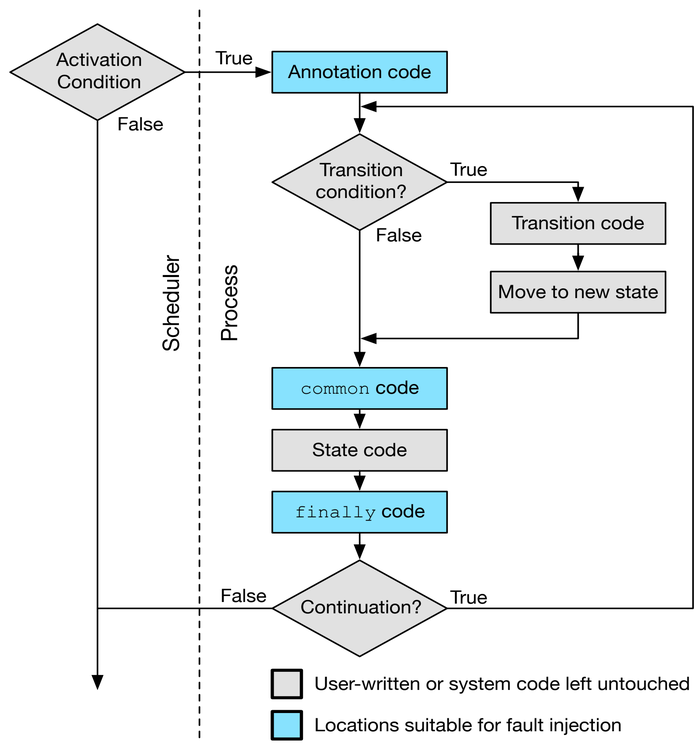


Fault category

Four categories of faults related to CPS have been identified, and the quality used to distinguish one type of fault from another is mostly the **entity** it affects rather than other attributes—for instance, the fault being transient or permanent.

- **Global state**: shared state information are typically held in a set of global variables, such as memory-mapped I/O ports and inter-process communication channels. By corrupting global objects, we can model spontaneous output actuations and communication failures.
- **Activation arguments**: model faults affecting local execution of one or more process instances at the interface level by corrupting its in or out arguments. The introspection mechanism provided by CPAL can be used to target a particular process instance.
- **Local instance variables**: model faults inner to a process instance, with a per-activation lifespan or persistent across different activations of the same process instance. No direct impact on other processes in the system.
- **Control flow disruption**: CPAL processes are implemented as well-structured, hierarchical FSM. State transition conditions are honored prior to the execution of any state code. Faults injected before the evaluation of transition conditions may lead the affected process instance to immediately reach a faulty state upon activation.

CPAL elementary execution step



Software Patterns for Fault Injection

Software patterns aim to capture structures, ideas or "key techniques known to expert practitioners", and ultimately solving recurring problems.

- **External process(es)**: one or more processes are dedicated to fault injection. In the latter case, associate a separate injector process for each target process
  - \+ Keep a clean boundary between the normal behavior of a system and its fault profile.
  - \- Restricted flexibility as it offers limited access to process inner state and hence exhibits coarse injection granularity
  - \- Overhead could be significant, in terms of injector activation frequency or increased number of processes in the system.
- **Pre/post conditions**: CPAL supports to specify code common to different states of a process by means of the common{} and finally{} blocks, which are executed before and after the state code respectively. They can be adopted for fault injection as well.
  - \+ Finer granularity as instance-specific fault injection is possible and allow access to both activation arguments and local variables
  - \+ Lower overhead because injection code run only when processes are activated
  - \- Limited impact on control flow
- **Annotation-based**: leverage existing annotation mechanism provided in CPAL for fault injection. It can be specified either at the process or instance level for non-functional properties
  - \+ Clean separation of concern
  - \+ Medium granularity of fault injection
  - \+ Low overhead as well since fault injection code runs on demand
  - \- Effective for global state and control flow, but not suitable for activation arguments and local instance variables.

	Global state	Act. args	Local varis.	Control flow
External process(es)	✓			✓
Pre/post conditions	✓	✓	✓	
Annotation -based	✓			✓

Further Development

Extend the software patterns to support injection of **timing** faults, another type of faults that jeopardizes the system correctness. This step will be continued with **automatic code generation** for fault injection.