Software Patterns for Fault Injection in CPS Engineering

Nicolas Navet*, Ivan Cibrario Bertolotti†, Tingting Hu*
* University of Luxembourg, 6 Avenue de la Fonte, L-4364 Esch-sur-Alzette, Luxembourg
Email: {nicolas.navet, tingting.hu}@uni.lu
†National Research Council of Italy – IEIIT, c.so Duca degli Abruzzi 24, I-10129 Torino, Italy
Email: ivan.cibrario@ieiit.cnr.it

***Abstract*—Software fault injection is a powerful technique to evaluate the robustness of an application and guide in the choice of fault-tolerant mechanisms. It however requires a lot of time and know-how to be properly implemented, which severely hinders its applicability. We believe software fault injection can be made more "affordable" by automating it and have it integrated within a model-driven engineering design flow. We first propose in this paper a framework supporting these objectives. Then, illustrating on the domain-specific language CPAL, we present injection patterns that can be embedded in the application code and discuss the types of faults each supports, as well as implementation issues.**

***Index Terms*—Software Fault Injection, Model-Driven Engineering, Software Patterns, Industrial Cyber-Physical Systems, CPAL.**

I. Introduction

A. Context of the study

If the term Cyber-Physical Systems (CPS) has been a buzzword over the last 10 years, CPS such as smart grids are starting to be deployed at large scale and will have a profound and pervasive impact on human societies. In the current state of the technologies, the features CPS offer, the time it takes to bring them to the market and their correctness depend importantly on our capability to efficiently write software, which still remains a challenge. In that regard, 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. Most CPS are, to some extent, subject to dependability constraints, which implies the use of verification techniques such as analytic and simulation models, and fault injection be it on models, prototypes or the deployed systems. A central challenge today in MDE is to make it possible to seamlessly integrate the verification activity within the design flow [1] and to fully, or partially, automate it. In the development of CPS with dependability constraints, a key dependability assessment technique is fault injection [2], which can be implemented both in hardware and software, the latter being the focus of the paper. In order to speed up the occurrence of errors, thus making the assessment more effective, often the *effects* of faults are injected, rather than the faults themselves, using an approach known as *error injection*. The software-implemented fault injection (SWIFI) method discussed in the following belongs to this category.

B. Contribution of the paper

Fault injection is certainly a powerful dependability assessment technique but it is time-consuming and requires extensive know-how to implement it correctly and to determine the verification coverage of an experiment. Our objective is to automate software fault injection as far as possible and have it integrated within a MDE design flow. This encompasses to solve several sub-problems such as defining the set of experiments to perform in order to achieve a certain evaluation goal (e.g., verification of the effectiveness of error detection mechanisms) and instrumenting the original code with fault injection patterns. This work is a contribution in that direction, we propose here a set of software patterns to implement fault injection in languages, or language extensions, like StateFlow®, CPAL [3] or Mbeddr [4] that natively support Finite State Machines (FSMs). Specifically, our work targets CPAL which is a representative of DSL for embedded systems designed for MDE. The software patterns proposed aim to capture structures, ideas, or "key techniques known to expert practitioners" (see the seminal paper [5]), and ultimately solve recurring problems. The code implementing the patterns discussed in this work is freely available in the samples of the CPAL distribution available on-line at http://www.designcps.com. This work takes place in the broader context of the development of a fault tolerance and fault injection framework relying on automated code instrumentation which is our ongoing work (see Section III and [6]).

C. Existing work

There is large body of literature on software fault injection that has been created over the last 2 decades. The reader is referred to [7] for a recent and comprehensive survey. Directly relevant to this study are the works in [8] and [9], which propose MDE-based software injection frameworks but targeting each a specific application domain. Other work, described in [10], aims at the validation of IEC 61131 software by means of fault injection. In further studies, like [11], the

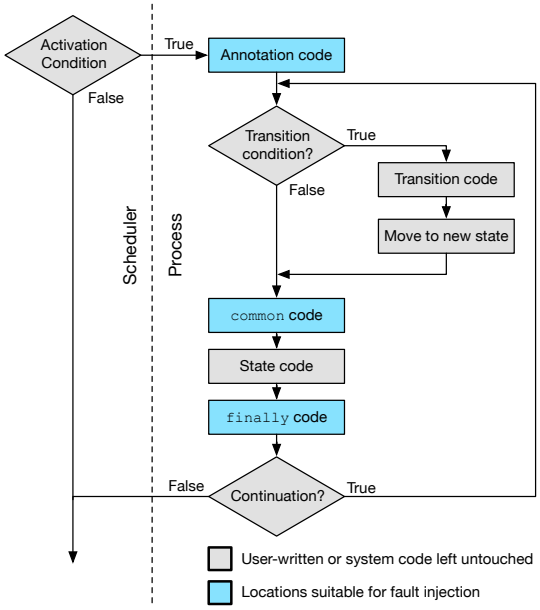

Fig. 1. CPAL process scheduling and elementary execution step.

semi-automatic generation of test cases is also foreseen in the same context, using data taken from the system specification and extracting control flow information from the software. However, both works focus on Programmable Logic Controller (PLC) rather than CPS software, which requires a more general programming paradigm. Moreover, they envisage fault injection only at the boundary between the controlled process and the control software, thus mainly considering hardware component failures rather than faults internal to the control software itself.

The paper is organized as follows. After giving a short introduction about CPAL and its role for embedded software engineering in Section II, the proposed framework and its fault injection patterns are discussed in Sections III and IV. Section V concludes the paper.

II. CPAL for Embedded Software Engineering

The Cyber-Physical Action Language (CPAL, see [3], [12]) is a new domain-specific language that provides 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. CPAL is a modelling and design language but it is also an implementation language as CPAL models can be interpreted on a real-time execution engine. In CPAL verification can be done by schedulability analysis, timing-accurate simulation and runtime observation. In particular, CPAL offers a number of introspection mechanisms [3] to implement error-detection (e.g., overload) and adaptive behaviours (e.g., control laws).

A. Processes as recurrent FSMs

In CPAL, *processes* can be seen as functions that can be activated with a user-defined period and offset relationships, or upon the occurrence of some external events. Active processes

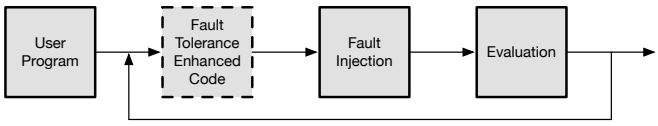

Fig. 2. Software fault injection framework based on code instrumentation.

are scheduled according to a chosen scheduling policy that is specified in a timing annotation. The logic of a process is defined as a Finite State Machine (FSM), possibly organized in a hierarchical manner, where code can be executed in the states, or upon the firing of transitions. The snippet of code below illustrates the definition of a process type and the declaration of an instance of that type.

```
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
}
```

B. Process elementary execution step

When activated by the scheduler a process resumes in the state in which it was at the end of the previous execution. It is then first checked whether a transition can be triggered or not (see Fig. 1). Next, optionally, a block of code common to all states is executed (`common` code), then comes the execution of the code of the current state and, finally, another optional block of code common to all states is executed (`finally` code). The process can continue its execution (if `self.continue` is set to `true`) or relinquish the CPU. These steps executed in sequence, as illustrated in Fig. 1, form an *elementary execution step* of the process.

III. A Framework For Automating Fault-Tolerance and Fault Injection

Fig. 2 depicts the framework we propose to enhance the dependability of CPS. Most importantly, the proposed workflow will be as much as possible automated and "invisible" to the user by keeping a clear separation between the application code and the fault-injection / fault-tolerance code. It includes the following modules:

1) the original program written by the programmer,
2) an optional component that adds fault-tolerance mechanisms,
3) a separate module that carries out software fault injection,

4) a last module that performs dependability assessment of the system under test. If requirements are not met, new choices will be made for the implemented fault-tolerance mechanisms.

Automated code instrumenting, a technique already supported by the CPAL development flow for test coverage estimation through the `cpal2x` utility, will be adopted to automate steps 2) and 3). Moreover, as commonly done [2], a supervisor will also be implemented to orchestrate and automate the overall process.

For what concerns fault-tolerance enhancement, we will provide a library of the most important fault-tolerance mechanisms, such as N-version programming (NVP) and Triple Module Redundancy (TMR) for fault masking, and control-flow check and CRC check for error detection. Each individual mechanism can be enabled through a code annotation mechanism defining a small internal DSL, similar to what already exists in CPAL for the timing dimension. The feasibility of this approach has been verified in [6] for the NVP pattern. The types of faults that can occur (e.g., fuzzing, bit-flipping, stuck-at, timing faults, etc), their location and when they occur (e.g., occurrence of an event, randomly, etc) will be an input provided by the users. Either users will provide an application-specific fault model, or they will instantiate generic models available in the framework, for instance, based on data collected at run-time. Typically, bit-flipping faults affecting global data with a probability function of the data size belong to the latter category of models.

The dependability objective can be stated in various ways such as how robust the system is to faults (e.g., probability that an error is detected and that the system is able to recover from it), or with more specific criteria, like a threshold on the probability that the system exhibits a specific failure mode (e.g., "fail-safe", "crash"). The process depicted in Fig. 2 will be iterated until the system reaches the desired dependability objective. The final outcome, that is the code patched with appropriate fault tolerance technique(s), can be used for further model-based evaluation such as performability (*i.e.*, joint evaluation of performance and availability) or for direct use in the deployed system.

IV. Patterns for Fault Injection

This section provides information on the fault injection block shown in Fig. 2, presenting the categories of fault that can be injected and discussing which code patterns can be used to this purpose. Since, as described in Section II, processes and their elementary execution steps are a key concept in CPAL concurrency, stipulating that fault injection operates at the elementary execution step level comes naturally. Moreover, the beginning and end of an execution step are also the points at which a CPAL process imports information from other processes and its execution environment, and makes its results observable.

Another important assumption is that fault injection code patterns lie at the same conceptual level as the code they are applied to, and hence, are written in CPAL itself without altering language semantics. This approach has the advantage of making the system model completely self-contained for what concerns the language, including any aspects related to fault tolerance, and also simplifies automatic code generation. At the same time, it does not preclude optimized (but semantically equivalent) implementations from being carried out, for instance, within the CPAL execution engine.

A. Fault categories

From the point of view of injection patterns, the quality that distinguishes one type of fault from another is mostly the *entity* it affects rather than other attributes—for instance, the fault being transient or permanent. According to this reasoning, the patterns devised so far are able to support four different categories of fault.

1) Global state: CPAL processes typically hold shared state information in a set of global variables that, from the low-level implementation point of view, reside in a pool of RAM allocated at link time. Hence, corrupting those variables can effectively model various kinds of memory cell failure. Since memory-mapped I/O ports and inter-process communication channels are often represented as global CPAL objects, tampering with them can model spontaneous output actuations and communication failures, too. On the other hand, granularity is coarse because faults affect all process instances activated after the fault has been injected, as long as the fault persists.

2) Activation arguments: To improve modularity and adhere to sound engineering practice, CPAL processes rarely refer to global variables directly. Instead, they get access to them through `in` and `out` arguments. Arguments are passed by value or by reference, respectively, upon process instance activation, that is, when the activation condition shown in Fig. 1 is met. On the implementation side this typically corresponds to copying the global variables or their address, respectively, onto the process instance stack or private local storage. Therefore, injecting faults on `in` arguments rather than the corresponding global variables conveniently supports the distinction between how different kinds of memory fail (often, off-chip DRAM is used for global variables, whereas on-chip, faster SRAM holds stacks). Furthermore, faults injected into activation arguments affect execution locally, at process instance activation granularity, rather than globally.

3) Local instance variables: Process instances usually make use of local storage that can be volatile, with a per-activation lifespan, or persistent across activations. As for activation arguments, injecting faults in these variables affects only a specific process instance activation with fine granularity.

4) Control flow disruption: This fault category is much harder to model with respect to the previous ones because CPAL, as virtually all high-level programming languages do, keeps all control flow details hidden from programmers. Hence, for instance, there is no way to alter the program counter by means of a language statement. However, the fact that CPAL enforces a rigorous, well-defined structure onto processes—the recurrent, hierarchical FSM organization outlined in Section II—provides a useful surrogate. More

```
processdef A_Proc(in uint32: x)
{
  state First {
    /* Body of the state */
  }
  on (true) to Second;

  state Second {
    /* Body of the state */
  }
  on (true) to First;
  /* Other states and transitions */
}

processdef Injector_A_Proc(
        out uint32: x,
        in Process_Instance: p)
{
  state A_State {
    /* State-specific fault injection */
    if (p.process_state == Process_State.First) {
      x = 2;
    } else
    if (p.process_state == Process_State.Second) {
      x = 3;
    }
  }
  /* Other states and transitions */
}


var uint32: global_variable = 1;

process A_Proc:p1[100ms](global_variable);
process Injector_A_Proc: p1_Injector[100ms](
        global_variable,
        p1);

@cpal:time {
  system.sched_policy  = Scheduling_Policy.NPFP;
  p1_Injector.priority = 1;
  p1.priority          = 0; /* Lower priority */
}
```

Fig. 3. Fault injection in external fault-injector that is specific to each instance of the process. The scheduling parameters are set so as to ensure the injector always runs before the process instance whose inputs will be corrupted.

specifically, since state transition conditions are honored prior to executing any state code (see Fig. 1), a fault injected before their evaluation may lead the affected process instance to immediately reach a faulty state upon activation. Other kinds of control flow disruption, like function return address corruption, cannot be modeled with this approach and most likely require support at a lower level of abstraction, typically within the execution environment.

B. Injection patterns

The sheer sophistication of the CPAL execution model supports three distinct approaches to fault injection as a minimum. All of them are suitable for automation and can be used in combination. Overall, they provide different trade-offs between overhead and ability to support the kinds of fault described in Section IV-A. Besides using one or more *external*, dedicated processes, the CPAL execution logic provides several additional locations suitable for hosting fault injection code, highlighted in Fig. 1.

1) External fault injectors: When using this approach, one or more processes are dedicated to fault injection, with the advantage of keeping a clean boundary between the normal behavior of a system and its fault profile. Moreover, this way of modeling corresponds to a *centralized* fault injection mechanism, even when modeling a distributed system, which is close to practice. However, it could hamper flexibility because it offers limited access to process state and exhibit coarse injection granularity. The ability of fault injection processes to affect individual process instances when multiple instances are released in parallel is also possible with the Non-Preemptive Fixed Priority (NPFP) policy, extending the approach shown in Fig. 3.

However, there may be significant overheads, especially when considering a *single* periodic injector process, because it should be executed before the activation of every possible process instance. Even when focusing only on in-phase, periodic processes, the injector activation frequency may therefore become very high unless process periods are harmonic.

Considering activation offsets and event-triggered processes further increases overheads and, in the second case, may make the approach unfeasible. Adopting a separate injector process for each target process alleviates this issue, because individual injector's activation conditions can be optimized, at the cost of doubling the number of processes in the system. An example illustrating this way of injecting faults is shown in Fig. 3.

It is worth remarking that, besides normal data types, CPAL also supports a data type named `Process_Instance`, which keeps record of all the characteristics of a process instance that can be queried at run-time, including its current state indicated by the `process_state` field.

2) Fault injection as pre/post conditions: CPAL supports per-process, state-independent code blocks (called `common` and `finally` blocks in Fig. 1) that are executed before and after state-specific code upon process activation. Since they are subject to the same scoping rules as state-specific code, they can access not only global variables, but also activation arguments and local variables. Instance-specific fault injection can be modeled, too, because they can query the unique instance identifier `self.pid`. With respect to external injectors, overheads are more limited because fault injection code runs "on-demand", only as often as processes are activated.

On the other hand, fault injection code is *internal* to processes in this case. Since `common` and `finally` blocks are user-accessible, this approach brings the disadvantage of mixing regular and fault injection code within the same syntactic elements, although the use of named blocks as done in Fig. 4 alleviates this drawback to some extent. It should also be noted that shadowing may somewhat impair access to global variables (when a local variable with the same name exists). More importantly, this method cannot directly affect state transitions, and hence, alter control flow. This is because, as also shown in Fig. 1, fault injection code is executed after transition conditions are evaluated.

The code snippet shown in Fig. 4 demonstrates an example of fault injection in terms of pre/post-condition. Input arguments are corrupted before they are used (in the `common` block), while faults are injected into output arguments after they have been updated (in the `finally` block).

```
processdef A_Proc(in uint32: x, out uint32: y)
{
  var uint32: x_corrupted = x;

  common {
    fault_injection: {
      if (self.process_state == Process_State.First){
        corrupt_uint32_fault_type_A(x_corrupted, 4);
      }
    }

    /* Regular code */
  }

  state First {
    /* Replace x by x_corrupted */
  }
  on (true) to Second;

  state Second {
    /* Replace x by x_corrupted */
  }
  on (true) to First;
  /* Other states and transitions */

  finally {
    /* Regular code */

    fault_injection: {
      corrupt_uint32_fault_type_B(y, 4, 7);
    }
  }
}
```

Fig. 4. Fault injection in pre/post-conditions within named block to keep a clean boundary with respect to regular code. Inputs are corrupted in a state-dependent manner and before being used, while outputs are corrupted after being updated.

```
processdef A_Proc(in uint32: x, out uint32: y)
{
  state A_State {
    /* Value of y is a function of x */
  }
  /* Other states and transitions */
}

var uint32: global_variable = 1;

process A_Proc:p1[100ms](global_variable);

/* The annotation is executed each time before p1 */
@cpal:time:p1 {
  /* Here the fault occurs within a given interval */
  if (1s500ms <= time64.time() <= 1s750ms) {
    global_variable = 2;
  }
}
```

Fig. 5. Fault injection in an annotation. Here we know the global variable will be updated after the perturbation interval, for instance as it is mapped to an I/O, if not we should restore the original value.

Moreover, fault-injection can be performed at a fine granularity, for instance, in a state-specific way. Named block, e.g. `fault_injection` shown in Fig. 4, is the syntactic sugar that can be adopted to group fault injection code and keep a well-defined boundary with respect to regular code.

As the input arguments of CPAL processes are read-only, local copy (or copies) that represents the corrupted input(s) have to be created, and any reference to the original input argument(s) within the process are replaced by the copy (or copies). Besides, dedicated functions can be defined for different types of fault injection, depending on the data type of the variable(s) to be corrupted. For instance, as shown in the illustrative example of Fig. 4, the input argument is set to a fixed faulty value, whereas the output argument is corrupted by a random error in a pre-defined range, here 4 to 7.

3) Annotation-based fault injection: The standard CPAL language natively supports *annotations* to express non-functional properties of a program and cleanly isolate them from functional properties. The same mechanism can be leveraged for fault injection, by envisaging an annotation whose code runs between process instance activation and transition condition evaluation, as illustrated by the *annotation code* block in Fig. 1.

Like for external injectors, this approach has access to global state and can directly affect flow control with even finer granularity because CPAL supports instance-specific, besides process-specific, annotations. Moreover, as for pre/post conditions, fault injection code runs on demand and introduces limited overhead. However, the fact that annotations are currently defined at the same hierarchical level as the process they refer to precludes them from accessing activation arguments and local variables.

In the current version of CPAL, the proposed approach should be implemented through a timing annotation as shown in Fig. 5. However, to enable a better separation of concerns, the language should be extended to support a dedicated annotation such as `@cpal:dependability`.

Table I summarizes the discussion by showing which fault categories the injection patterns just described can support.

TABLE I
MAPPING OF FAULT CATEGORIES ONTO INJECTION MECHANISMS

Mechanism \ Fault category	Global state	Act. args.	Local vars.	Control flow
External process(es)	✓			✓
Pre/post conditions	✓	✓	✓	
Annotation-based	✓			✓

V. CONCLUSION

This paper has shown how software-implemented fault injection can profitably be implemented solely at the DSL level and soundly assist CPS development, without breaking the continuity between the verification activity and the design flow. More specifically, using CPAL as a case study, several different injection patterns have been presented and discussed, showing how they can effectively introduce *data errors*, as defined in [7], at different locations. Up to a more limited extent, some of the patterns can also mimick *code changes* by affecting state transitions of CPAL processes. Even more importantly, all patterns can be applied by means of automatic code generation. In this way, low-level pattern-related details can be kept hidden from end-users, who will instead work at a higher level of abstraction and focus on specifying the types of faults that are relevant for their systems.

Besides data error, another type of faults that jeopardizes the system correctness are timing faults such as abnormal

execution times, priority inversions or jitters. The patterns described in this study can be extended to handle timing faults, which can be done trough timing annotations at the global level or locally within a process. Although the extent of the work has mainly considered fault injection patterns so far, preliminary results look promising and lead us to consider the implementation of the complete fault-tolerance and fault injection framework outlined in this paper as a future work.

REFERENCES

[1] G. Brau, J. Hugues, and N. Navet, "A contract-based approach to support goal-driven analysis," in *Proc. 18th IEEE International Symposium on Real-Time Distributed Computing (ISORC)*. Los Alamitos, CA, USA: IEEE Computer Society, 2015, pp. 236–243.

[2] M.-C. Hsueh, T. K. Tsai, and R. K. Iyer, "Fault injection techniques and tools," *Computer*, vol. 30, no. 4, pp. 75–82, Apr. 1997.

[3] N. Navet and L. Fejoz, "CPAL: High-level abstractions for safe embedded systems," in *Proc. 16th Workshop on Domain-Specific Modeling*, ser. DSM'16. Amsterdam, Netherlands: ACM, 2016.

[4] M. Voelter, D. Ratiu, B. Schaetz, and B. Kolb, "Mbeddr: An extensible C-based programming language and IDE for embedded systems," in *Proc. 3rd Annual Conference on Systems, Programming, and Applications: Software for Humanity*, ser. SPLASH '12, 2012, pp. 121–140.

[5] J. O. Coplien, "Idioms and patterns as architectural literature," *Software, IEEE*, vol. 14, no. 1, pp. 36–42, 1997.

[6] T. Hu, I. Cibrario Bertolotti, and N. Navet, "Towards seamless integration of N-Version Programming in model-based design," in *Proc. 22nd IEEE International Conference on Emerging Technologies and Factory Automation (ETFA)*, Sep. 2017, to appear.

[7] R. Natella, D. Cotroneo, and H. S. Madeira, "Assessing dependability with software fault injection: A survey," *ACM Comput. Surv.*, vol. 48, no. 3, pp. 44:1–44:55, Feb. 2016.

[8] C. Buckl, D. Sojer, and A. Knoll, "FTOS: Model-driven development of fault-tolerant automation systems," in *Proc. 15th IEEE Conference on Emerging Technologies and Factory Automation (ETFA)*, Sep. 2010, pp. 1–8.

[9] I. Ayestaran, C. F. Nicolas, J. Perez, A. Larrucea, and P. Puschner, "Modeling and simulated fault injection for time-triggered safety-critical embedded systems," in *Proc. 17th IEEE International Symposium on Object/Component/Service-Oriented Real-Time Distributed Computing*, Jun. 2014, pp. 180–187.

[10] M. Marcos, E. Estévez, N. Iriondo, and D. Orive, "Analysis and validation of IEC 61131-3 applications using a MDE approach," in *Proc. 15th IEEE Conference on Emerging Technologies and Factory Automation (ETFA)*, Sep. 2010, pp. 1–8.

[11] S. Rösch, D. Tikhonov, D. Schütz, and B. Vogel-Heuser, "Model-based testing of PLC software: test of plants' reliability by using fault injection on component level," *IFAC Proceedings Volumes*, vol. 47, no. 3, pp. 3509–3515, 2014, 19th IFAC World Congress.

[12] N. Navet and L. Fejoz, *The CPAL Programming Language*, version 1.19 ed., June 2017. [Online]. Available: https://www.designcps.com/wp-content/uploads/cpal-intro.pdf