Property-Based Fault Injection: A Novel Approach to Model-Based Fault Injection for Safety Critical Systems

Athira Varma Jayakumar(✉) and Carl Elks

Electrical and Computer Engineering Department, Virginia Commonwealth University, Richmond, VA, USA
{jayakumarav,crelks}@vcu.edu

Abstract. With the recent popularity of model-based design and verification (MBDE), fault injection testing at the functional model level is gaining significant interest. The reason for this interest is it aids in detecting design errors and incorrect requirements on fault detection and tolerance features, very early in the development lifecycle. This is evidenced by the fact that functional safety standards like IEC 61508 and ISO 26262 identify fault injection testing as a highly recommended technique for SIL-3 and SIL-4. The main challenges to date with model-based fault injection are lack of completeness in the fault injection space, semi-manual integration and insertion of fault injection modules into the models and manual identification of fault activation conditions. The work presented in this paper describes a novel model-based fault injection technique that is *property-based* and applies formal model checking verification methods at the functional model level of design thereby guaranteeing a near-exhaustive state, input and fault space coverage. This method also introduces the usage of properties and model checking capabilities to automate the identification of fault activation conditions for all the faults within the fault space. We describe the workflow and implementation of the property-based Fault injection using Simulink Design Verifier and its application on the functional model of a representative safety-critical system.

Keywords: Fault injection · Fault tolerance assessment · Model-based fault injection · Safety-critical systems · Model-checking

1 Introduction

With technology advancements in semiconductor electronics, communication networks, system-on-chip architectures we have been witnessing continuing high density integration of CPS technology. At the same time we are seeing an increase in software intensive systems for critical applications that take advantage of the capabilities of today's highly integrated devices. To identify and understand potential failures and their consequences, the use of systematic approaches for assuring digital system safety (dependability) is gaining acceptance across many industries from process control, nuclear, automotive, to aerospace. These systematic approaches include methods like safety case analysis

M. Zeller and K. Höfig (Eds.): IMBSA 2020, LNCS 12297, pp. 115–129, 2020.
https://doi.org/10.1007/978-3-030-58920-2_8

methodologies, STPA, STAMP, and formal design assurance methods [1]. Almost all of these systematic approaches require some form of analysis or study of failures, faults, and losses within a system context. Dependability evaluation involves the study of failures and errors and their potential impact on system attributes such as reliability, safety and security. This is evidenced by the fact that functional safety standards like IEC 61508 and ISO 26262 identify fault injection testing as a highly recommended validation technique for SILs 3 and 4.

Fault Injection (FI) is defined as a dependability evaluation technique based on the realization of formal controlled validation experiments in which system behavior is observed while faults are explicitly induced in the system by the deliberate introduction (injection) of faults [2]. At a broad stance, contemporary fault injection approaches fall into 3 categories; physical fault injection, simulation-based fault injection, and SWIFI (Software Implemented Fault Injection). While all of these contemporary approaches to fault injection are needed for assuring dependability and safety case arguments, we suggest a new class of fault injection should be defined by the dependability community – *model based fault injection.* With the recent popularity of model-based design and development (MBDE) and testing, fault injection testing at the functional model level is gaining significant interest. The reason for this interest is it aids in detecting design flaws, omissions and incorrect requirements that impact hazard mitigation very early in the lifecycle development process.

The work presented in this paper describes a model-based fault injection framework that is implemented in the Mathworks Simulink environment. The innovation of the work is that the fault injection technique is *property based* and applies formal model checking at the functional model level of design thereby guaranteeing a near-exhaustive state, input and fault space coverage. This work is noteworthy in that it links fault injection to properties (safety, functional, and liveness) that are vital for making safety case arguments in the presence of hazards, failures and faults. This work also solves the challenge of manually identifying fault activation conditions during fault injection by employing model checking tool to automate it. Our fault injection framework implements a comprehensive insertion of fault modules throughout the system functional model thus enabling an exhaustive fault location coverage.

2 Background and Challenges

Fault injection is often characterized by its statistical nature, meaning that statistical models are used to govern the fault injection experiment [3]. The classical fault injection approach involves iteratively conducting a set of tests (called trials) to inject faults into the system and observing the response of the system. As such, the fault space often involves the dimensions injection time (t), fault location (l), fault type (f_m) as sampled from fault classes, fault value (v) and fault duration (Δ). An extremely important metric for assessment of safety-critical systems is fault coverage, C which denotes the conditional probability that the system detects or corrects a fault given that a fault is present in the system. Due to the multi-variable nature of the statistical experiments, fault injection experiments without a-priori knowledge of what types of fault to inject, where to inject, and at what time to inject becomes very combinatorically challenging, which can lead

to very large numbers of fault injection experiments. System safety and reliability are highly sensitive to fault coverage and therefore high levels of fault coverage estimation is often required for such systems [4]. Another major challenge with simulation-based FI is the manual identification of fault activation conditions which is very essential for conducting effective fault injections and avoiding no-response faults. The Property-based FI proposed in this paper is capable of addressing these challenges.

3 Related Work

In the last decade, FI community has started gaining interest on the application of formal methods to fault injection and to the analysis of fault tolerance mechanisms. Few works that extend formal verification techniques like model checking, assertion based verification and symbolic analysis to fault injection are discussed below. Scott et al. [5] present a methodology called ABVFI which is fault injection based on assertion based verification, that checks for critical properties at the hardware level by embedding assertion statements into the hardware design (RTL). Krautz et al. [6] used formal verification to exhaustively measure the fault coverage of a VHDL design. Symbolic simulation of sequential circuits is performed by creating Binary Decision Diagrams of the circuit's state space. Leveugle [7] introduced the new approach of combining property with mutation of the circuits to conduct fault injection experiments on circuits. Critical properties are checked for any violations in the mutated circuits thereby helping in identifying the undesirable effects of multiple faults in the circuit. Daniel et al. [8] were the first to demonstrate the effectiveness of symbolic fault injection on a SIHFT application in Java platform. This approach involved the injection of symbolic faults during the symbolic execution of the software. Another work by Vedder et al. [9] that combines property-based testing with fault injection enables automatic generation of testcases from specified system properties and verifying them in the presence of faults. Most of these works combining fault injection with formal verification are targeted for hardware design level. Whereas this study is targeted to analyze the feasibility and effectiveness of formal verification for fault injection in a model-based design environment.

Few works like [10] and [11] have proven the usage of model checking and fault injection to automate FMEA and deviation analysis on Behavior Tree models and NuSMV models respectively. xSAP tool [12] offers a platform to formally conduct faults/safety analysis, but necessitates the system model to be represented in NuSMV. We extend on these works by introducing the use of exhaustive model checking for fault injection and fault behavior analysis on executable functional models in Simulink capable of progressing into code generation. These Simulink behavioral models are deterministic functional/logical models with no probabilistic aspects. While studies [13] and [14] describe the usage of Design Verifier model checker to formally prove safety properties in the presence of faults introduced in the functional models, we go beyond these works by addressing the practical problem of identifying fault activation conditions during fault injection. Faults even though triggered may not be activated if other input/state conditions are not satisfied. Inactivated faults could cause the model checker to falsely validate the safety properties. This practical problem is systematically addressed using model checking in this paper. In addition, while other studies including [13] and [14] selectively

introduce failure modes in the model to capture the various ways in which the system components malfunction, our work implements exhaustive fault saboteur insertion, thus allowing for an exhaustive fault behavior analysis by the model checker. An automation script that is part of our FI framework facilitates insertion of fault saboteurs throughout all signal lines within the model, thus ensuring complete fault location coverage. The framework also enables an extensive and diverse fault model selection that consists of permanent, transient and delay faults [15].

4 Property-Based Fault Injection

The conceptual idea of property proof based fault injection is shown in Fig. 2. The basic premise of our approach is to use the power of *model checking* to overcome some of the challenges associated with classical fault injection methods – namely the burden of executing large numbers of FI combinatorial experiments and finding fault activation conditions. This approach is complementary to classical fault injection methods in that it identifies potential problems as early as possible in the design process. Model checking is a formal verification technique that mathematically verifies the validity of critical properties of a model of a system through systematic state exploration. Unlike testing, where the input and expected output vectors have to be fed into the system, there is no need to feed in input sequences for model checking. It is a verification procedure that involves an *exhaustive search* of the state and input space of the design to ensure that a system always satisfies specific property under given constraints.

Fig. 1. Classical fault injection vs property based fault injection

To understand the difference between property-based FI and classical statistical-based FI, we consider the FI experiment space exploration achieved with each of these methods as shown in Fig. 1. Classical FI experiment covers a single point within the FI experiment space of the system, per experimental trial. As such, with classical FI there are always points in the FI experiment space (input/fault/state space points) that are not covered, which could overlook a faulty part of the design to go unnoticed. In contrast, fault injection based on property proving works at the property level. Given a safety property and a functional model, property based FI exhaustively searches the entire fault, input and state space for safety property violations, by covering all possible faults in all possible input and state conditions – for that given property.

Referring to Fig. 2, the fault tolerance properties and fault activation conditions are specified in a formal language accepted by the chosen model-checker. Fault tolerance property denotes critical system behavior or safety property that needs to be met by the system outputs in the presence of faults. Fault activation conditions specify the input/state sequences that cause the injected fault to manifest as error and propagate within the system design. To activate a fault, the necessary input/state preconditions are specified as constraints/assumptions to the model checker. Under the given preconditions, the model-checker systematically explores the state, input and fault space of the system functional model and mathematically verifies that a fault from the representative fault space does not violate the fault tolerance property for any given state and input sequence. While maintaining the assumptions, if any violation of the property is encountered, the model checker falsifies the property and generates a counterexample showing the input and fault vector that caused the safety property to fail.

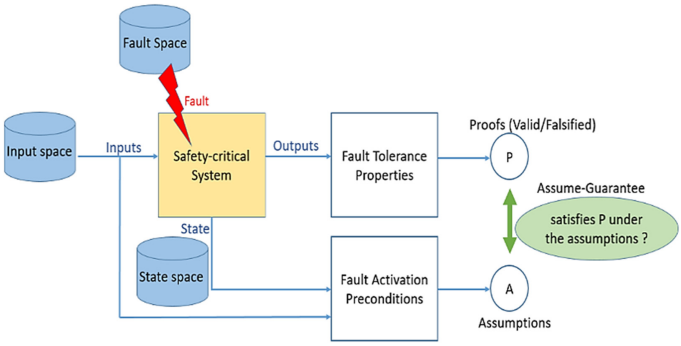

Fig. 2. Fault injection with property proving

We define property based fault injection as: *Let system functional model be represented as five-tuple* $M = \langle I, O, S, S_0, T \rangle$, *where I-set of inputs, O-set of outputs, S-set of states,* $S_0 \subseteq S$ *– set of initial states, T-S X S - set of transitions.*

We say a fault tolerance property P_{FT} *for the model M is satisfied if every state reachable from the initial state* s_0 *(by following all possible transition relations within T) in the Model M satisfies the fault tolerance property* $P_{FT,}$ *in the presence of fault* f_i *taken from the fault space F and the fault activation assumptions* A_{fa}.

$$M \vDash P_{FT} \quad If \quad All\ Reachable_T\ (s_0) \vDash P_{FT} \quad | \quad f_i\ and\ A_{fa} = true \tag{1}$$

Else

$$M \nvDash P_{FT} \quad If \quad Any\ Reachable_T\ (s_0) \nvDash P_{FT} \quad | \quad f_i\ and\ A_{fa} = true \tag{2}$$

where $s_0 \in S_0, f_i \in F$ *and* $A_{fa} =$ *Fault activation conditions as assumptions.*

Figure 3 shows the efficacy of the property based fault injection approach in terms of the coverage of the fault, input and state space. Referring to Fig. 3, the left side enumerates the assumptions made by the model checker during model checking based

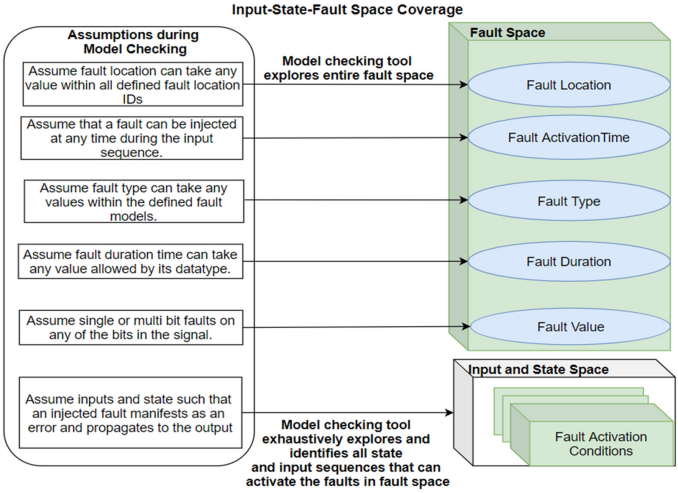

Fig. 3. Property based fault injection - fault-input-state space coverage

on the constraints we specify in the model. During property-based FI, the model checker is allowed to assume the fault location control signal to take any value from the list of defined fault location IDs thus ensuring a complete exploration of the fault location dimension within the fault space. Model checker assumes that the fault can be injected at any time instant during an input/state sequence and for any duration (as permitted by the datatype of the Fault duration control signal). This causes the model checker to completely explore the entire fault activation time and fault duration dimensions of the fault space to find property violation. Model checker explores and covers all fault types within the defined fault model list with constraints to consider only the defined fault models in the fault injection framework. Model checker is allowed to assume single or multi bit faults on any of the bits in the signal thus causing the model checker to completely explore and cover the fault value dimension of the fault space. Thereby, the model checker exhaustively explores the entire input and state space and identifies all possible input and state sequences that can activate the considered faults in the fault space. In this manner, each fault tolerance or safety property verification completely covers the corresponding fault activation space from within the entire input and state space.

5 Implementation of Property-Based FI Using Simulink Design Verifier

We implement the proposed property-based Fault injection technique on Simulink behavioral models using Simulink Design Verifier (SDV) property prover tool [16]. The workflow for realizing the Property based Fault injection in Simulink is given below in Fig. 4. There are few preparation steps (steps 1, 2 and 3) to be performed before starting the property proving within Simulink Design Verifier (DV).

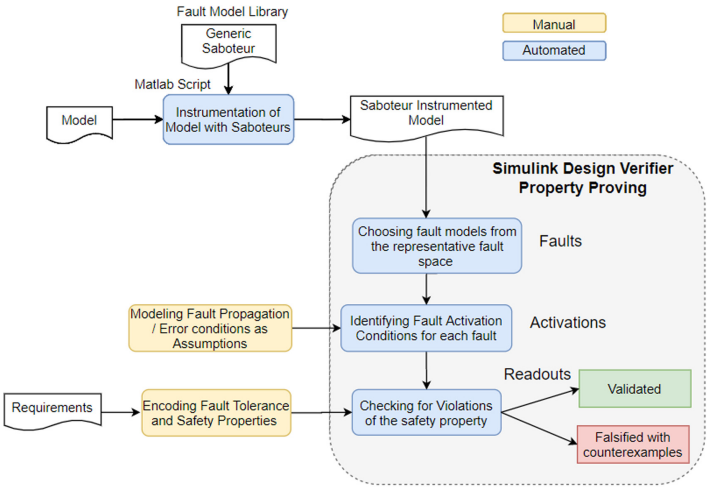

Fig. 4. Property based fault injection workflow

1. **Instrumentation of Model with Saboteurs** - The process starts with an automated insertion of a generic saboteur into all signals in the given test model. This automated saboteur insertion framework is described in detail in [15]. In the model-based context, a 'Saboteur' can be defined as a special module added between signal drivers and receivers in the model-based design, which when activated alters the value or timing characteristics of the signals, thereby simulating faults for Fault Injection [17]. The saboteur in the considered FI framework [15] is a generic one and supports multiple fault models. The saboteur instrumented model generated in this process is further used for the property based fault injection process.
2. **Modeling Fault propagation/Error conditions as assumptions** – With the automatic exhaustive saboteur insertion within the model, we end up with a very large set of fault locations within the model. Thus manually finding the activation condition for each fault location could be an extremely time-consuming process. It would require a complete analysis of the model and identifying all possible input scenarios that can activate the faults – which is infeasible for system models commensurate of practical system scales. The approach we employ to solve this problem is to use the model checking tool to automatically identify the fault activation conditions (e.g. the input conditions) for any given fault. In this method, instead of manually exploring the entire input space to identify and specify the input conditions to activate the faults, the tester has to specify the state/output condition that indicates an error or system failure as an assumption that becomes true after a fault is injected. The error propagation scenario is modeled as an assumption using Simulink Design Verifier blocks.
3. **Encoding Fault Tolerance and Safety Properties** - Finally, the fault tolerance properties derived from the safety requirements of the system that needs to be validated have to be modeled using the proof objective blocks in Simulink DV library.

After completing all the above preparation steps 1, 2 and 3, property proving is executed in Simulink Design Verifier as described in the below steps 4, 5 and 6.

4. **Choosing fault models from the representative fault space** - When property proving is initiated on the saboteur inserted model, Simulink DV selects fault locations/signals from the entire set of available fault locations and chooses fault models from all the available ones implemented in the Saboteurs to consider for fault injection. DV model checker proves the safety properties in the presence of different kinds of faults and exhaustively covers the fault space.
5. **Identifying fault activation conditions for each fault** - With the erroneous/fault propagation behavior being specified as an assumption in the model, the Design Verifier explores the entire input space and identifies all the input sequences that can cause the injected fault to manifest as an error and propagate. In addition to relieving the tester from the laborious task of identifying the activation input sequences for each fault, it also ensures a complete fault activation space coverage (from within the input and state space) for each and every faults in the fault space and avoids no-response faults.
6. **Checking for Violations of the safety property** - Finally, the safety properties modeled as proof objectives are proven to be valid or false. Validated properties would mean that the fault tolerance functionality is capable of tolerating all the specified faults (from considered fault space). If the property falsifies, a counterexample is generated which clearly shows the location, duration, type, value and time of injection of the fault that bypassed the tolerance functionality or safety feature.

6 Application of Property Based Fault Injection

Property-based fault injection technique was applied to verify "the fail-stop/fail-fast" semantics and fault tolerant features of a safety-critical digital I&C architecture called *SymPLe* [18] developed for Nuclear Power Plant applications. Referring Fig. 5, SymPLe architecture is an FPGA based architecture comprised of three basic control hierarchies: the global sequencer, local sequencers or tasks, and a complete set of Function Blocks (FB) per task lane. In SymPLe, all executions occur in task lanes. These are independent processing stations where function blocks organized as function block programs are executed. Functions blocks assigned to a task lane are scheduled via a deterministic sequence of task executions. Functions blocks receive data, operate on that data per intended functionality, and provide results in their output registers. The global sequencer is concerned with scheduling of task lanes and the marshaling of data I/O in the architecture. The local sequencers' function is to locally coordinate the triggering of a task lane and marshal data to function blocks while executing. Function Blocks (FB) are the elementary "program or computation" units that SymPLe architecture employs for its program organization or application building.

Fault detection and tolerance mechanisms in SymPLe architecture are implemented at several levels, however, in this case study we evaluated the fault handling mechanisms at the Function Block level. Among several fault detection techniques, hardware redundancy is one of the main fault tolerance strategies implemented at the function block

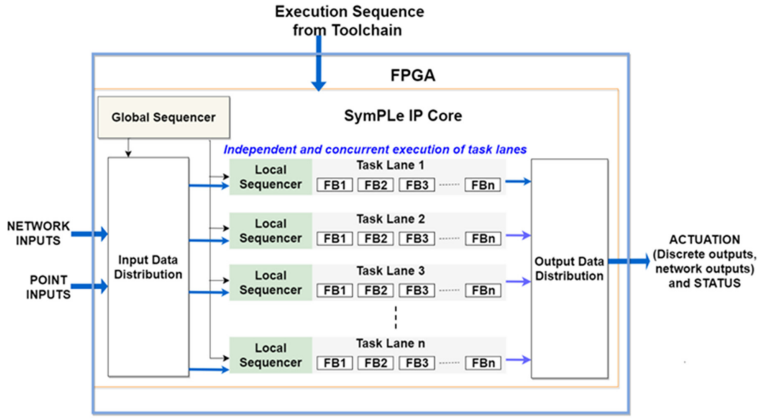

Fig. 5. SymPLe architecture

level. As shown in Fig. 6, each FB is constructed as a duplex system as a means of detecting single and multi-event upsets. The inputs, outputs and states of the duplicated function blocks are compared to report data mismatch/State error that happens due to any kind of faults within the individual FBs. The individual function blocks in the duplex structure are each designed to detect errors in SymPLe's control flow execution and data path execution. An execution error is an error that violates the execution semantics of SymPLe computational model like input register read overflow, output register write overflow, execution timeout and so on. Prolonged execution of a function block could be a sign of malfunction within the function blocks due to transient or permanent physical faults. This execution error is detected and reported using the execution timeout feature of the function block controller. In addition, computation errors that affect logic and arithmetic operations like datatype error, underflow, overflow, division by zero etc. are also detected within the function blocks. Described below are two use case examples of applying the property-based fault injection technique to verify the safety properties related to hardware redundancy and execution timeout feature of the function block array consisting of 32 different function blocks in the SymPLe architecture.

The application of the property-based fault injection method on SymPLe architecture is facilitated by an automated fault injection framework developed in Simulink. This model-based FI framework facilitates FI process on a Simulink model by enabling fault injection control within the functional model and automating saboteur insertion throughout the model. A generic saboteur that implements a diverse set of fault models including 'Single/Multi Bit Flip', 'Stuck at 0', 'Stuck at 1', 'Floating', and Delay faults are automatically inserted on all signal lines within the test model by a Matlab script. The Saboteurs are designed using basic Simulink blocks and encapsulated in a masked subsystem. More details on the Saboteur implementation, fault model and the algorithm of the saboteur insertion script can be found in [15].

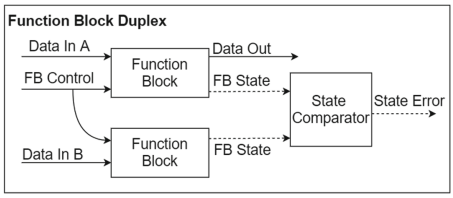

Fig. 6. Duplex function block architecture

6.1 Safety Property 1 - Verifying Redundancy of SymPLe Function Blocks

For verifying the hardware redundancy feature of the function block, the critical property to be considered is that the state comparator module detects anomalies and sends a high on 'state_error' signal when the function blocks in the duplex configuration hold different states. The property for verifying the hardware redundancy feature of the 'ADD' function block in SymPLe architecture is modeled as shown in Fig. 7. The mismatch property states that: *"A difference between the states of the ADD1 and ADD2 function blocks, implies that the 'state_error' signal is true"*.

$$\textbf{\textit{Prop}}\textbf{: } (\textbf{\textit{ADD}}1\textbf{\textit{\_State}} \neq \textbf{\textit{ADD}}2\textbf{\textit{\_State}}) \rightarrow (\textbf{\textit{state\_error}} = \textbf{\textit{true}}) \tag{3}$$

The fault injected in function blocks gets activated when erroneous values propagate to the outputs of the function blocks causing a mismatch between the two function blocks. For each fault location, different preconditions are applicable to ensure the fault is activated and actually propagates far enough to cause a failure at the output. To direct the model checker to consider input conditions that can activate the faults, an assumption condition as stated below is modeled as shown in Fig. 7.

"There is mismatch between the states (which comprises of the inputs read into FBs and FB outputs) of the ADD1 and ADD2 function blocks, within 5 clock cycles after a fault is injected".

$$\textbf{\textit{Assumption}}\textbf{: } (\textbf{\textit{Fault Injected}}) \rightarrow (\textbf{\textit{ADD}}1\textbf{\textit{\_State}} \neq \textbf{\textit{ADD}}2\textbf{\textit{\_State within 5 cycles}}) \tag{4}$$

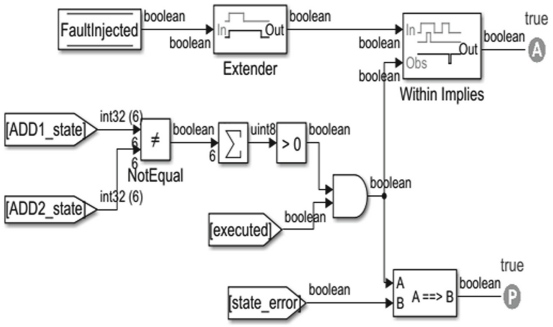

Fig. 7. Property for proving hardware redundancy of Function Blocks

With this assumption in place, the Design verifier explores the entire input space and finds all possible input sequences that can activate a specific fault in the model. Another important assumption that is made is that the *"redundant data inputs to the two function blocks are always equal"*. ***Assumption: DataInA = DataInB***. This is to ensure that the state mismatch between the duplex function blocks is not caused due to the mismatch of inputs to the function blocks, but instead caused due to faults injected within one of the function blocks. Assumptions on the fault injection control signals help to constrain the fault space to a representative set of fault locations and fault models during the FI experiment. The fault location is assumed to be lying within a range of values. The fault type is constrained to take only the defined fault models. Fault injection time, duration, and value parameters are unconstrained thus allowing DV to validate the property by exploring or considering all possible parameter values.

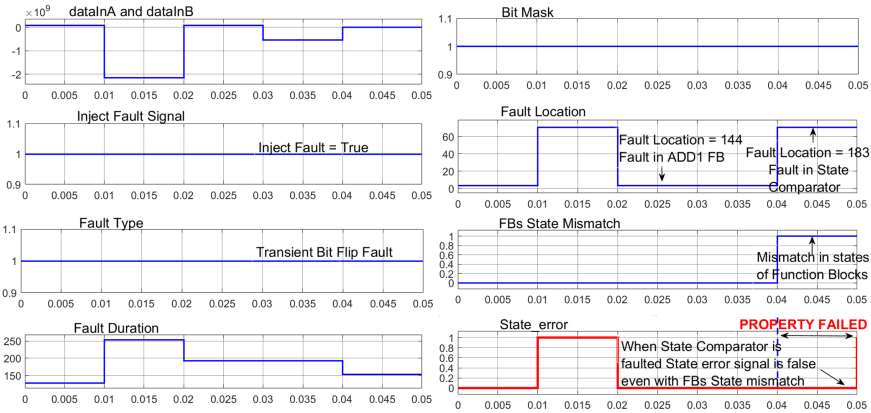

Fig. 8. Counterexample for failed property

The summary of the FI campaign on 6 different function blocks (FBs) are provided in Table 1. The results indicate that faults injected on any of the FBs in the duplex configuration satisfied the safety property as it always resulted in state mismatch error being detected by the system by pulling the 'state_error' signal high. But any transient/permanent faults within the State Comparators in the FBs caused the safety property to be violated with a counterexample as shown in Fig. 8. The counterexample shows a scenario where two separate faults are injected in the function block one after the other. First, a transient bit flip fault is injected in one of the function blocks (ADD1) (fault location ID = 144) leading to output mismatch between the two redundant FBs ADD1 and ADD2 in the next cycle, followed by another transient bit flip fault injected in the State Comparator block. This second fault in State comparator (fault location ID = 183) causes 'state_error' signal to remain false even when there is a mismatch between the redundant FB state/outputs. Thus, the State Comparators were identified as the single point of failures within the FB design. These results are reasonable with hindsight, as state comparators are a vital circuit in the SymPLe function block architecture. The state comparator fault tolerance capabilities were overlooked in the design. The data in

Table 1 shows an increase in the property validation time with an increase in the number of fault locations considered for each experiment.

Table 1. Property-based fault injection campaign results: SymPLe FB redundancy feature

Fault location IDs	No. of locations	Faults injected in module	Function block	Time taken (mins)	Proof validity
485–520	36	AND1	AND	15.3	Valid
449–484	36	AND2	AND	16.00	Valid
489–520, 521–528	40	AND1 & State Comparator	AND	17.3	Falsified
405–440	36	OR1	OR	14.09	Valid
369–404	36	OR2	OR	14.37	Valid
409–440, 441–448	40	OR1 & State Comparator	OR	16.06	Falsified
220–252	33	MAX1	MAX	11.17	Valid
187–219	33	MAX2	MAX	11.51	Valid
221–252, 253–260	40	MAX1 & State Comparator	MAX	17.25	Falsified
36–70	35	MIN1	MIN	16.01	Valid
1–35	35	MIN2	MIN	15.12	Valid
37–70, 71–78	42	MIN1 & State Comparator	MIN	17.53	Falsified
129–178	50	ADD1	ADD	26.16	Valid
79–128	50	ADD2	ADD	26.24	Valid
144–178, 179–186	43	ADD1 & State Comparator	ADD	15.43	Falsified
311–360	50	SUB1	SUB	26.36	Valid
261–310	50	SUB2	SUB	28.51	Valid
326–360, 361–368	43	SUB1 & State Comparator	SUB	19.3	Falsified

6.2 Safety Property 2 – Verifying Execution Timeout of Function Blocks

Another critical safety feature within SymPLe architecture is its detection of malfunction of the system by detecting a prolonged function block execution. The function block execution is expected to timeout, send an error and restart the task, when the execution of a function block extends beyond 50 cycles. The critical property that is expected to hold true is that: "*When 'execute' signal is detected high for 50 cycles, the 'error' signal and*

timeout bit in the error code shall be set to true indicating the timeout error detection."

$$\textit{Prop: ('execute' = true for 50 cycles)} \rightarrow \textit{('error' = true) and ('Timeout' bit in error code = true).} \quad (5)$$

This property is modeled as shown in Fig. 9. To verify this property, a fault has to be injected that can cause function block execution to timeout and cause an error. This error condition is modeled as an assumption as given in Fig. 9. It states that: *"Once 'execute' signal has gone high and execution has started, for the next 50 cycles execution is not completed and 'executed' signal does not become high".*

$$\textit{Assumption: ('execute' = true)} \rightarrow \textit{('executed'} \neq \textit{true for 50 cycles).} \quad (6)$$

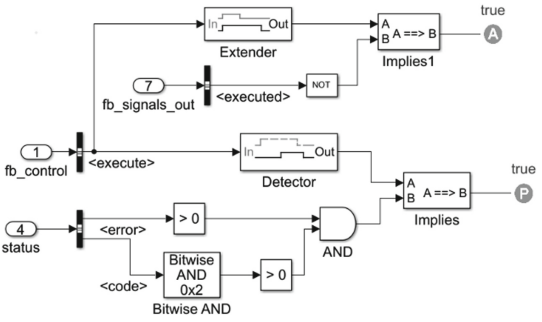

Fig. 9. Property for execution timeout functionality verification

This assumption causes Design Verifier to explore the design and identify all possible faults and input conditions to activate these faults. One direct example of a fault that can cause timeout error is the 'stuck at 0' fault (in bit 0) on the boolean 'executed' signal output from the function blocks. Similarly, there could be several faults that can create a delay in execution. This example also depicts the complete automation of fault and input space exploration by Design Verifier with minimal specification from the tester on the error scenario and fault tolerance property. The time-out property was validated true, indicating that a prolonged execution of function block extending beyond 50 cycles was detected and signaled as an error. The property proving experiment, for a single fault type (stuck-at-0) and all fault locations in a single function block took ~50 min to complete the time-out property verification.

7 Discussions and Future Work

As device geometries and clock speeds shrink, physical faults affecting the operation of devices are becoming a bottleneck to the dependability of safety-critical cyber physical systems. This paper introduces and develops a novel fault injection framework that utilizes the benefits of model checking and model-based design to achieve an efficient exploration of representative fault space, input and state space to attain near exhaustive

fault injection on functional models of safety-critical systems. The proposed method overcomes some of the deficiencies of traditional fault injection methods such as coverage of the fault injection experiment space and identifying fault activation conditions for each fault in the fault space. The property-based fault injection framework we developed provides an algorithmic/automated means to identify the fault activation conditions for all faults in the representative fault space using model checking. The time savings achieved with Property-based Fault Injection as compared to traditional simulation based fault injection is orders of magnitude more efficient. Our experiments with Simulink Design verifier and results (in Table 1) indicate that Property based fault injection method was very efficient with respect to exhaustive single fault location injection for our case studies (e.g. ~30 s/fault).

The limitations of Property based fault injection is that it works at the functional level of system development, that is, pre-software and hardware development. In the age of automatic code generation from Simulink models, the connection from models to executable code is robust and mature. As example, fault handling properties of error detection mechanisms can be discharged by property-based fault injection at the model level and then be synthesized into C code or VHDL code with greater design assurance. Another limitation is state space explosion for the Design Verifier model checker. All model checkers suffer from state explosion; we found that property based fault injection conducted at the sub-module level is more effective at controlling model checker state explosion issues. This leads one to construct fault injection campaigns based on "composability" arguments from system design – that is, verifying relations between high level system requirements and lower level design properties. In the future, we intend to investigate these composability aspects of property based fault injection derived from the traceability of system requirements. We also plan to extend the proposed property based fault injection framework with respect to hardware-level fault injection to investigate equivalence. Finally, we will examine the utility of the property based fault injection to assist in system level hazards analysis and failure modes and effects analysis on functional models of safety-critical applications.

References

1. Leveson, N.G., Fleming, C.H., Spencer, M., Thomas, J., Wilkinson, C.: Safety assessment of complex, software-intensive systems. SAE Int. J. Aerosp. **5**(2012-01–2134), 233–244 (2012)
2. Elks, C.R., et al.: Application of a fault injection based dependability assessment process to a commercial safety critical nuclear reactor protection system. In: 2010 IEEE/IFIP International Conference on Dependable Systems Networks (DSN), pp. 425–430, June 2010
3. Yu, Y., Bastien, B., Johnson, B.W.: A state of research review on fault injection techniques and a case study. In: Proceedings of Annual Reliability and Maintainability Symposium, Alexandria, VA, USA, pp. 386–392. IEEE (2005)
4. Elks, C.R., George, N.J., Reynolds, M.A., Miklo, M., Berger, C.: Development of a fault injection-based dependability assessment methodology for digital and I&C systems. United States Nuclear Regulatory Commission, Office of Nuclear Regulatory Research, NUREG/CR-7151 (2012)
5. Bingham, S., Lach, J.: Enhanced fault coverage analysis using ABVFI. In: Workshop on Dependable and Secure Nanocomputing, Charlottesville, p. 6, June 2009

6. Krautz, U., Pflanz, M., Jacobi, C., Tast, H.W., Weber, K., Vierhaus, H.T.: Evaluating coverage of error detection logic for soft errors using formal methods. In: Proceedings of the Design Automation Test in Europe Conference **1**, 1–6 (2006)
7. Leveugle, R.: A new approach for early dependability evaluation based on formal property checking and controlled mutations. In: 11th IEEE International On-Line Testing Symposium, pp. 260–265, July 2005
8. Larsson, D., Hähnle, R.: Symbolic fault injection. In: International Verification Workshop (VERIFY), vol. 259 (2007)
9. Vedder, B., Arts, T., Vinter, J., Jonsson, M.: Combining fault-injection with property-based testing. In: Proceedings of International Workshop on Engineering Simulations for Cyber-Physical Systems, Dresden, Germany, pp. 1–8, November 2013
10. Grunske, L., Winter, K., Yatapanage, N., Zafar, S., Lindsay, P.A.: Experience with fault injection experiments for FMEA. Softw. Pract Exp. **41**(11), 1233–1258 (2011)
11. Heimdahl, M.P.E., Choi, Y., Whalen, M.: Deviation analysis through model checking. In: Proceedings 17th IEEE International Conference on Automated Software Engineering, Edinburgh, UK, pp. 37–46 (2002)
12. Fondazione Bruno Kessler and Embedded Systems Unit. xsap - - The xSAP safety analysis platform. https://xsap.fbk.eu/. Accessed 13 July 2020
13. Joshi, A., Heimdahl, M.P.E.: Model-based safety analysis of Simulink models using SCADE design verifier. In: Winther, R., Gran, B.A., Dahll, G. (eds.) SAFECOMP 2005. LNCS, vol. 3688, pp. 122–135. Springer, Heidelberg (2005). https://doi.org/10.1007/11563228_10
14. Güdemann, M., Ortmeier, F., Reif, W.: Using Deductive Cause-Consequence Analysis (DCCA) with SCADE. In: Saglietti, F., Oster, N. (eds.) SAFECOMP 2007. LNCS, vol. 4680, pp. 465–478. Springer, Heidelberg (2007). https://doi.org/10.1007/978-3-540-75101-4_44
15. Jayakumar, A.V.: Systematic model-based design assurance and property-based fault injection for safety critical digital systems. Virginia Commonwealth University (2020)
16. Mathworks: Simulink® Design Verifier Reference, September 2018. https://www.mathworks.com/help/releases/R2018b/pdf_doc/sldv/sldv_ref.pdf
17. Shafik, R.A., Rosinger, P., Al-Hashimi, B.M.: SystemC-based minimum intrusive fault injection technique with improved fault representation. In: 2008 14th IEEE International On-Line Testing Symposium, Rhodes, Greece, pp. 99–104, July 2008
18. Elks, C., Gibson, M., Hite, R., Gautham, S., Jayakumar, A.V., Deloglos, C., Tantawy, A.: Achieving verifiable and high integrity instrumentation and control systems through complexity awareness and constrained design. USDOE Office of Nuclear Energy (NE), 15–8044, July 2019