Software Fault Propagation And Failure Analysis For UML Based Software Design

DISSERTATION

Presented in Partial Fulfillment of the Requirements for the Degree Doctor of Philosophy in the Graduate School of The Ohio State University

By
Chetan Vasant Mutha
Graduate Program in Mechanical Engineering

The Ohio State University
2014

Dissertation Committee:
Dr. Carol Smidts, Advisor
Dr. Tunc Aldemir
Dr. Raymond Cao
Dr. Manoj Srinivasan
Abstract

The research objective is to develop an early design stage fault propagation analysis method for software systems. The software designs considered are assumed to be developed in the standard Unified Modeling Language, which includes fourteen different diagrams. The proposed methods are called Failure Propagation and Simulation Approach (FPSA), and Integrated System Failure Analysis (ISFA).

FPSA allows fault propagation and failure impact analysis for UML-based design. It integrates several UML diagrams such that we can analyze the fault propagation paths within a particular diagram and also across different diagrams. The fault propagation paths are obtained through a simulation process, which consists in a complex execution of the design.

The ISFA method is developed for failure analysis of software-driven hardware systems. It semantically integrates the hardware design diagrams and the software UML diagrams. The impact of software faults on hardware functionality and vice-versa can be analyzed using this technique. ISFA is also simulation-based and follows the algorithm defined in this research.

These early design analysis methods will enhance the system designers understanding of the interplay between the multi-faceted elements of a design and allow through evaluation of their design options. The technique allows simultaneous
propagation of different type of faults from various domains and evaluates their functional impact. Based on the output, unique failure patterns can be built, and better fault tolerance strategies developed early in the design phase. Thus more reliable systems can be developed.

The mathematical formulation of FPSA and ISFA methods is extremely challenging. To calculate the fault propagation probability and system reliability thereof, the concept of flat parts is adopted. I propose an interval-arithmetic based rules formulation to determine the flat parts. Seven unique rules are defined for the basic arithmetic operations, three for advanced operators. These rules can be integrated into a tool along with a control-flow based algorithm, which can determine the faulty variables and the sequence of functions. Flat part based fault propagation analysis can give quick and accurate results early on. Function based propagation is not a widely studied topic of research; however it could be an indispensible approach to design of software intensive systems.
Dedication

This document is dedicated to my Advisor, my Family and Friends. My dad Mr. Vasant Mutha, mom Chetana Mutha, brother Dinesh, and sister Gauri. And to all my friends who supported me in this endeavor.
Acknowledgments

First of all, I would like to thank my advisor, Dr. Carol Smidts, for guiding, supporting and believing in me throughout my PhD. It has been an honor and fun to work with her, especially the intense discussions and arguments during the proposal and paper writing. In my experience, “I run on coffee” slogan best describes her. Under her guidance, I grew as researcher, as a critical writer, and as a person in general.

I would also like to thank Dr. Tunc Aldemir, Dr. Richard Denning, Dr. Giorgio Rizzoni, and Dr. Manoj Srinivasan for serving on my candidacy and dissertation committee. Their inputs, technical insights, and questions were valuable in advancing my research.

Thanks to all my friends, who kept me motivated, helped maintain my sanity and made this journey a really enjoyable one with all the fun parties and adventures. A special thanks to Shrikant, Ravi, Amita, Chaitanya, Vinayak, Akon, Shweta, Sahoo, Sushma, Praneet, Arpit, Roshni for lending your shoulder and feeding me when I was sick and bedridden.

Also thanks to the sponsors AFOSR, DOE, and NRC for supporting the research on my topic area.

Above all, this endeavor was possible because of my family support and encouragement.
Vita

2002 ...............................................................Loyola High school, Pune
2006 ...............................................................B.E., Mechanical, University of Pune
2006 to 2008 .................................................Software Engineer, Infosys Technology
2009 to present ..............................................Graduate Research Assistant, Department of
                Mechanical and Aerospace Engineering,
                The Ohio State University

Publications

Functional Failure and Propagation Analysis Approach for Safe System

Operational Profile- OP definition. ACM computing survey journal.

Fields of Study

Major Field: Mechanical Engineering
# Table of Contents

Abstract ............................................................................................................................... ii  
Dedication .......................................................................................................................... iv  
Acknowledgments ............................................................................................................... v  
Vita ..................................................................................................................................... vi  
List of Tables ..................................................................................................................... xi  
List of Figures .................................................................................................................. xiii  
Chapter 1: Introduction ...................................................................................................... 1  
  Contributions ................................................................................................................... 4  
Chapter 2: Literature Review ............................................................................................. 8  
Chapter 3 Failure Propagation and Simulation Approach ................................................ 16  
  Formalization of FPSA .................................................................................................. 18  
  Behavioral Simulation ................................................................................................. 25  
  Elements of execution algorithm ............................................................................... 26  
  Action model ............................................................................................................. 27  
  Interaction model ...................................................................................................... 29
Trace Model ............................................................................................................. 32
Relationship between Action and Interaction Model ........................................... 33
Timing elements of simulation ............................................................................. 34
Chapter 4. FPSA Execution Model ....................................................................... 36
Case Study ............................................................................................................... 53
  Case 1.1- A fault missing Tank class ................................................................. 59
  Case 1.2 Missing message "getdata" ................................................................. 82
  Case 1.3 Missing attribute “Level” ................................................................. 82
  Case 2.1: Design 1 with missing activity “Read Position” ................................. 89
  Case 2.2: Design 2 with missing activity “Read Position” ................................. 89
  Case 3: Two simultaneously occurring faults Missing activity and Extra message ................................................................. 92
Chapter 5. Integrated System Failure Analysis ...................................................... 94
  Functional Failure Identification and Propagation .............................................. 95
  Formalization of FFIP ......................................................................................... 95
  Behavioral Simulation ......................................................................................... 100
  Formalization of ISFA ....................................................................................... 101
<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Interface</td>
<td>102</td>
</tr>
<tr>
<td>Transaction</td>
<td>105</td>
</tr>
<tr>
<td>TimingConstraint</td>
<td>107</td>
</tr>
<tr>
<td>ISFA Execution Model</td>
<td>112</td>
</tr>
<tr>
<td>HW design execution</td>
<td>114</td>
</tr>
<tr>
<td>SW design execution</td>
<td>117</td>
</tr>
<tr>
<td>Evaluation of System Function Status</td>
<td>119</td>
</tr>
<tr>
<td>Case Study</td>
<td>120</td>
</tr>
<tr>
<td>System Models</td>
<td>121</td>
</tr>
<tr>
<td>Chapter 6. Mathematical Formulation</td>
<td>139</td>
</tr>
<tr>
<td>Basic Arithmetic Operations And Flat Part Determination</td>
<td>143</td>
</tr>
<tr>
<td>A. Addition: ( f(x) + g(x) )</td>
<td>146</td>
</tr>
<tr>
<td>B. Subtraction: ( f(x) - g(x) )</td>
<td>147</td>
</tr>
<tr>
<td>C. Multiplication: ( f(x) \cdot g(x) )</td>
<td>148</td>
</tr>
<tr>
<td>D. Division: ( f(x)/g(x) )</td>
<td>149</td>
</tr>
<tr>
<td>Advanced Arithmetic Operations And Flat Part Determination</td>
<td>150</td>
</tr>
<tr>
<td>A. Integration: ( \int f(x) , dx )</td>
<td>150</td>
</tr>
</tbody>
</table>
B. Derivative: \( df(x)dx \) ................................................................. 151

C. Composition: \( g(x) \circ f(x) \) .......................................................... 152

Flat Parts Of Function Composition ...................................................... 153

A. Constant ............................................................. 154

B. Linear ............................................................. 155

Missing Function Fault ................................................................. 162

Propagation Probability Calculation ............................................. 164

Chapter 7. Conclusions and Discussion ........................................ 167

References ......................................................................................... 173

Appendix- Table and Proofs .......................................................... 177
List of Tables

Table 1 FPSA-Specific Relationships................................................................. 20
Table 2 Variable And Design Limitations Associated With Software Component “Valve Controller” ........................................................................................................................ 24
Table 3 Timing Elements Used In This Research ................................................. 35
Table 4 High-Level FPSA Simulation Results For The Three Fault Cases .......... 83
Table 5 Faults and sample patterns..................................................................... 88
Table 6 Result Of Missing Activity “Read Position” Fault In Two Different Designs.... 89
Table 7 Comparison between Design 1 and Design 2 ........................................... 92
Table 8: Sample behavioral rules and functional failure logic of an interface ............ 105
Table 9: All possible combinations of [Start, Finish] and corresponding interpretation of Status......................................................................................................................... 108
Table 10: Mapping of hardware component to hardware function.......................... 124
Table 11: Mapping of transaction to provided and required interfaces...................... 125
Table 12: Variable and design limitations associated with software component “valve controller” ............................................................................................................................ 125
Table 13: Simulation Results. O = Operating, L = Lost, IA = Inactive, C = Complete . 138
Table 14 FFIP Modeling Elements............................................................................. 177
Table 15 Behavioral rules and Function Failure Logic (FFL)................................. 178
Table 16 Behavioral rules and Function Failure Logic (FFL). O=Operational; L=Lost; U=Unknown

Table 17 Activity and corresponding trace specification

Table 18 Event Sequence Diagram syntax and semantics
List of Figures

Figure 1 Software Design Levels, Dimensions And Perspectives Of Different Experts.... 3
Figure 2 FPSA Execution Levels................................................................................... 5
Figure 3 The Failure Propagation And Simulation Approach Mapping Metamodel ...... 19
Figure 4 (a) The software component “Valve Controller” and its input–output variables
(P2 and ControlCommand), (b) the activity “valve control logic,” (c) a sample behavioral
rule in terms of the relationship between input–output variables, and (d) a sample
function failure logic...................................................................................................... 24
Figure 5 Meta-Model Of The Action Model Used In This Research.......................... 27
Figure 6 Part Of An Interaction With Concepts Of Interaction Model ...................... 29
Figure 7 Meta-Model Of Interaction Model Used In This Research........................... 30
Figure 8 The Relationship Between The Action And Interaction Models............... 34
Figure 9 The Relationship Between Action And Event.............................................. 34
Figure 10 A Conceptual Representation Of The Action Model ESD......................... 36
Figure 11 A Portion Of The “Order Creation” Interaction........................................ 37
Figure 12 Conceptual Representation Of The Interaction Model ESD....................... 38
Figure 13 Action Model ESD.................................................................................... 48
Figure 14 Interaction Model ESD.............................................................................. 49
Figure 15 Hold-Up Tank System Example (extracted from [Mutha et al. 2012])......... 53
Figure 32 Event pools of Pressure sensor, Position sensor, Tank, & Controller - Case 1.1
........................................................................................................................................... 84

Figure 33 Event pools of Pressure sensor, Position sensor, Tank, & Controller - Case 1.2
........................................................................................................................................... 84

Figure 34 Event pool of Pressure sensor, Position sensor, Tank, Controller and Inlet valve
- Case 1.3 (Simulation step 1) .......................................................................................... 84

Figure 35 Functional Failure Identification And Propagation Metamodel.............................. 96

Figure 36: (a) The valve component, its input–output variables, and the flow; (b) the
valve function and flow; (c) valve behavioral rules in terms of input–output relationship;
and (d) valve function failure logic...................................................................................... 100

Figure 37: (a) Integration of the functional failure identification and propagation (FFIP)
and failure propagation and simulation approach (FPSA) metamodels ......................... 104

Figure 38: The Algorithm_TStatus .................................................................................. 111

Figure 39: A simulation step and simulation run ................................................................ 112

Figure 40: The integrated system failure analysis execution model at time step t. ................. 114

Figure 41: An evaluation of the system function status ......................................................... 120

Figure 42: ISFA functional model of the holdup tank system .............................................. 122

Figure 43: ISFA component diagram .................................................................................. 122

Figure 44: Sample transaction instance <<transaction>>T1 .................................................. 123

Figure 45 Illustration of flat parts ...................................................................................... 141
Figure 46 Canonical form of an execution path.............................................................. 142
Figure 47: Addition......................................................................................................... 147
Figure 48: Subtraction................................................................................................. 148
Figure 49: Multiplication.............................................................................................. 149
Figure 50: Division......................................................................................................... 150
Figure 51: Integration .................................................................................................. 151
Figure 52: Differentiation ............................................................................................ 152
Figure 53: Functions f(x) and g(x)............................................................................... 159
Figure 54 Composition g o f(x) .................................................................................. 159
Figure 55 Composition g o f(10x) .............................................................................. 159
Figure 56 Composition g o f(10x) .............................................................................. 159
Figure 57 Example function flow of software system with missing function fault MF. 160
Figure 58 Statemachine diagram of a variable .............................................................. 161
Figure 59: Missing function fault demonstration.......................................................... 163
Figure 60 Composition of $f_3 \circ MF \circ f_1$ ................................................................. 164
Figure 61: NPP calculation.......................................................................................... 165
Chapter 1: Introduction

Software reliability is the probability that software performs the required function without any failures for a specified period of time within a specified operating environment [IEEE1633, 2008]. In case of safety-critical systems such as aircrafts, automobiles, nuclear power plants and medical devices, software failures can cause heavy financial losses, loss of human life, and major mission failures.

Software failure is said to occur when faults within the software or faults from software operating environment are triggered through execution and create perturbations that propagate to cause an undesired output, typically disastrous. Understanding software faults, their origin, their propagation mechanism, and associated failure impact is a highly complex problem; and essential in building reliable systems. Software faults are reported to cost $59.5 billion per year [Tassey, 2002]. Only one-third of this cost can be reduced by improved testing [Tassey, 2002]. Additionally, the cost of fixing bugs in the testing phase is between 10 to 100 times more expensive than fixing the same bugs in the design phase. Clearly, to reduce the remaining two-third of the cost, failure analysis of requirements and design faults are needed.

My research focuses on the design-level safety and reliability analysis of software-intensive systems. Particularly, the objectives of the research are twofold. First, establish a fault propagation and failure analysis framework for software-intensive system design.
Second, formulate a mathematical basis to enable the quantification of fault propagation and reliability. Since, designs are abstract and multi-dimensional in nature the problem becomes highly complex and has several challenges.

First, software is highly interactive in nature which renders prediction of the exact fault execution paths difficult. A fault of one type may propagate differently from another type, or different types of faults occurring simultaneously may propagate in a similar manner [Hiller et al., 2004]. Thus the problem of fault propagation quickly escalates. In addition, faults originate from varied sources. Faults are introduced during each phase of development; faults introduced in earlier phases of development are inherited in the later phases. For instance, faults from the requirement phase enter the software design; new faults are added in the design phase, and all these faults further enter the coding phase where finally coding faults are added to the mix. Fault distribution analysis in [Leszak, 2000] reported that 43% of faults were introduced in the early design phases; of these, 48% were algorithm faults and 44% were functionality faults. At a high level of abstraction, software faults can be broadly classified into categories such as omission and commission, however there is no unique classification at a lower level of abstraction. A common consensus is seen on some high-level faults such as logic and data handling faults [Beizer, 1990]. Thus, to develop reliable software it is essential to analyze the design faults, their impact, & how to capture and fix the faults. This is not a trivial task. Second, the software design representation is multi-level and multi-dimensional (see figure 1). Information related to the requirements and implementations coexist in an abstract form. For example, use case, and activity diagrams are used in the design to
represent requirements. These diagrams capture the functional dimension. The component diagram allows representation of the structural dimension. These diagrams are a part of the “high level” design representation and are particularly developed by domain experts. Diagrams such as sequence and class diagrams are used to represent information about the implementation. The sequence diagram captures the communication information in a time dimension (i.e. the communication carried out by a software object with other entities in its lifetime). The class diagram allows representation of the structural dimension (at the implementation level). These diagrams constitute the part of the design representation which will be referred as “low level” representation in this research. The implementation level design representation is for the benefit of software developers or coders.

Figure 1 Software Design Levels, Dimensions And Perspectives Of Different Experts
The software architect or designer orchestrates the entire software development; hence he must understand, and analyze the impact of a fault through all the levels of design representation.

Third, the development of a single mathematical basis for such a complex, multi-dimensional problem is a daunting task. Each level, each dimension, and each aspect of the dimension can potentially follow different mathematical formulations. For instance, to formulate the communication aspects of the structural dimension (particularly the sequence diagram representation), petri-nets or state-machine representations could be used. To formulate the control-flow and data-flow aspects, represented in the activity diagram, liveness and reachability based algorithms can be used. These two formulations follow notations that cannot be integrated into one. Developing a hybrid mathematical theory for the complex fault propagation analysis will take years of research.

In this research, the design representation adopted is the Unified Modeling Language (UML), which is a standard modeling language used to design the structure and the behavior of object-oriented software. UML provides seven different diagrams for structural design and seven different diagrams for behavioral design. Each diagram conveys specific design information from different points of view (eg. user, architect, or developer as shown in figure 1).

**Contributions**

To tackle the challenges discussed above, I propose the Failure Propagation & Simulation Approach (FPSA) that integrates the design diagrams such that fault paths can be traced early on, before the software implementation. A design-mapping meta-model is
defined [Mutha et al. 2012] to address the multi-level and multi-dimension design issue. The meta-model also enables propagation analysis of different types of faults originating in different diagrams. The propagation of the most commonly occurring fault types—“missing”, “incorrect” or “extra” design element is demonstrated.

To achieve fault propagation analysis through a multi-level and multi-dimension design; two execution algorithms are defined: the high level and low level execution (see figure 2) algorithms. These design execution algorithms collect information to enable identification of unique failure patterns. Sample failure patterns are demonstrated with the help of a case-study. Within FPSA, the functional failure impact of the fault is determined using the behavioral rules and functional failure logic as proposed in [Tumer et al., 2011].

![Figure 2 FPSA Execution Levels](image)

One of the outputs of the proposed simulation-based approach is fault propagation paths. The fault propagation paths are derived from the activity (at high-level) and sequence diagrams (at low-level). The feedback from low-level execution may indicate that the actual path (at high-level) is different from the high-level path initially assumed.
Such fault propagation path revelations are important for an accurate assessment of potential risks hiding in the design or arising from the design changes.

Software is a part of a system. It drives the electro-mechanical components to achieve a system function. Hence, I further extended the FPSA method to analyze systems composed of two sub-systems—physical hardware (HW) and a software (SW) that interfaces with the hardware. The physical subsystem represents the electromechanical components; the software subsystem handles the control and decision logic for achieving the functionality of the physical system. The new method is called Integrates System Failure Analysis (ISFA). The ISFA method is composed of two methods: FPSA and Functional Failure Identification & Propagation (FFIP). The FFIP method was developed for fault propagation and effects analysis of electromechanical systems [Kurtoglu & Tumer, 2008; Tumer & Smidts, 2011].

As discussed earlier, a standard fault taxonomy for software faults is not available. I propose a list of design faults to be studied based on the existing literature on software fault classification and software fault taxonomy. Particularly, the list of faults is developed based on most commonly occurring fault types such as “missing”, “incorrect” or “extra” design element.

The proposed approaches have several advantages:

- The FPSA and ISFA techniques will enable the designer to (a) proactively analyze complex software functionalities and their failure modes; (b) visualize complex functional interactions and the resulting failure modes; (c) identify failure-propagation paths within a particular software subsystem; (d) identify which
function(s) will be lost, their impact on the overall system, and safeguards/redundancies that should be added; and (e) provide a safety analyst with sufficiently detailed results so that s/he can understand the safety risk(s).

- The fault propagation path will provide insights into the interplay between different entities of the design. These insights could be effective in determining the functional impact of the faults. Faults may propagate from a non-critical function to a critical function causing a catastrophic failure.

- *Design modification support*- A design modification may be necessary to increase the reliability of a software design and consequently of the software product. Design modification decisions can be supported by the analysis of the faults, their propagation paths, and their functional impacts on the new design.

- *Design Trade-offs*: A design modification may not be viable due to increased resources and expenditures necessary for its implementation. As such trade-off analyses may be required, to select the most appropriate design. FPSA based analysis will provide a basis for such design trade-offs.
Chapter 2: Literature Review

Safety and reliability assessment is one of the primary activities performed during each stage of software development life-cycle of a safety-critical system. At the early stages of development, the reliability assessment is performed according to the set standards [MIL-STD-1629A, 1984; FAA, 2000]. The standards used in aerospace, nuclear, or medical fields mandate the use of several techniques Failure Modes and Effects Analysis (FMEA), Fault Tree Analysis (FTA), Probabilistic Risk Assessment (PRA) which includes the Event Tree(ET), and Event Sequence Diagram (ESD) techniques [NUREG/CR-2300], Petri-nets, and State-based techniques (e.g. Markov chain models). The analysis results may necessitate total change of the system design.

FMEA is an inductive technique for systematic risk analysis of the system. During the analysis a team of experts enumerates failure modes, their causes and their effect for each component in the system. Further a quantitative risk assessment is provided based on the rating scale (1-10) for severity, likelihood, and detectability. Although this is a valuable risk assessment technique at early design stage, it is not the most effective one for complex systems. There are inherent limitations to this approach. First, the experts manually identify the effect of fault propagation. Second, only single faults can be analyzed at a time. Third, FMEA does not explicitly capture component interactions. Finally SW FMEA has limited applicability since the software faults, their evolution, and
their impact are more complex and difficult to understand without execution of actual software code. In my research, I propose to address these issues by integrating the HW and SW design in one unified and formalized model, followed by a qualitative simulation to identify fault propagation paths and their functional consequences. The proposed techniques can be a complement to FMEA analysis for software-intensive systems.

While FMEA attempts to identify system-level consequences from a component-level fault, FTA decomposes a critical system-level failure into logical combinations of component-level failures. FTA is a deductive technique that identifies the possible root causes of an undesirable system state. FTA provides fault propagation paths between the basic events (root causes of component failure) and the top event (system failure) in the form of Boolean logic. However FTA possesses some fundamental limitations. First, FTA is a snapshot of a system state, there is no concept of dynamic evolution of system state. Secondly, the Boolean logic is informally constructed during the expert identification of event–consequence relationships, or using other less informal approaches such as digraphs [Lapp et.al., 1977], decision tables [Lee et. al. , 1985], or qualitative simulations [Lee et. al. , 1985]. The development of the Boolean logic becomes increasingly difficult as the system gets complex. Third, in case of software systems, the FTA analysis is mostly done at the code-level [Towhidnejad et al., 2003].

Originally these techniques were built to study hardware systems. However as the systems evolved into complex software intensive systems, these techniques were also adopted for development of safe/reliable software. But, software safety and reliability
cannot be thoroughly addressed by these techniques for several reasons. Software systems are different from hardware systems in following ways:

1) Software systems are primarily composed of logical components;

2) There is no wear and tear of the software due to usage;

3) Software engineering is still in infancy and is constantly evolving. For instance, a transition between design paradigms, i.e. the transition between sequential programming and object-oriented design was introduced just two decades ago;

4) Software interacts with other hardware and software systems, which have their own behavior. Thus a complex interaction pattern evolves.

FMEA and FTA are static in nature and capture limited faults and failures. To capture additional faults and their impact, dynamic reliability assessment methods should be employed early in the design process. Dynamic methods model the software behavior using representations as state-machine, sequence diagram, and activity diagrams. However, only few dynamic methods have been thoroughly developed to analyze reliability and among these a few account for the fault propagation aspect. These dynamic techniques are popularly are typically component-based or architecture-based. The following literature review focus on the design-stage component-based reliability analysis techniques.

[Gokhale et al., 1997] proposes a “composite method” and a “hierarchical method” to determine software reliability. These methods are based on structural models composed of software architecture and failure behavior. The software architecture captures the interaction dynamics between different components and is modeled using Markov chains.
The failure behavior of the component and its interfaces are specified as probabilities of failure (or reliability) or failure rates. Fourteen different analytical expressions to calculate reliability corresponding to the discrete-time-markov-chain (DTMC) and the continuous-time-markov-chain (CTMC) respectively are also described. Further in [Gokhale et al., 2005] proposes a simulation-based reliability estimation approach, since analytical models become mathematically intractable when software reliability growth and fault tolerance are considered. A simulation algorithm to assess the impact of fault detection and repair of the component on the application reliability during the testing phase is described in the paper. However, a one-to-one mapping between the fault and failure (failure of each component leads to failure of the application) is assumed. This assumption highly underestimates the software reliability.

[Gokhale, 2007] describes the limitations of state-of-the-art models. Some of the modeling limitations are: i) Concurrent execution of components is not considered; ii) Non-markov transfer of control between components is not considered; iii) Failure independence (failure of one component does not propagate to other components) is considered. However, dependency arises due to data exchange, or message passing between the components; iv) Component failure models are homogenous. A capability to incorporate failure models of varied nature can significantly improve the reliability estimation. For example, for in-house software time-dependent failure rates may be available, while for commercial off-the-shelf (COTS) softwares only component reliabilities may be available; v) Varied architecture styles such as event-based, or database style are observed in developed software. It is unclear how the reliability of
applications with heterogeneous style architectures should be calculated. In my research, the integration of different UML models provides a basis for heterogeneous style reliability modeling. Secondly, I work with the available UML design representation, so there is no need to build special Markov chain architectural models. Interesting UML based reliability analysis research is carried out by few researchers as described next.

[Singh et al., 2001] proposes a UML-based approach to calculate the software reliability. The use case diagram (UCD) is annotated to calculate the access probability of a particular use case. The annotations include: i) probability of user ‘i’ being selected (q_i); ii) Probability of the user ‘i’ accessing a use case ‘j’ (P_ij). Further, for each use case a corresponding sequence diagram (SD) is assumed to exist. The components in SD interact with each other by exchanging messages. Each component is annotated with its probability of failure. The reliability models, built from these annotated UCD and SD, are based on two important assumptions: i) independence of failure among components; ii) regularity- a component failure probability is equal across all the messages of a component. These assumptions are acceptable for initial reliability assessments; however they do not reflect the true behavior of the software execution. Component dependencies are intricate and may not be explicit. In the proposed ISFA technique implicit component dependencies can be studied by observing the patterns of component failures and functional failures captured in the result tables.

[Cortellessa at al., 2007] proposes a recursive equation to calculate the software reliability, where recursion accounts for \( k \)-transactions from component ‘i’ to ‘j’. The equation is composed of two mutually exclusive probabilities: i) an internal component
fault $intf(i)$; ii) an error propagation of component $ep(i)$. Cortellessa claims that the lack of error propagation consideration underestimates the system reliability significantly. The claim is supported by the experimental results, which show that a 10% decrease in error propagation (i.e. from 1 to 0.9) leads to a 74% increase in the whole system reliability. Although Cortellessa makes a strong case that error propagation is an important aspect of reliability calculation, the model he proposes is based on assumptions which may not be true. These assumptions include: i) data error always propagates through the control flow; ii) Operational Profile (OP) of component $C_i$ is known and it follows a markov property. In the proposed ISFA technique, a more complex error propagation model is built that takes into account the execution of the hardware environment within which the software operates, thus the markovian assumption is not required.

[Popovic et al, 2005] implements error propagation models proposed by [Nasser et al., 2004] and [Cortellessa et al., 2007] in the Early Component-based Reliability Assessment (ECRA) tool. He presents an experimental analysis of the PACS system and a Monte-Carlo simulation of system failure equations. The experimental and analytical results indicate that error propagation makes a significant difference in system reliability prediction, especially if components’ erroneous states are complex and frequently used.

[Yacoub et al., 2004] proposes a scenario-based reliability analysis (SBRA) algorithm. The algorithm is based on a component dependency graph (CDG) derived from the UML sequence diagram. CDG consists of “nodes” annotated with the component name, reliability, and execution time information and the “arcs” are annotated with transaction name, reliability, and execution probability information. The reliability
and execution probabilities are based on the scenarios. Scenarios represent component interactions and are derived from the operational profile and equivalence partitioning. Limitations of the proposed approach include identification of independent scenarios, sequential execution of the components, and lack of fault propagation consideration. [Goseva et al., 2003] proposes an approach for architecture-level risk analysis, which uses FMEA and markov models to obtain the risks. The analysis is based on two UML diagrams specifically UCD and SD. However even her approach does not explicitly include fault propagation.

Evidently, fault propagation and resulting failure plays a vital role in reliability analysis of complex system [Cortellessa at al., 2007] [Popovic et al, 2005] [Nasser et al., 2004]; however there is a lack of fault propagation consideration in the proposed approaches and the corresponding mathematical formulation.

The mathematical formulation of reliability models are either: state-based and path-based. Compared to state-based approaches, the research on path-based reliability analysis approaches is limited. Some of the notable research is discussed next.

Hiller et al. [Hiller et al., 2002] mentions that propagation characteristics are dependent on the error type i.e. an error of one type may propagate differently than an error of another type. The author reports that the propagation approach is more effective than the heuristic approach for the placement of error detection mechanisms. The use of the propagation approach cuts down the resource requirements by half while maintaining detection coverage.
Hiller et.al [Hiller et al., 2001] proposes an error propagation analysis method, which does not use state-transition representation as seen in earlier research. The authors study propagation of data-errors. The propagation is characterized by an input-output based error permeability measure. The measure is a probability that an error is will be observed in the output, given an error occurred on the input. Once the measure is calculated for each input-output pair, a permeability graph is constructed, which is used for propagation analysis. A graph node represents a system module/component and the incoming and outgoing arcs represent the inputs and outputs respectively. Each arc is assigned an error permeability value. Two different propagation analyses i.e. output error tracing (backtracing), and input error tracing (forward tracing) can be performed. These measures are used to capture the propagation information, which is further employed in determining the locations for error detection, recovery mechanisms, and effect on system operation.

Hiller et al.‘s path-based approach is closely related to my research. In my research, the forward path tracing algorithm is proposed. I do not assume an “error permeability” probability but propose a closely related concept called flat parts to calculate the fault propagation probability analytically. This analytical approach is a white-box functional approach, as opposed to the black-box component approach. This functional path-based approach establishes the mathematical basis of fault propagation calculation. Further, I adopt the PIE theory proposed by [Voas, 1992] for reliability calculations.
Chapter 3 Failure Propagation and Simulation Approach

Early design-stage software safety and reliability analysis is grueling and traditional risk assessment techniques (such as FTA and FMEA) are ill-equipped to handle the complexity of software. To address the software related gaps in the risk analysis, a novel approach called the Failure Propagation and Simulation Approach (FPSA) is introduced in this chapter. A part of the FPSA was developed in my Master’s research. The research continued to grow into my PhD work. The initial FPSA approach was modified and further refined to include all the elements of activity diagram, and sequence diagram. It was also modified to enable the development of ISFA method (discussed in chapter 5).

FPSA is a UML-based software fault propagation and effects analysis method. The important concepts of FPSA are summarized below, followed by formalization of the FPSA method.

**Mapping**- The FPSA mapping metamodel depicts the inter-diagram relationships (figure 3). These relationships semantically map one diagram element to other diagram elements. The original UML metamodel of individual diagrams such as activity, state machine, and use case are preserved while newer, more specific relationships are established. These across-diagram relationships facilitate navigation from one diagram to another.

**Behavioral Rules**- Behavioral rules of a component are qualitative in nature. The rules model both nominal and faulty modes of the component. A nominal mode can transition
to another nominal or faulty mode and vice-versa. The transition is determined by the underlying first principles of the system, which is primarily represented as a relationship between the input and output variables the component. The component implements different functions (represented as activities). These functions transform the inputs to the outputs. An incorrectly executed activity will result in incorrect output values and consequently trigger a nominal or faulty mode of other components. The behavior models are represented in the *if-then-else* conditions format.

**Functional Failure Logic**- The functional failure logic is a reasoning tool that relates the component behavior to the operating state of system functions. An operating state is defined as operating (O), Lost (L), or Unknown (U). The FFL could be implemented as *StateMachine*, or *if-then-else*.

**Behavioral Simulation**- The UML-based design provides several options for behavioral simulation. The simulation can be driven by the state machine, the activity diagram, or the sequence diagram. Of these, the simulation driven by the activity diagram fulfils our need to simulate the overall system. There two levels of simulation: high-level and low-level.

The high level simulation employs the Activity diagram, Component diagram, Behavioral Rules, and FFL. The simulation determines the high-level function failures. The low-level execution employs an Activity diagram and Sequence diagram. It provides insights into the low-level cause of the function failure. The insights obtained can be used to develop patterns of function failure for different designs. Different designs may exhibit different patterns of failure. These patterns are qualitative in nature.
The high-level behavioral execution of the FPSA is a simple process involving the Activity diagram, components, their behavioral rules and functional failure logic. The Activity diagram presents different paths of execution. For execution purpose we select one of the paths of execution and each activity of the Activity diagram is traversed following the control flow edges. Each activity is mapped to a component via the activity partition. A component is associated with the behavioral rules and FFL. We execute the behavioral rules and FFL to determine whether the component’s function (activity) status is operating, lost or unknown. Results of each activity execution are recorded in a table format for further analysis and pattern evaluation.

The execution of an activity involves a set of messages that are passed between the objects and representation in sequence diagram. Since sequence diagrams represent the implementation level details and closer to actual execution of the software, we refer to this execution as low-level execution and it is explained next.

\[
\text{Lower-level execution} = \text{Activity} + \text{Sequence diagram} \rightarrow \text{Action ESD} \leftrightarrow \text{Interaction ESD} \rightarrow \text{Reasons of failure, patterns etc.}
\]

The low-level execution is a complex algorithm that involves execution of two different diagrams (Activity and Sequence) simultaneously. The algorithm is composed of the executable actions defined in the UML semantic architecture [UML, 2010].

**Formalization of FPSA**

Failure Propagation and Simulation Approach (FPSA) is formalized as shown in figure 3 and table 1. FPSA focuses on the conceptual design phase where the functions
and system structure is more important than the implementation-level details. The functional requirements and software architecture are depicted in form of use case, component, and deployment diagrams. Additional diagrams that capture details such as the control logic, behavior of the objects at runtime, and the software structure are necessary for safety and reliability analysis. These details are captured by the activity, sequence, state machine, and class diagram. Each of these diagrams has certain unique features and certain level of duplication of information. Hence, it is necessary to understand these diagrams and the relationship between their unique elements. Figure 3 depicts important the relationships between different diagram elements and are formally explained in table 1.

Figure 3 The Failure Propagation And Simulation Approach Mapping Metamodel
<table>
<thead>
<tr>
<th>Relationship</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Activity – Activity</td>
<td>This unidirectional association indicates that each Activity should be Partition surrounded by one or more Activity Partitions. In other words, the elements represented by the activity partition are responsible for the enclosed activity. This relationship modifies the Activity’s association “partition: ActivityPartition [0..<em>]” (specified in [OMG, 2010]) to “partition: ActivityPartition[1..</em>].”</td>
</tr>
<tr>
<td>Activity Partition –</td>
<td>This unidirectional association indicates that one or more Activity Partitions are represented by one Component that is a part of the Component diagram. This relationship is based on the Activity Partition’s association of “represents: Element [0..1]” (specified in [OMG, 2010]) to “represents: Component [1].” This association connects the Component diagram to the Activity diagram.</td>
</tr>
<tr>
<td>Component</td>
<td></td>
</tr>
<tr>
<td>Activity – Use Case</td>
<td>This simple association indicates that an Activity acts as the behavior of the Use Case. The multiplicity indicates that an Activity can be used in multiple Use Cases. This association is one way to specify the Use Case behavior. This association connects the Activity diagram and Use Case diagram.</td>
</tr>
<tr>
<td>Component – Class</td>
<td>This compositional relationship indicates that a Component is composed of one-to-many Classes. This relationship is derived from the Component’s association “packagedElement: PackageableElement [*]” specified in [OMG, 2010]. Here the packageableElement is Class. This association connects the Class diagram to Component diagram.</td>
</tr>
<tr>
<td>Component – Interface</td>
<td>This compositional relationship indicates that each Component is composed of multiple required or provided Interfaces. These interfaces act as a contract between the two Components that share the services. The relationship is mentioned here because it is used in ISFA while integrating the hardware and software domain (even though it is the same as the one defined in [OMG, 2010]).</td>
</tr>
<tr>
<td>Component – State Machine</td>
<td>This compositional relationship indicates that each Component is composed of one State-Machine diagram, which captures the functional failure logic (FFL). The relationship FFL is not a part of [OMG, 2010].</td>
</tr>
<tr>
<td>Component – Opaque Action</td>
<td>This compositional relationship indicates that each Component is composed of one Opaque Action, which captures the behavioralRule. The relationship behavioralRule is not a part of [OMG, 2010].</td>
</tr>
<tr>
<td>Component – Variables</td>
<td>This simple association indicates that each Component can have multiple input/output Variables. These variables will be marked on the connectors. Constraint: C2 Context: Component Inv: If component.interface = provided implies Component.variables = output Inv: If component.interface = required implies Component.variable = input</td>
</tr>
</tbody>
</table>

Table 1 FPSA-Specific Relationships

Continued
Table 1 continued

<table>
<thead>
<tr>
<th>Association</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Class – State Machine</td>
<td>This simple association indicates that behavior of the object of type Class is represented with a State Machine diagram.</td>
</tr>
<tr>
<td>Class – Use Case</td>
<td>This simple association indicates that each Class is a subject of multiple Use Cases. This relationship is derived from the Use Case association “subject : Classifier[*]” specified in [OMG, 2010] This association connects the Class diagram and Use Case diagram.</td>
</tr>
<tr>
<td>Use case – Interaction</td>
<td>This simple association indicates that each UseCase behavior can be described by one Interaction, a behavioredClassifier. This association is one way to specify the UseCase behavior. This association connects the Use case diagram to Sequence diagram.</td>
</tr>
<tr>
<td>Class – Message</td>
<td>The relationship indicates that the Message signature is assigned to one Class. The Message signature is represented as operation in the Class. This relationship is derived from the Message’s association “/signature:NamedElement[0..1]” specified in [OMG, 2010]. This association connects the Class diagram to the Sequence diagram. <strong>Constraint: C3</strong>&lt;br&gt;<strong>Context:</strong> Message&lt;br&gt;<strong>Inv:</strong> Message.signature = class.operation → not empty()</td>
</tr>
<tr>
<td>Lifeline – Class</td>
<td>This type association indicates that the object represented by the Lifeline is of type Class. This relation is derived from the Lifeline’s association “represents: ConnectableElement[0..1]” specified in [OMG, 2010]. This connects the Class diagram to the Sequence diagram.</td>
</tr>
<tr>
<td>State – Activity</td>
<td>This relationship is specified in [OMG, 2010]. It is an important relationship because in the case of event-driven software, a part of the process may be executed on occurrence of an event that may affect the execution sequence of the entire process flow. An additional constraint is defined for this relationship that will ensure the mapping between activity and state, thereby enabling the ability to track states/events/triggers that leads to execution of an out-of-sequence activity. <strong>Constraint: C4</strong>&lt;br&gt;<strong>Context:</strong> State&lt;br&gt;<strong>Inv:</strong> (State.entry</td>
</tr>
<tr>
<td>Deployment Diagram – Component Diagram</td>
<td>The deployment diagram imports the component diagram to establish a connection between the external hardware components and software components. This relationship assists in visualizing fault propagation from external components into the software system and vice-versa. This relationship is not a part of the UML Specification.</td>
</tr>
</tbody>
</table>
Table 1 continued

<table>
<thead>
<tr>
<th>Fault – Variable</th>
<th>This relationship is established to study the input/output types of faults associated with the software components. A fault is injected by manipulating the software variable values. This relationship is not a part of the UML Specification.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Fault – OpaqueAction</td>
<td>This relationship is established to distinguish the faults originating from the physical system with which the software interacts. This relationship incorporates the external faults by manipulating the variables, statements, etc. of the Opaque Action. This relationship is not a part of the UML Specification.</td>
</tr>
<tr>
<td>Fault – Message</td>
<td>This relationship is established to study message-related faults such as incorrect sequence of messages, missing message, etc. These types of fault are very common and can have a disastrous effect on overall system behavior. This relationship incorporates the message faults by modifying the message-related properties such as signature, parameters, order, etc. This relationship is not a part of the UML Specification.</td>
</tr>
</tbody>
</table>

The `behavioralRule` represents a novel concept introduced to study the input/output value related to failures such as value, range, type, and amount [Li et al., 2006]. Behavioral rules are presented as if-then-else condition statements, where the condition is defined as the relationship between the input and output variables of a particular component. Because these rules are defined by the analyst, there is no standard format; therefore, they are implemented as `Opaque Action`. The `behavioralRule` captures the nominal and faulty operation modes of a component. These modes are defined as relationships between input and output variables. The input variables associated with the component are transformed into output variables by the activities that the component represents. An incorrect action/decision execution of the activity will result in incorrect output values that will further trigger the component’s nominal or faulty modes. For example, consider a faulty execution of the decision node D1 \( (P_2 < P_{Lth}) \), where \( P_2 = 0 \) (a
value less than $P_{L_h}$). If the condition D1 is evaluated to “false” instead of “true,” then the variable “ControlCommand” will equal “Open.” In this case, the Faulty2 mode, as shown in Figure 4(c), is triggered.

The FFL is a powerful reasoning tool to determine the software functional effect resulting from different modes defined in the behavioralRules. The FFL of each software component is implemented as StateMachine. We can easily infer the system-level functional failure based on the results obtained from FFL. The software function is represented by the Activities that the component represents. Depending on the component mode, the Activity status will either be Lost, Operating, or Unknown. Furthermore, the low-level functional effect can be related to the high-level failure effect based on the relationship between Activity and Use Case. An Activity may be usedIn multiple Use Cases; therefore, one Activity failure may lead to multiple Use Case failures. Furthermore, according to the standard relationship between Use Case and Actor, a use case may provide an output to multiple actors that may represent an external component such as a HW_Component. Therefore, we can conclude that failure of an Activity may lead to failure of multiple Use Cases which in turn will affect the external component inputs. In this research, we limit the discussion to activity failure; however, we can further extend a formal deduction of use case failure simply based on the mapping relationship between use case and activity.
Figure 4 (a) The software component “Valve Controller” and its input–output variables (P2 and ControlCommand), (b) the activity “valve control logic,” (c) a sample behavioral rule in terms of the relationship between input–output variables, and (d) a sample function failure logic.

Table 2 Variable And Design Limitations Associated With Software Component “Valve Controller”

<table>
<thead>
<tr>
<th>Variable</th>
<th>Values</th>
</tr>
</thead>
<tbody>
<tr>
<td>P_in</td>
<td>P_valid = {P_{in}</td>
</tr>
<tr>
<td>ControlCommand</td>
<td>{Open, Close, Null}</td>
</tr>
</tbody>
</table>

Figure 4 and Table 2 demonstrate a simple example of behavioral rules and FFL for the software component “Valve Controller.”

In addition to input/output-value types of failure, other failure types can be studied. These include failure due to incorrect control logic or incorrect decisions; state-based
failures; communication-related failures such as incorrect sequence of events, object missing failures, and message missing failures.

**Behavioral Simulation**

The behavior simulation of the FPSA is a complex process since it accounts for diagrams of different nature. The UML-based design provides several options for behavioral simulation. The simulation can be driven by the state machine, the activity diagram, or the sequence diagram. Of these, the simulation driven by the activity diagram fulfils our need to simulate the overall system and does so better than the other options. The UML superstructure v2.3 states,

“All the behavior formalisms are potentially intra-object, if they are specified to be executed by and access only one object. However, state machines are designed specifically to model the state of a single object and respond to events arriving at that object. Activities can be used in a similar way, but also highlight input and output dependency between behaviors, which may reside in multiple objects. Interactions are potentially intra-object, but generally not designed for that purpose.” (OMG, 2009)

Each node of the activity diagram is traversed following the control flow edges. A node is mapped to the components, which contains the BehavioralRules and FFL. Execution of each node triggers the execution of the component and its elements. This is done to evaluate the status of the software function (i.e., activity and use case) at each node.
Results are propagated to other diagrams using the mapping relationships established earlier in table 1.

The FPSA execution engine is represented using the Event Sequence Diagram framework. The ESD [Swaminathan et al., 1999] is a modeling framework that allows the representation of the sequence of events occurring in an accident scenario. It has been used to build event trees and fault trees for qualitative and quantitative analysis. It provides elements that facilitate modeling of conditions, concurrent processes, mutually exclusive outcomes, synchronization processes, and other time-dependent processes. These processes model the dynamics of the events leading to a particular behavior. This helps in identifying and constructing sequences of action in a manner similar to a flowchart. In addition, ESD is a representation of actual event occurrence and the results of these events.

Software exhibits dynamic behavior because of event occurrences during the execution process. Software execution involves both static elements (such as classes, attributes, and components) and dynamic elements (such as actions, use-cases, and state transitions). The ESD framework is flexible enough to model situations ranging from purely static to dynamic. In table 18 (in the Appendix), the subset of ESD elements used in this research is described. The elements used in the execution algorithms are discussed next.

Elements of execution algorithm

The FPSA execution algorithm is based on the elements of UML semantic architecture particularly the elements of the action block. The action block defines the semantics of an “action” as the fundamental unit of any behavior. Actions are defined to execute fine-
grained behavior such as object creation (defined by CreateObjectAction), method
 invocation (defined by CallOperationAction), structural entity reading (defined by
 ReadStructuralFeatureAction), and so on. The elements of the action package are defined
 in chapter 11 of the UML superstructure [OMG, 2010]. A subset of these action elements
 is used to define the FPSA execution algorithm. The subset of actions describes the
 communication between objects. The communication between objects involves execution
 of particular actions. The Action Model is selected as the basis for execution engine of
 FPSA based on the premise that “all behavior in a modeled system is ultimately caused
 by actions (emphasis added) executed by so-called ‘active’ objects” [OMG, 2010].

Action model

![Diagram of the Action Model](image)

Figure 5 Meta-Model Of The Action Model Used In This Research

“Action” is an abstract class and a generalization of specific executable actions such as

CreateObjectAction, ReplyAction, InvocationAction, ReadSelfAction, and
AcceptEventAction. “Action” consists of InputPin which holds the input data or object necessary to execute an action. This relationship is represented by a solid diamond where the role of InputPin is represented as “argument.” “Action” is always executed in the context of a behavior. The association relationship between “Action” and “use-case” specifies that use-case is the context of the action. The use-case is a part of some “Classifier” which acts as the subject of the use-case. The subject of a use-case could be a physical system or any other element that has a behavior, such as a component, subsystem, or class [OMG, 2010]. Figure 5 defines the meta-model of the Action Model used for defining FPSA at the execution level. The elements of this metamodel are defined in UML specification. The important relationships and their meaning are discussed below.

- **InvocationAction** is an abstract class and a generalization of specific executable actions such as SendSignalAction, CallOperationAction, StartObjectBehaviorAction, and SendObjectAction.

- **CallBehaviorAction** consists of InputPin which holds the information of the target object. CallBehaviorAction is also associated with the Operation that is invoked by the action execution.

- **SendSignalAction** consists of InputPin which holds the target object to which the signal is sent.

- **SendObjectAction** consists of InputPin that can hold two types of objects: the “request” object, which is transmitted to the target object, and the “target” object, to which the request (or any other object) is sent.
• *StartObjectBehaviorAction* consists of *InputPin* which holds the behavior type of the object, i.e., an object which is either a behavior or a *BehavioredClassifier*.

• *ReplyAction* is executed only in the case of synchronous calls and accepts the values and information of the caller. It consists of *InputPin* that holds the set of return values accepted by this action, represented as *replyValue*. Secondly, the *InputPin* holds the value containing return information produced by a previous *AcceptCallAction*.

**Interaction model**

An “Interaction” depicts the communication between different objects. Interactions are well understood by both system designers and end users. They are used to describe the high-level function e.g. a use-case, or lower-level function e.g. an activity. Typically they are not used to describe the complete system. Interactions can be depicted in sequence, and interaction diagrams. Each diagram provides different information suitable for particular applications. The most commonly used features of the interaction are the sequence of messages passed and the object involved in the interaction. This aspect of message passing between objects is important in understanding the system and is well
represented with the help of a sequence diagram. The central concept of an interaction is
the concept of “trace.” The trace is a sequence of event occurrences. The message passing
between the objects can be represented in terms of events. For example, in figure 6, the
message createOrder is passed between objects “Customer” and “Order.” This message
passing can be represented using concepts such as OccurrenceSpecification, Lifeline, and
ExecutionSpecification defined for the Interaction Model. As shown in figure 6, a Lifeline
is associated to the objects “Customer” and “Order.” The message consists of two
message ends: send and the other (depicted by an arrow) is receive. The element
OccurrenceSpecification is used to represent the message ends and the message. The
action “Create an order number” is represented by an ExecutionSpecification.

![Figure 7 Meta-Model Of Interaction Model Used In This Research](image-url)
The communication between objects is a result of a sequence of event occurrences. The sequence of events follows a specific order. This ordering of the events is built from the communication model described in UML superstructure. This event ordering is represented in the form of an Interaction Model ESD.

Figure 7 shows the metamodel of the interaction model for defining FPSA at the execution level. The elements of this metamodel are defined in UML specification. The important relationships and their meaning are discussed below.

- The focus of the Interaction Model is on interactions. An interaction is a type of “Behavior.” The main elements of an interaction are “Lifeline” of an object, and “Message” passed between the objects.

- “Message” represents the message that is passed between the objects. A message is associated with a MessageOccurrenceSpecification which is a type of occurrence specification. A MessageOccurrenceSpecification provides information such as the send event and the receive event occurrences associated with the message.

- The OccurrenceSpecification are basic semantics of an interaction. OccurrenceSpecification’s are arranged in order along the “Lifeline” of an object. An OccurrenceSpecification is associated with an event and provides the details of the event occurrence.

- An interaction represents an execution scenario e.g. Use case. “Action” is one of the executable elements of the interaction. The details of the execution of an action are provided by the ActionExecutionSpecification. It is a type of Execution-Specification that provides information about the start and finish of the execution of an action.
An “Event” is an abstract class and represents any event occurrence. Different types of events can be found such as an `ExecutionEvent`, a `CreateEvent`, a `MessageEvent`, and a `DestructionEvent`. The `MessageEvent` is a generalization of more specific events such as the `SendOperationEvent`, `SendSignalEvent`, `ReceiveOperationEvent`, and `ReceiveSignalEvent`.

The sequence diagram is one of the interaction diagram used during the more details phase of software development and provides an implementation-level view of the software.

**Trace Model**

The Trace Model acts as the connection between the Action Model and Interaction Model. “Trace” is the central concept of an Interaction. Interactions capture the behavior of a Classifier. Information is passed between the different elements of the Classifier in the form of messages. “Interaction” consists of objects, messages, and owned actions as shown in figure 6. A trace is a sequence of event occurrences associated with the messages that are passed between different objects. Each event is described by an `OccurrenceSpecification`. Trace can be “valid,” or “invalid” or of another unknown type. Unknown traces are those that cannot be described by an Interaction. For simplicity, we only consider valid or invalid traces in this work.

The semantics of an Interaction are given by a pair \([P, I]\), where \(P\) is the set of valid traces and \(I\) is the set of invalid traces. A trace is a sequence of event occurrences denoted as \(\langle e_1, e_2, ..., e_n \rangle\). In our work, an event occurrence will also include information about the values of all relevant objects.
Each Interaction construct (such as sequence, switch, if-then, loop, etc.) can be expressed in terms of its relation to a set of valid and invalid traces. For simplicity, we refer only to the set of valid traces because these are mostly modeled.

**Relationship between Action and Interaction Model**

The Action Model handles all the actions, and the Interaction Model handles traces or sequence of events. The relationship between an action and an event which connects the Action Model to the Interaction Model is described in figure 8 and figure 9. A cause–effect relationship exists between an action and an event. In figure 9, an action invokes a behavior also termed “executing behavior” and the same action also causes an event occurrence. This event may act as a trigger of another behavior execution also called “emergent behavior.” It is important to note that “all behaviors are caused by actions executed by active objects, but actions do not necessarily cause an event and not all events are caused by actions” [15].

In the chapter “Common Behaviors” [OMG, 2010] we find an Execution/Action Model described by a sequence of action execution. The relationship between this Action Model and the Interactions Trace Model is shown in figure 8.

Figure 7 shows a sample of the relationship that exists between the Action Model and the Interaction Model. The figure shows that “action” is a generalization of a specific action such as InvocationAction, which in turn is a generalization of more specific action such as SendSignalAction and CallAction. Similarly, in the case of the Interaction Model, an “event” is a generalization of a specific event such as Message which is a generalization of more specific events such as CallEvent and SignalEvent. An action
causes an event or accepts an event. Fig 6 shows that a SendSignalAction causes a SignalEvent, a CallAction causes a CallEvent, and the action AcceptEventAction accepts an event. These relationships are derived from the semantic description of elements of the Action Model and the Interaction Model in the UML superstructure [OMG, 2010].

Figure 8 The Relationship Between The Action And Interaction Models.

Figure 9 The Relationship Between Action And Event

**Timing elements of simulation**

The action model and interaction model together ensure complete communication between the objects. In addition to communication, the timing is an important factor to determine the reliability and safety of a safety-critical system. Time facilitates model
simulation, which captures the evolution of system state and variables over time.

Evaluation of timing properties is essential not only for real-time systems but also for verification of timing constraints on synchronous processes. Based on the timing characteristics of a process or occurrence of a faulty event, certain fault propagation paths could be initiated or terminated, thus affecting the reliability of the system. For instance, in an accident scenario such as core-melt-down, an event occurring too late could be disastrous. The timing information captured by the timing elements of each model element during their execution are described in Table 3.

<table>
<thead>
<tr>
<th>Timing elements</th>
<th>Model elements</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Clock</td>
<td>-</td>
<td>Records the simulation time.</td>
</tr>
<tr>
<td>Start(Ts)</td>
<td>Action, operation</td>
<td>A starting time</td>
</tr>
<tr>
<td>End(Te)</td>
<td>Action, operation</td>
<td>An ending time</td>
</tr>
</tbody>
</table>

Table 3 Timing Elements Used In This Research
Chapter 4. FPSA Execution Model

The Failure Propagation and Simulation Approach (FPSA) includes a complex execution model to enable fault propagation through different UML diagrams automatically. The execution model is composed of two sub-models: 1) Action model ESD, which handles the execution of control-flow related elements; 2) Interaction model ESD, which handles the execution of a particular characteristic of object-oriented system i.e. the object-to-object communication.

The conceptual representation of the Action Model ESD is shown in figure 8. This model is executed for each execution step of the activity diagram simulation. The model captures information about who is responsible for a particular action’s execution, the generated events, and which objects are created and passed.

![Figure 10 A Conceptual Representation Of The Action Model ESD](image-url)
Within FPSA the simulation is activity diagram driven. The Action Model ESD built to support FPSA is initiated by an action and is iterative in nature. The ESD begins with the identification of the action to be executed. An action is ultimately a part of a method of a specific instance of a classifier referred to as a “host object.” Before the action is actually executed, a sequence of messages is passed between objects. These sequences of messages are represented by the Trace Model. According to the communication model of the UML superstructure, an object called “host” or “sender object” will invoke a method on another object called the “target object.” This target object will contain the action to be executed. This target object in turn becomes the host object because it hosts the action to be executed. For example, in figure 11, “Create an order number” is the action to be executed, “Customer” is the host object, and “Order” is the target object. The host object Customer invokes the method \textit{createOrder} on the target object Order. The target object contains the action “Create an order number” which is executed. The target object Order now becomes the host object.

In figure 12, the process of object-to-object communication is represented in a simplified manner by the portion that begins with “invocation” followed by creation of the “request” object which is “received by the target” leading to execution of the sent request.
For example, in figure 11, the object “customer” invokes the method `createOrder` of “order.” A request object (not represented) is generated for this invocation which is received by the target object “order” which further leads to execution of the invoked method.

Figure 12 depicts the concept of the Interaction Model ESD. The execution is initiated with the reading of OccurrenceSpecification. An `OccurrenceSpecification` defines any event occurrence. Depending on the type of event, different paths of execution are followed. For example, consider a case of send-receive communication between the objects. A host object sends a message; the target object receives the message and proceeds with the execution. In this case, the `OccurrenceSpecification` determines that a “Send Event” has occurred and the next step is to “Identify Message.” This message is received by the target object. Before the message is actually received by the target, there must be a receive event for that particular message. The receive event can be found in the
event pool. After identification of the message, the next step is to “Check EventPool” for the receive event. After “ReceiveEvent Found,” the target object proceeds to “Execution.” During this execution, events may be generated. These events will have an occurrence specification and, depending on the event, the ESD will take different paths. The above discussion shows that the Interaction Model ESD is an iterative process similar to the Action Model ESD.

Two ESD are developed to provide executable capability to the FPSA. These ESDs are iterative in nature and generate multiple threads during execution. These multiple threads are capable of representing the execution of an action under consideration, the execution of the next action, and the execution of the interaction model. The ESD in figure 13 consists of elements of the action meta-model and hence termed action model ESD. Action model ESD can also be viewed as a pictorial representation of an execution algorithm for the activity diagram. To execute the activity diagram, the trace for each action needs to be defined. The execution sequence of the action model ESD is explained in detail below:

The execution steps are described below. The numbers do not describe the execution sequence. There are concurrent/parallel paths in the execution that execute simultaneously.

1. The execution of action model ESD starts with the execution of initiating event i.e. “Execute the initiating trace”. The initiating trace collects all the data required to pass the control from the initial node in the activity to the following action. Following the initiating event is the AND1 gate which starts two parallel paths. One path leads to the
OR1 gate and the other leads to the process "Start clock and record the time" which starts recording the simulation time.

2. The OR1 gate is activated by one of the three inputs. First input is the output of AND1, second input is comes from the connector C3, and the third comes from the connector C5. Connectors C3 and C5 are a result of the iterative nature of the execution. Connector C3 is generated along the path p2 where the trace ends. This implies start of the execution of the next action and creation of a new execution thread. Connector C5 is generated due to the execution of the interaction model and acts as the communication point between the two execution models. For the first iteration, C3 and C5 are not generated and therefore ignored. OR1 has one output which leads to the process “Read the type of the element”. The element could be a fork, a join, a decision, or an activity. The execution of each element is different. The execution of an activity is explained next.

3. For an activity the path leads to the process “Read the properties of the action A(i)” process collects all the information (attributes and associations) related to the action A(i). The information collected is shown in the comment box following this process. The information from the above process provides the “context” of the action.

4. The next step is a condition check: “Is the context already instantiated?”
   a. If Yes, move to OR2.
   b. If No, the condition leads to the process “Execution of StartObjectBehaviorAction”. This process instantiates the behavior represented by the context i.e. the Use case according to our metamodel.
For the first iteration, the condition is evaluated to “No.”

5. OR2 has one output that leads to OR3, which can be activated either by C2 or output from OR2. During the first iteration, C2 is unavailable and therefore ignored. OR3 has a single output that leads to the process “Read Trace”, fetches the trace information associated with the action A(i). The output of this process is a “Message m(i)” which is represented in the comment box following the process. The process also marks the beginning of the execution of the action A(i), thus the start time of action A(i) must be recorded. Thus the path splits into two concurrent paths p2 and p3 at AND2 gate. P2 leads to the process “Record start time of the action A(i)” and ends after the process. P3 leads to a condition check “End of trace?” which checks if the events defined in the trace have been exhausted or not.
   a. If Yes, it marks the end of execution of action A(i) and a new thread must be generated to start the next iteration. Along the ‘Yes’ path the AND3 gate creates two concurrent paths p4 and p5. P4 leads to the process “Record end time of the action A(i)” and ends after the process. P4 leads to C3, which further leads to OR1 and a new thread for next action begins executing as described in Step 2.
   b. If No, the condition check leads to OR4.

6. OR4 is a multiple output gate which leads to one of the three paths p6, p7, or p8. The path selection depends on the type of event associated with the message. In case of send operation event path p6 is selected, in case of send signal path p8 is selected, and in case of receive event path p7 is selected. Execution of each path is explained in detail below.
6.1. **Path p6**: Path p6 leads to the process “Execution of CallOperationAction.” The information obtained from this process is represented in the comment box. This information is used by the following process “Generate Request class.” This process generates a request class with attributes represented in the following comment box. This request class is further instantiated by executing the process “Execute the CreateObjectAction on Request class.” The object created by the process is represented in the following comment box. Once the object is created, the next step is to send the object to the target object for execution of the request. There are a number of steps involved between sending the request object to the target and execution of the request by the target. First, upon creation of the request object, the process “Execute SendObjectAction on Request class” is performed. This process leads to the AND4 gate with two concurrent outputs represented by paths p9 and p10.

**Path p9**: The request object is sent to the target object, as given in the comment box. This leads to a condition check. The condition check “Is isSync = true?” applies to the operation that was invoked by the host object.

a. If evaluated to **Yes**, it implies that the operation call is synchronous, i.e., the host object waits for the reply from the target object. This path leads to the OR5 gate which further leads to execution of the target object action, and execution of the reply action.
b. If evaluated to No, the host object continues its execution regardless of the execution of target object. This development is modeled as the output of AND6, which creates two concurrent paths p14 and p15. Path p14 leads to the connection C2 which is connected to OR3 thus indicating that the execution of trace continues without waiting for current message to finish its execution.

Path p15 leads to OR5 which further leads to execution of the target object.

**Path p10:** The purpose of this path is to generate a sendEvent which is handled by the Interaction Model ESD and to update the event pool. The event pool is similar to a stack or a queue which stores information related to all the events that occur during the execution of the software. Path p9 leads to the process “Generate a SendOperationEvent” which generates a send event. The next process is to “Create an OccurrenceSpecification” which is used by the interaction model ESD to start its execution. The path p9 is further divided into three paths (p11, p12, and p13) at AND6. Path p11 leads to the process “Update the event pool of the Host object” and ends at “End of path.” Path p12 leads to C1 which acts as the communication link between the action model ESD and the interaction model ESD. Along path p13, the time of occurrence of send event is recorded by the process “Record time of occurrence of receive signal”.

Paths p9 and p15 lead to a multiple input OR5 gate whose output leads to the process “Execute the AcceptCallAction.” Completion of this process indicates that the target object is ready to receive the request object. The process further leads to a multiple output AND7 which creates three concurrent paths p16, p17, and p18.
Path p16 leads to the process “Update the trace with receive message” and ends at the “End of path”. Path p17 leads to the AND8 gate which acts as the synchronization point. The synchronization occurs between the current execution thread and the thread of execution, identified by the connector C5, originating from the interaction model ESD. The output of AND8 leads to the process “Collect the return information of the execution”, which collects any return information passed by the interaction model. The path further leads to a condition check “Is isSync = true?”. If the condition evaluates to No, the execution thread ends at “Execution complete”. If the condition evaluates to Yes, the process “Execute ReplyAction” is executed, which passes the control back to the calling object supplying the return information. Further AND9 create two paths p21 and p22. Path 21 marks the “end of the path p6”, while p22 leads to connector C2, which further leads to the process “Read trace” for further execution.

Path p18 is similar to p10 explained earlier. The path p18 leads to the process “Generate a ReceiveSignalEvent” which generates a receive event indicating the request was received by the target object. Next, the process “Create an OccurrenceSpecification” creates an occurrence specification of receive event. Further AND10 gate divides p18 into two concurrent paths p19 and p20. Path p19 leads to the process “Record time of occurrence of receive signal” which records the occurrence time of the receive event. Path p20 leads to the process “Update the event pool of the target object”, which updates the event pool with the receive event, and ends at “End of path.”
6.2. **Path p7**: It is selected if a receive Message m(i) is associated an external trigger. In such case, the execution cannot proceed until the external trigger generated by some external hardware such as a sensor or a timing event. This requires a synchronization point. Further the external signal is always treated asynchronously, i.e., the host object continues its execution irrespective of the execution of the target object. These requirements are resolved by AND11 gate. AND11 serves two purposes, one it acts as the synchronization point and also creates three concurrent paths p23, p24, and p25. Path p23 leads to C2 which connects to OR3, where the next message from the trace is ready to be executed, thus fulfilling the asynchronization condition. Path p25 records the time of occurrence of the external event and ends after the time is recorded. Path p24 leads to the process “Execute the AcceptEventAction” via OR6 gate. The process accepts any event such as a signal or message that occurs during the execution of the software. The output of this process consists of the information of the trigger caused by the event and the result which provides the details of the signal and its attributes. Once the event is accepted, the next step is to generate the receive event and execute the operation on the target object. AND12 creates two concurrent paths p26 and p27.

**Path p26** leads to a synchronization point AND13, a multiple input gate. AND13 synchronizes the current execution thread with the thread (C5) originating from interaction model, which executes the behavior associated with the receive signal. The output of AND13 leads to the process “Collect the return information of the
execution” that further leads to AND14. AND14 created two paths p30 and p31. Path p30 leads to connector C2 which connects to OR3 and indicates a continuation of the execution of the next trace message.

**Path p27** leads to the generation of the receive event and is identical to path p9 explained earlier. The path p27 is further divided into two paths p28, and p29 at AND15. Path p29 leads to the process “Update the event pool of the Host object” and ends at “End of path”. Along path p28, the time of occurrence of receive event is recorded by the process “Record time of occurrence of receive signal”.

6.3. **Path p8**: It is selected when the execution of the host object involves the sending and receiving of the operation stereotyped ‹‹signal››. The send signal is an asynchronous call, i.e., the host object sends the signal and continues its execution regardless of the execution of target objects. The process “Execution of SendSignalAction” creates a signal instance that is transmitted to the target object. Completion of this process provides the attribute related to the signal object and the target object. This information is given in the following comment box. The next process “Execution of SendObjectAction” sends the signal object to the target object. This process leads to AND16 which divides the path into three concurrent paths p32, p33, and p34. Path p32 indicates the signal object was sent and leads to OR6 gate. Path p34 leads to generation of the send event and is similar to path p10 (explained earlier). Path 33 captures the asynchronous nature of the send signal. Path p33 leads to C2 which connects to OR3 and indicates a continuation of the execution of the next trace message.
The execution engine creates several parallel threads of execution via connector C2. To avoid unnecessary thread creations, a restriction of one C2 connector per execution thread is imposed. Such a restriction will avoid any accidental or out of order execution of activities, thus preserving the order of activity execution.
Figure 13 Action Model ESD
Figure 14 Interaction Model ESD

Figure 14 is based on the sequence diagram elements and represents handling of the sequence of events raised during the execution of actions in the Action Model ESD. The interaction model considers the simple send-receive communication between the objects.
Complex semantics such as *CombinedFragments* (e.g., sequence, if-else, and switch) are not a part of this model. The interaction model elements presented earlier in the metamodel are used to build the Interaction Model ESD for executable FPSA.

Following is the detailed explanation of figure 14.

1. The execution of the interaction model starts from the occurrence of an event. The event can be an external event or one generated from the execution of an action model as explained earlier. The initiating event for the interaction model is “Read *OccurrenceSpecification*” captures the details of any event occurrence and is triggered by C4, C1, or the “External Signal Trigger.” This is modeled as three inputs to the OR1 gate. One input C1 originates from the execution of the action model ESD to communicate with the interaction model. During the execution, the action model ESD raised events such as *SendSignalEvent*, *SendOperationEvent*, and *ReceiveSignalEvent*. Each of these events has an *OccurrenceSpecification* associated with it. Second input C4 is internal to the interaction model ESD and is generated during its execution. During the first iteration, C4 is not present and therefore ignored. Third input is external trigger caused by any external event e.g. a sensor signal, time out, or switch signal.

2. After reading the occurrence specification, a condition check “Event found?” is performed.

   a. If evaluated to *Yes*, the path leads to the process “Read Event” which reads the details related to the event. The process leads to a condition check “Type of Event...
E(i)”. Depending of the type of event read one of the available paths is selected for further execution.

b. If evaluated to No, the execution ends at “End of execution”.

3. If the event type is $\text{SendOperationEvent}$, the process “Read event” will return the information about “Operation” (represented in the comment box along the path). This path leads to OR2 gate. If the event type is $\text{SendSignalEvent}$, the name of the signal is returned and the path leads to OR2 gate. If the event type is $\text{CreationEvent}$, the path leads to the process “Create an object” e.g. a request object or a signal object and further to OR1 via the C4 connector. If the event type is $\text{DestructionEvent}$ the path leads to the process “Destroy the object” and terminates at “Object deleted” end node.

If the event type is a $\text{ReceiveEvent}$, the path leads to OR3 gate.

4. OR2 gate leads to the process “Read the $\text{MessageOccurrenceSpecification}$”, which returns a message and event as shown in the comment box. Once the message is identified, the process “Read $\text{Message}$” provides all the details associated with the message. The output of this process is represented in the following comment box.

Once the message details are identified, the next step is to check the event pool for a receive event corresponding to the send event. The condition check “$\text{ReceiveEvent}$ occurred for Receiver (j)?” for the receive event is performed on the event pool. If the condition evaluates to $\text{Yes}$, the path leads to OR4. If the condition evaluates to $\text{No}$, the execution enters a loop till it finds the receive event in the event pool or till a timeout event occurs. The loop consists of the process “Read event pool” followed by the
5. The OR4 gate leads to the process “Read OccurrenceSpecification for object (j).” The output of this process is represented in the following comment box. The receive event occurrence indicates that the target object is ready to execute. Thus process “Read ExecutionOccurrenceSpecification” is performed to mark the start/finish of the execution and to determine the ExecutionSpecification. Next is a condition check “Type of Execution Event?”

a. If the condition evaluates to Start, the path leads to AND1 gate which creates two concurrent executions. One path leads to the process “Record start time of the operation execution” and ends thereafter. Second path leads to the process “Read ExecutionSpecification”, which provides the action/behavior to be executed. Further the process “Execution of the Action/Behavior” executes the behavior of interest. The behavior can be a message, an action, or a state machine. The behavior execution may further involve message passing and thus more events will be generated. These additional messages are handled in separate thread generate by AND2 gate. The process leads back to the process “Read ExecutionOccurrenceSpecification.”

b. If the condition check evaluates to Finish, the thread of execution leads to AND3 gate which creates three concurrent execution paths. One path leads to the process “Record end time of operation execution” and ends thereafter. Another path leads to connector C5, which leads to the action model execution’s thread at AND11
gate thus indicating that execution the required behavior is complete and the action model can continue with the execution of the thread. And the last path leads to the end of the execution of the interaction model.

**Case Study**

![Image of Hold-Up Tank System Example](image)

Figure 15 Hold-Up Tank System Example (extracted from [Mutha et al. 2012])

In this section we demonstrate the FPSA technique using the “Hold-up Tank” system, a hypothetical case-study. The case-study is adapted from [Mutha et al. 2012] and further extended to enable low-level software model execution. As shown in the figure 15, the system consists of an inlet and an outlet valve, associated position sensors, pipes, a tank, a pressure sensor, a computer controller (software component). The system also includes a backup system, which consists of a backup pump and a limited-capacity reservoir. The
main function of this system is to regulate a steady flow of water from the tank to the pipe. Water enters the tank via inlet valve and exits the tank via outlet valve. The command to open and close the inlet and outlet valves is received from the computer controller. The computer controller implements the valve control logic, which maintains the desired water level in the tank. We assume a failure scenario will occur if the water supply is lost for more than 5 units of time. A backup system is provided as a safeguard. However the backup system has limited availability. We ignore the backup system to simplify the analysis.

We consider two system failure modes “Tank dry”, and “Loss of Regulate Fluid” in our discussion. The “Tank Dry” failure mode is triggered when the water level in the tank goes below the lower threshold and the outlet valve is open. Second, the “Loss of Regulate Fluid” failure mode is triggered when the outlet valve remains closed. Note, that “Tank Dry” can also trigger the “Loss of Regulate Fluid” failure mode.

The software design models in figure 16-23 are developed for FPSA demonstration. We have developed two different designs, Design 1 and Design 2, for discussion purposes. The demonstration includes three faulty cases enumerated below.

1. Three structural faults- Missing class/object, Missing message, and Missing attributes. This case is discussed with the help of Design 1.

2. One behavioral fault- A behavioral fault, Missing Activity, and its effect will be analyzed. Different effects are observed for different designs. This case is discussed with the help of Design 1 and Design 2.

3. One combination of fault-Missing Activity and Extra message.
Each case is analyzed in detail and results are discussed. The discussion covers different perspectives like propagation paths, their effect, safeguards or fault tolerance measures for each fault, and comparison between effects of different faults.

Figure 16 Component Diagram

Figure 17 Activity Diagram 1
(extracted from [Mutha et al. 2012])
Figure 18 Activity Diagram 2- Valve Control Logic

Figure 19 Use Case Diagram
Figure 20 Class Diagram

Figure 21 Sequence Diagram For Configure System Use Case
Figure 22 Sequence Diagram For Capture Sensor Data Use Case

Figure 23 Sequence Diagram For Control Inlet Valve And Control Outlet Valve Use Cases
Case 1.1- A fault missing Tank class

A missing Tank class fault analysis presents interesting insights and results that will be compared with other cases. Intuitively, the design of Tank class is apparent and hard-to-miss. Nevertheless, this case will aid in understanding the FPSA execution framework. Also, a comparison between this hard-to-miss fault and other possible faults will aid the severity analysis of the seemingly harmless faults.

The simulation begins with the initial (or nominal) condition of the system being the water level in the tank within desired range ($L_L, L_U$), and the inlet and the outlet valves being open. The missing tank fault is injected in the design and the simulation results are presented in Table 2. The FPSA simulation process for read pressure and calculate level activities is illustrated next.

High-level FPSA execution

During the first simulation run, the execution of the activity diagram (figure 17) begins with the “start” and executes all the activities till Valve Control logic. The Valve control logic activity is decomposed into other activities (figure 18). For this execution of the activity, the execution traces the path

\[ D_1 \rightarrow D_5 \rightarrow D_9 \rightarrow D_{10} \rightarrow \text{Exit} \]

Along the path, the first activity is configure system (Figure 17). Its enclosing component is configuration manager, which has no inputs, and one output i.e., ConversionData (Table 2). The activity configure system is executed, which may cause a change in the output variable value. Thus, the activity execution may trigger a nominal or
a faulty behavior of the component. Since no faults were injected in this component or activity, its behavioral rule indicates a nominal behavior. Further, the FFL is executed to determine the activity (function) status. The FFL indicates the configure system is operating (O) (see Table 2).

After the configure system the control flow branches into two parallel flows at the fork (see Figure 17). The first branch contains activities: Read pressure, and Calculate Level. The second branch contains activities: Read position, and Store Pos.

Along the first branch the read pressure is executed. Its enclosing component is Sensor, which has two inputs: pressure, position of inlet valve, and position of outlet valve (stored as an array Pos [I, O], where I implies input, and O implies output), and three outflows: P, Level and Pos (Table 2). The activity read pressure is executed, and the input pressure is read from the pressure sensor. The read pressure activity may modify the output variables value. So, the behavioral rule of the sensor is executed, and a nominal behavior Nom1 is observed (Table 2). For Nom1, the FFL indicates the read pressure is operating (O).

The next activity is calculate level, which is also implemented by the component sensor. Calculate level is executed followed by execution of the behavioral rule of sensor. The result indicates a faulty behavior faulty3 (Table 2) was triggered. The faulty behavior is attributed to the variable level containing a NULL value. The level has NULL value, due to a missing tank class. The tank object was supposed to calculate the level (as will be demonstrated in lower-level execution). For the faulty3 behavior, the FFL indicates the calculate level status is lost (L) (Table 2).
Along the second branch *read position* is executed. The corresponding component is *sensor*, which has two inputs: pressure, position of inlet valve, and position of outlet valve, and three outflows: P, Level and Pos (Table 2). The activity *read position* is executed, and the position of inlet valve and outlet valve are read from the respective position sensors (Pos [Open, Open]). *Read position* activity may modify the output variables values. So, the behavioral rule of the component *sensor* is executed, and a nominal behavior Nom 2 (Table 2) is observed. For Nom 2, the FFL indicates the *read position* is operating (O).

The next activity is *store Pos*. This activity is implemented in the *sensor*. *Store Pos* is executed which modifies the output variables value. So, the behavioral rule of the component *sensor* is executed, and a nominal behavior Nom 2 is observed (Table 2). For Nom 2, the FFL indicates the *store Pos* is operating (O).

The two parallel branches are synchronized before transferring the control to the next activity *Valve control logic*. The valve control logic implemented in the component *Valve controller* (Figure 17), which has three inputs: pressure (P), position of inlet valve and position of outlet valve (Pos), and level, and one output: ControlCommand (Table 2).

The execution of Valve control logic traces the path (Figure 18). The decision D1 evaluates to false (N) since the level is NULL (as discussed in *calculate level* activity execution). D1 leads the control flow to the decision D5, which also evaluates to false (N) since level is NULL (Note: We assume that for a NULL value of a variable, a decision involving that variable is always evaluated to false). D5 leads to the decision D9. D9 evaluates to true (Y) since the input valve position read was ‘open’.
D9 leads to a fork which split the control flow into two concurrently executing branches. The first branch leads to the decision node D10. D10 evaluates to true (Y) since the outlet valve position read was ‘open’. D10 leads to exit and the execution of one branch ends. The second branch leads to an activity *Close inlet valve*. The activity is implemented in the component *ValveController*, which has three inputs: P, Pos, and level, and one output: ControlCommand (Table 2). The activity *Close inlet valve* is executed and leads to closing of the inlet valve (HW component). On execution of the behavioral rule of the *ValveController* a faulty 1 is observed (Table 2). For faulty 1, the FFL indicates the *Control valve logic* status is lost (L). After the *close inlet valve* activity, the control is returned back to the *Valve control logic*. And the control flow enters a cycle creating a loop starting at the first fork (in Figure 17). As all the activities are covered the first simulation run ends and a second simulation run begins.

The second simulation begins at the fork which splits the control flow into two concurrent executions. These parallel executions are synchronized and directed to the *Valve control logic*. The execution up to the *valve control logic* is same as the first simulation run. However, the execution of the *Valve control logic* traces the path **D1-D5-D9-Exit**, which is different from the first simulation run. The decision D1 evaluates to false (N) since the level is NULL, as discussed in *calculate level* activity execution. D1 leads the control flow to the decision D5, which also evaluates to false (N) since level is NULL. D5 leads the control flow to the decision D9, which checks whether the inlet valve position is open. D9 evaluates to false (N) since the input valve position is ‘close’ (recall the first simulation run executed the activity *close inlet valve*). D9 leads the
control flow to the exit, and the control is returned to the Valve control logic. On execution of the behavioral rule, the ValveController is found to exhibit a faulty behavior. Next, the execution of sensor’s FFL indicates the Control valve logic status is lost (Table 2). The control is returned to the Valve control logic, which directs the control flow to a cycle creating a loop starting at the first fork (in Figure 17). As all the activities are covered the first simulation run ends and a second simulation run begins.

The path followed in the second run is maintained for all the subsequent runs. As a result, the Control valve logic always exhibits a lost status and is never recovered. The results of high-level execution for 10 simulation runs are presented in Table 4. The impact of this software behavior at system level is detrimental. At system level, the “Tank Dry” failure mode is triggered, since the inlet valve remains closed. “Tank Dry” further triggers “Loss of Regulate Fluid” failure mode after 5 units of time.

The high-level execution only presents the results of functional perspective. It does not truly capture the underlying mechanism of failure or the software behavior. It does not capture the objects and messages that were passed, the time at which messages were passed etc. The lower-level execution framework captures the implementation level details that aid in predicting the issues that may arise if a particular design is implemented.

Low-level FPSA execution
For each activity along the path discussed in high-level execution, an underlying execution mechanism exists. This underlying mechanism represents the sequence of events that occur. The sequence of events is implementation specific. In other words, different implementations will produce different sequence of events and consequently different functional failures may be observed at the higher-level. The ability to evaluate the pros and cons of different implementations from system perspective gives a huge economical advantage.

Every activity is associated with a valid trace specification. A trace is derived from the messages passed between different objects in a particular sequence diagram. A trace is a set of \(<\text{Send event (!)}, \text{Receive event (?)}>\) associated with the messages. The activity and their respective valid trace specification are presented in Table 3.

The low-level execution begins with \textit{start} to \textit{Valve control logic} followed by the path
\begin{center}
\[ D1 \rightarrow D5 \rightarrow D9 \rightarrow \text{Close inlet valve} \rightarrow D10 \rightarrow \text{Exit} \]
\end{center}
(recall this path was observed during the first run).

At the \textit{start} (Figure 17), the initiating trace was executed. Since there is no initial trace, we execute the trace associated with the next activity \textit{configure system} followed by \textit{read pressure} and \textit{calculate level}. \textit{Configure system} has no faults and its execution has no issues, as such no interesting observations were recorded. On the other hand, the execution of \textit{Read pressure} and \textit{calculate level} has more interesting observations, hence discussed in detail next.
Figure 24: Thread TH1 of the execution of Activity Read pressure.
Figure 25: Thread TH2 of execution of Activity Read pressure.
Figure 26: Thread TH3 of the execution of activity Read pressure.
Figure 27: Thread TH4 of the execution of activity Read pressure.
The action model ESD and interaction model ESD (figure 24-27) represents the execution of the *Read pressure*. The ESDs work in collaboration with each other. The algorithm is multi-threaded.

The execution of the action model ESD i.e. **TH1** (figure 24) begins at C3. The connector C3 was established from the execution of the previous activity, *configure system*. C3 leads to OR1, which has one output. The output of OR1 leads to the process “Read the type of the element” followed by a condition check “Type?” The condition check evaluates to activity that leads to the process “Read the properties of the action *read pressure*” and its output is given in the following comment box. The comment box leads to an OR2 gate.

Gate OR3 can receive another input from C2, but this is the first iteration of the execution of *read pressure*, C2 is not established and is hence ignored. The output of OR3 leads to the process “Read Trace.” The output of this process is the message *getdata* and the event is of type receive. The flow then leads to AND2 gate which creates paths p2 and p3. The path p2 records the timing information. Path p3 leads to the condition check “End of trace?” which evaluates if all the events contained in the trace were read. Since, *getdata* is the first element of the trace, the condition evaluates to false (No) and the flow leads to the OR4 gate.

OR4 is an output OR gate with three possible outputs: p6, p7, or p8. Since *read pressure* involves an external signal trigger from the pressure sensor (HW component), path p7 is selected, which leads to AND11. Note that the external trigger has also started the execution of Interaction ESD (thread **TH2**), which is discussed after the execution of
action ESD. The output of AND11 creates three concurrent paths: p23, p24, and p25.
Path p23 leads to the creation of a new execution thread TH3, path 24 is a part of TH1 execution, and path p25 records the time of event creation.

Path p23 (TH3) starts a cycle by creating a connection C2, an input to OR3 gate. Thus, the next element in the trace is read i.e. !convert. Path p24 (TH1) leads to OR6 and its output lead to the process “Execute AcceptEventAction.” The output of this process is a set of attributes of the accepted trigger, i.e., «getdata». The flow further leads to AND12. The outputs of AND12 are two concurrent paths: p26 and p27.

Path p26 leads to an AND13 gate, where the execution (TH1) waits for Interaction ESD to finish the execution thread TH2, and create a connection C5. Once the connection is established, the information from TH2 execution is collected by the process “Collect the information of the execution”. The process then leads to AND14, which creates two concurrent paths: p30 and p31. Path p30 starts a cycle by creating a connection C2, which is an input of OR3 gate. Note that OR3 ignores this input C2, since it was already created along p23 and the next element of trace read. Path p31 indicates the end of the thread TH1.

Path p27 leads to the process “Generate a ReceiveSignalEvent” which creates a ReceiveSignalEvent for getdata. The process directs to “Create an Occurrence-Specification” which creates the OccurrenceSpecification for the receiveEvent. This process leads to AND15 which has two concurrent paths: p28 and p29. Path p28 leads the process “Record time of occurrence of receive signal”. Path p29 leads to the process “Update the event pool of the pressure sensor object”. Thus the event pool of pressure
sensor is updated with the receiveEvent for *getdata*. Path p27 creates no active threads and is terminated at the “End of path.”

The execution of the Interaction Model ESD i.e. **TH2** (Figure 25) begins with the initiating event “Read OccurrenceSpecification”. The initiating event is triggered by one of the inputs of the OR1 gate, “External Signal Trigger” from *pressure sensor* (HW component). The “Read Occurrence-Specification” returns an event receive event as shown in the comment box. The next element is a condition check “Event found?” The condition evaluates to true (Yes) and the path leads to the process “Read Event.”

The process “Read Event” is followed by the condition check “Type of Event E(i).” The condition leads to selection of “ReceiveEvent” path. The path further leads to OR3. The output of OR3 leads to a condition check “ReceiveEvent occurred for Receiver (j)?” The receiver object in this case is *pressure sensor*. Recall, the **TH1** generated a ReceiveSignalEvent i.e. $?getdata$ along path p27. Thus the condition check evaluates to ‘Yes’, which leads to the OR4 gate.

The output of the OR4 gate leads to the process “Read OccurrenceSpecification for *Pressure sensor*.” The output of this process is given in the following comment box. Once the target object, *pressure sensor*, receives the signal, *getdata*, execution of the target object begins. The next process is “Read ExecutionOccurrenceSpecification.” The output of this process is given in the following comment box.

Next is a condition check “Type of ExecutionEvent?” Since this is the *start* of the execution of *getdata*, the flow leads to the AND1 gate. AND1 creates two concurrent paths. One path records the start time of the operation execution. The other path leads to
the process “Read ExecutionSpecification”. The execution specification may be a state machine, action, or piece of code. We assume the behavior is a piece of error-free code. The flow further leads to the process “Execution of the Action/Behavior initiated” and then to AND2 gate, which creates two concurrent paths. One path creates a cycle that leads to the process “Read ExecutionOccurrence Specification”. Since the execution of the code (specification) was complete, the process outputs a finish event. The flow then leads to the condition check “Type of Execution Event?” which leads to the finish path. The finish path leads to an AND3 gate. One of the paths creates a connector C5, which leads to the action model ESD. Recall, the action model ESD execution was waiting at AND13 for Interaction model ESD to finish the execution of the message and send the control back to the action model. The execution of TH2 ends here. While TH3, created during the TH1 execution is in active status.

TH3 execution (Figure 26) starts at OR3, which is triggered by the input C2. Recall, C2 was created during the action model execution (along p23). TH3 reads the next element in the trace i.e. !convert. TH3 execution up to OR4 gate is similar to the TH1 execution. However, in TH3, the path p6 is selected since convert is a call operation and not a send signal nor an external trigger.

Path p6 leads to the process “Execution of CallOperationAction.” The information obtained from this process is represented in the comment box. This information is used by the following process “Generate Request class.” This process generates a request class with attributes represented in the following comment box. This request class is further instantiated by executing the process “Execute the CreateObjectAction on Request class.”
The object created by the process is RQ_CONVERT. Once the object is created, the next step is to send RQ_CONVERT to the pressure sensor for execution of the request. There are a number of steps involved between sending the request object (RQ_CONVERT) to the target (pressure sensor) and execution of the request by the target. First, the process “Execute SendObjectAction on Request class” is performed. This process leads to the AND4 gate, which creates two concurrent paths p9 and p10.

Path p9: The request object is sent to the target object and leads to a condition check.

The condition check “Is isSync = true?” evaluates to true (Yes) and the path leads to the OR5 gate which further leads to execution of the target object action, and execution of the reply action.

Paths p9 further leads to a multiple input OR5 gate whose output leads to the process “Execute the AcceptCallAction.” Completion of this process indicates that the target object is ready to receive RQ_CONVERT. The process further leads to a multiple output AND7 which creates three concurrent paths p16, p17, and p18.

Path p16 leads to the process “Update the trace with receive message” i.e. \( ?\text{convert} \), and ends at the “End of path”.

Path p17 leads to the AND8 gate which acts as the synchronization point. The synchronization occurs between the current execution thread (TH3) and the thread of execution (TH4) i.e. the connector C5, originating from the interaction model ESD. The output of AND8 leads to the process “Collect the return information of the execution”, which collects any return information passed by the interaction model. The path further leads to a condition check “Is isSync = true?”, which evaluates to Yes. Thus, the process
“Execute ReplyAction” is executed, which passes the control back to the calling object and provides the return information. The path further leads to AND9, which creates two concurrent paths p21 and p22. Path p21 marks the end of TH3 executions. Path p22 starts a cycle by creating another thread TH5 (C2). C2 leads to OR3 gate the process “Read trace”. But, the end of trace is reached, hence thread TH5 execution is terminated and a new series of execution threads will be created for the next activity Calculate level.

Path p18 is similar to p9 explained earlier. The path p18 leads to the process “Generate a ReceiveSignalEvent” which generates a receive event for convert. Next, the process “Create an OccurrenceSpecification” creates an occurrence specification of receive event. Further, AND10 gate divides p18 into two concurrent paths p19 and p20. Path p19 records the occurrence time of the receive event. Path p20 leads to the process “Update the event pool of the target object”, which updates the event pool of pressure sensor with the receive event !convert.

Path p10: It leads to the process “Generate a SendOperationEvent” which generates a send event for convert. The next process is to “Create an OccurrenceSpecification” which is used by the interaction model ESD to start its execution. The path p10 is further divided into three paths (p11, p12, and p13) at AND5. Path p11 leads to the process “Update the event pool of the Host object” and ends at “End of path.” Path p12 leads to connector C1, which creates a new execution thread TH4 and acts as the communication link between the action model ESD and the interaction model ESD. Along path p13, the time of occurrence of send event is recorded.
The execution of the Interaction Model ESD (Figure 27) i.e. TH4 begins with the initiating event “Read OccurrenceSpecification.” The initiating event is triggered by one of the inputs of the OR1 gate i.e. C1 created in TH3 along path p12. The “Read OccurrenceSpecification” returns an event SendOperation, given in the comment box. The next element is a condition check “Event found?” The condition evaluates to true (Yes) and the path leads to the process “Read Event.”

The process “Read Event” is followed by the condition check “Type of Event E(i).” The path “SendOperationEvent” is selected since the event is a send operation i.e. !convert. The path further leads to OR2. The output of OR2 leads to the process “Read MessageOccurrenceSpecification.” The output of this process is given in the comment box.

The next process is “Read Message”, which further leads to OR3 gate. The output of OR3 leads to a condition check “ReceiveEvent occurred for Receiver (j)?” The receiver object in this case is pressure sensor. Recall, the TH3 generated a ReceiveSignalEvent i.e. ?convert along path p18. Thus the condition check leads to the OR4 gate.

The output of the OR4 gate leads to the process “Read OccurrenceSpecification for object (j).” The output of this process is given in the following comment box. Once the target object pressure sensor receives the signal convert, execution of the target object begins. The next process is “Read ExecutionOccurrenceSpecification.” The output of this process is given in the following comment box.

Next is a condition check “Type of ExecutionEvent?” Since this is the Start of the execution of convert, it leads to the process “Read ExecutionSpecification” via AND1,
which also records the start time of the operation execution. The execution specification may be a state machine, action, or piece of code. We assume the behavior is a piece of error-free code. Further the process “Execution of the Action/Behavior initiated” is executed and leads to completion of the execution specification. After completion the cycle leads to the path Finish. The finish path leads to an AND3 gate, which has three concurrent paths. One path creates a connection C5. C5 leads to the synchronization point AND13 in thread TH3. Other path marks the end of TH4.

Thus all the threads TH1, TH2, TH3, TH4, and TH5 run to completion and no errors, or incomplete execution was observed. TH3 created a connection C3 along path 17 leading to execution of next activity i.e. calculate level (not shown in the figure). Thus, a new series of execution of threads for calculate level will be created and discussed next.
Figure 28: Thread TH1 of execution of activity Calculate level
Figure 29 Thread TH2 Of Execution Of Activity Calculate Level And The Corresponding Path In Activity Diagram
The execution thread (TH1) is created for calculate level. TH1 execution (Figure 28) is initiated by the connector C3. C3 leads to OR1, which has one output. The output of OR1 leads to the process “Read the type of the element”, which indicates the element is an activity. Further the condition check “Type?” chooses the path marked activity. The path leads to the process “read the properties of the action calculate level” and the output is given in the following comment box. The comment box leads an OR3 gate.

Gate OR3 can receive another input from C2, but for the first iteration of the execution of calculate level, C2 is not established and is ignored. The output of OR3 leads to the process “Read Trace.” The output of this process is the message calcLevel and the event is of type send. Next is the condition check “End of trace?” which checks if all the events contained in the trace were read. Since, !calcLevel is the first element of the trace, the condition evaluates to false (No) and flow leads to an OR4 gate.

OR4 is an output OR gate with three possible outputs: p6, p7, or p8. The path p6 is selected since !calcLevel is a call operation and not a send signal nor an external trigger.

Path p6 leads to the process “Execution of CallOperationAction.” The information obtained from this process is represented in the comment box. This information is used by the following process “Generate Request class.” This process generates a request class with attributes represented in the following comment box. This request class is further instantiated by executing the process “Execute the CreateObjectAction on Request class.” The object created by the process is RQ_CALC. Once the object is created, the next step is to send RQ_CALC to the tank for execution of the request. Now, the process “Execute
SendObjectAction on RQ_CALC” is performed. This process leads to the AND16 gate with two concurrent outputs represented by paths p9 and p10.

Path p9: The request object RQ_CALC is sent to the target object i.e. tank. Along the path a condition check “Is isSync = true?” evaluates to true (Yes). Further, the path leads to a multiple input OR5 gate whose output leads to the process “Execute the AcceptCallAction”. Note that AcceptCallAction is executed only if the target object i.e. tank is ready to accept the call operation i.e. calcLevel. In case 1.1, this process is not executed to completion, since the tank object is not found. Thus the missing tank class/object fault is triggered. As a result, the execution of TH1 ends abruptly.

Path p10: It leads to the process “Generate a SendOperationEvent” which generates a send event for calcLevel. The next process is to “Create an OccurrenceSpecification” which is used by the interaction model ESD to start its execution. The path p10 is further divided into three paths (p11, p12, and p13) at AND10. Path p11 leads to the process “Update the event pool of the Host object”, which does not complete since the Tank object is missing. Path p12 leads to connector C1, which creates a new execution thread TH2 and acts as the communication link between the action model ESD and the interaction model ESD. Along path p12, the time of occurrence of send event is recorded.

The execution of the Interaction Model ESD (Figure 29) i.e. TH2 begins with the initiating event “Read OccurrenceSpecification.” The initiating event is triggered by one of the inputs of the OR1 gate i.e. C1 created in TH1 along path p12. The “Read OccurrenceSpecification” returns an event SendOperation, given in the comment box.
The next element is a condition check “Event found?” The condition evaluates to true (Yes) and the path leads to the process “Read Event.”

The process “Read Event” is followed by the condition check “Type of Event E(i).” The condition selects the “SendOperationEvent” path since the event is a send operation $calcLevel$. The path further leads to OR2. The output of OR2 leads to the process “Read MessageOccurrenceSpecification.” The output of this process is given in the comment box.

The next process is “Read Message”, which further leads to OR3 gate. The OR3 leads to a condition check “ReceiveEvent occurred for Receiver (j)?” The receiver object in this case is $tank$. Recall, TH1 execution thread of the Action Model was stuck at the AcceptCallAction process. As such the receiveEvent for $calcLevel$ was never generated. Therefore, the condition check evaluates to No. This leads to the process “Read event pool” followed by a condition check “ReceiveEvent found?” The condition evaluates to No, since receive event for $calcLevel$ was not found. The receiveEvent is never generated during execution of the Action Model ESD (TH1) and hence the condition check “ReceiveEvent found?” never evaluates to Yes. As a result, TH2 execution thread is stuck in an infinite loop.

To evaluate the software design, the FPSA execution model is simulated for multiple time steps. A “simulation step” is defined as “one full execution of all the processes contained in the execution model” [Mutha et al. 2012], i.e., execution until the final-node of the last activity in the activity diagram. A “simulation run” is defined as “a repetition of
a simulation step until the point of interest (e.g., a pre-specified mission time defined in multiples of the simulation step) or point of failure” [Mutha et al. 2012].

These concepts are defined in Figure 30. As displayed in the figure, the clock is reset after each simulation step $t_1$, $t_2$, …, $t_n$, and a simulation run lasts for time $t$ where $t = t_1 + t_2 + … + t_n$.

**Figure 30 Definitions of “simulation step” and “simulation run”**

Case 1.2 Missing message "getdata"

In this case, we inject a missing message “getdata” fault in the class tank and the propagation of this fault and its impact is analyzed. Intuitively, this fault seems less severe than missing tank class/object fault. However, the results obtained from FPSA analysis are counterintuitive. The results of high-level propagation are tabulated in Table 4. The low-level execution is not explained in detail in this case, but the observations and insights obtained are discussed later.

Case 1.3 Missing attribute “Level”

In this case, we inject a missing attribute “level” fault in class tank and the propagation of this fault and its impact is analyzed. Intuitively, this fault seems less severe than missing tank class/object fault or missing message “getdata” fault. However, the results obtained from FPSA analysis are counterintuitive. The results of high-level propagation
are tabulated in Table 4. The low-level execution is not explained in detail in this case, but the observations and insights obtained are discussed later.

<table>
<thead>
<tr>
<th>Simulation step</th>
<th>Configuration Manager</th>
<th>Sensor</th>
<th>ValveController</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Configure system</td>
<td>Read pressure</td>
<td>Calculate level</td>
</tr>
<tr>
<td>1</td>
<td>O</td>
<td>O</td>
<td>L</td>
</tr>
<tr>
<td>2</td>
<td>O</td>
<td>O</td>
<td>L</td>
</tr>
<tr>
<td>3</td>
<td>O</td>
<td>O</td>
<td>L</td>
</tr>
<tr>
<td>...</td>
<td>O</td>
<td>O</td>
<td>L</td>
</tr>
<tr>
<td>10</td>
<td>O</td>
<td>O</td>
<td>L</td>
</tr>
</tbody>
</table>

**Case 1.2 Missing message "getdata" from Tank Class**

<table>
<thead>
<tr>
<th>Simulation step</th>
<th>Configuration Manager</th>
<th>Sensor</th>
<th>ValveController</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>getdata</td>
<td>O</td>
</tr>
<tr>
<td>1</td>
<td>O</td>
<td>O</td>
<td>O</td>
</tr>
<tr>
<td>2</td>
<td>O</td>
<td>O</td>
<td>O</td>
</tr>
<tr>
<td>3</td>
<td>O</td>
<td>O</td>
<td>O</td>
</tr>
<tr>
<td>...</td>
<td>O</td>
<td>O</td>
<td>O</td>
</tr>
<tr>
<td>10</td>
<td>O</td>
<td>O</td>
<td>O</td>
</tr>
</tbody>
</table>

**Case 1.3 Missing attribute Level**

<table>
<thead>
<tr>
<th>Simulation step</th>
<th>Configuration Manager</th>
<th>Sensor</th>
<th>ValveController</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>getdata</td>
<td>O</td>
</tr>
<tr>
<td>1</td>
<td>O</td>
<td>O</td>
<td>O</td>
</tr>
<tr>
<td>2</td>
<td>O</td>
<td>O</td>
<td>O</td>
</tr>
<tr>
<td>3</td>
<td>O</td>
<td>O</td>
<td>O</td>
</tr>
<tr>
<td>...</td>
<td>O</td>
<td>O</td>
<td>O</td>
</tr>
<tr>
<td>10</td>
<td>O</td>
<td>O</td>
<td>O</td>
</tr>
</tbody>
</table>

Table 4 High-Level FPSA Simulation Results For The Three Fault Cases

Figure 31 Event pools of Pressure sensor, Position sensor, Tank, Controller and Inlet valve- Nominal case
The event pools of Pressure sensor, Position sensor, Tank, Controller object are same as Nominal case in Fig. 29

Figure 34 Event pool of Pressure sensor, Position sensor, Tank, Controller and Inlet valve - Case 1.3 (Simulation step 1)
Observations/Insights:

The results of FPSA high-level simulation (Table 4) indicate interesting patterns of function failure. Case 1.1 shows a pattern with two function losses i.e. Calculate Level and Valve control logic. While case 1.2 and 1.3 show a pattern with one function loss i.e. Valve control logic. These failure patterns can be translated to the system level impact indicators. The system fails in all three cases, even though two different function failure patterns are observed. A simple boolean logic reduction indicates that the loss of Valve control logic is the root cause of the system failure. Thus, we can establish a simple cause-effect rule:

**RULE1: Valve control logic failure causes system failure**

Although an expert can easily identify that valve control logic is a critical function, the question is which faults can potentially propagate and trigger the Valve control logic failure? The three fault cases presented in Table 4 are of interest; however, many fault combinations are possible. The number of fault combinations is given as below:

\[
\text{#fault combinations} = \sum_{i=1}^{\#\text{design elements}} \binom{\#\text{design elements}}{i} \times (\#\text{fault types})^i \quad \ldots (1)
\]

For instance, consider the three most commonly occurring faults types: “Missing (M)”, “Extra (E)” or “Incorrect (I)” design elements for design 1. Based on above equation, we have the number of uni-variate fault types (i.e. for \( i = 1 \)) as:
Additionally, multi-variate fault combinations need to be analyzed. There are three possible combinations of two simultaneous fault types and one possible combination of three simultaneous fault types. Note that the same design element cannot have two fault types, hence the number of fault combination for two simultaneous fault types becomes (i.e. for i = 2):

\[(\#design\ elements)(\#design\ elements - 1) \times (3)^2\]

And the number of fault combination for three simultaneous fault types becomes (i.e. for i = 3):

\[(\#design\ elements)(\#design\ elements - 1)(\#design\ elements - 2) \times (3)^3\]

Clearly, identifying the fault combinations of interest, for example faults leading to Valve control logic failure, is a daunting task. An automated method is essential for such a fault exploration effort. To automate the process and make an accurate determination of a fault’s impact, a low-level execution algorithm is proposed. The low-level execution allows automatic execution of the software models at the implementation level and thus accounts for possible implementation level fault types and their fault combinations. The low-level execution was illustrated in case 1.1. In addition, ways to reduce the fault space can be designed based on dependency rules and probability considerations as discussed in the conclusions.

The low-level execution captures the implementation level details (while not being at the code level), which assists in developing new fault patterns. These details include the
events that are triggered, their timing, and the execution time of the high-level behavior. The low-level execution captures propagation paths that may be different than expected. For instance, in case 1.1 the high-level propagation path included . However, the low-level execution indicates that this part of the path will never be executed. The execution stops at the activity calculate level (see figure 29). Recall, thread TH1 and TH2 execution was incomplete.

During the low-level execution of case 1.1, it was realized that the receive event of message calcLevel never occurred. Since, the event pool for Tank object was missing (Fig. 30), the event pool of Controller was also empty. The Tank and the Controller event pools exhibit a different behavior than in the nominal condition (Figure 31). In case 1.2, during the execution of activity Valve control logic, the send event of message getdata to the tank never occurs, since the missing getdata fault was injected. Consequently, the corresponding receive event never occurs. The execution thread TH1 of activity Valve control logic ends abruptly. Thus, the Tank’s event pool (Fig. 31) contains one less message than in the nominal case. Moreover, since the messages are synchronous, no return action is performed and this leads to the empty event pool of the Controller object. Thus a deviation from nominal behavior is observed. In case 1.3, for simulation step 1 all the execution threads executed to completion (Fig. 32) and the Inlet valve’s event pool contains one message, which is different from the nominal case. However, for simulation steps 2-10, all event pool behaviors are the same as in the nominal case (Figure 31). The capability of observing such low-level details at an early design phase can provide interesting insights.
The insights gained from the low-level execution can be used to develop unique failure patterns. It was observed for the three different fault cases that the number of objects created in each case is different, and the event pool of the objects have different number of messages. Three sample patterns, based on a simple counting of objects and messages, can be developed as shown in Table 5. Similarly, complex patterns could be developed by adding information regarding the timing of events.

<table>
<thead>
<tr>
<th>Faults</th>
<th>Patterns</th>
</tr>
</thead>
<tbody>
<tr>
<td>Case 1.1: Missing object</td>
<td>1. # objects created &lt; # objects expected</td>
</tr>
<tr>
<td></td>
<td>2. # messages in the object’s event pool &lt; # messages expected</td>
</tr>
<tr>
<td>Case 1.2: Missing message getdata</td>
<td>1. # objects created = # objects expected</td>
</tr>
<tr>
<td></td>
<td>2. # messages in the object’s event pool &lt; # messages expected</td>
</tr>
<tr>
<td>Case 1.3: Missing attribute level</td>
<td>1. # objects created = # objects expected</td>
</tr>
<tr>
<td></td>
<td>2. # messages in the object’s event pool = # messages expected</td>
</tr>
</tbody>
</table>

Table 5 Faults and sample patterns

Thus for a given design (Design 1 in this case), a critical function (Valve control logic) is identified and the different ways in which the function may fail are evaluated. Now, the designer can develop and implement safeguards against the failure of the critical function. In case of multiple design choices a comparative analysis of advantages and disadvantages of each design is necessary and is illustrated in the following case 2.1 and case 2.2.
Case 2.1: Design 1 with missing activity “Read Position”

The position is read from the hardware “Position sensor”. Hardware systems instrumented with sensors are commonly observed. However, the FPSA based analysis shows that it may not be reasonable to trust the sensors in some conditions. The results of the high-level execution are presented in Table 6 followed by the discussion on observations and insights from the complete FPSA analysis.

Case 2.2: Design 2 with missing activity “Read Position”

The position is read from the software database itself. Commands issued to the inlet and the outlet valves in preceding software runs are stored in a database to access the current position of the valves. In subsequent runs, the “read position” activity reads the data stored in the database. However, the FPSA based analysis shows that even such a design may not be appropriate in some conditions. The results of the high-level execution are presented in Table 6 followed by a discussion of observations and insights from the complete FPSA analysis.

<table>
<thead>
<tr>
<th>Case 2- Missing Activity &quot;Read position&quot;</th>
<th>(Design 1/Design2: Nominal condition)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Simulation step</td>
<td>Configuration Manager</td>
</tr>
<tr>
<td>1</td>
<td>O</td>
</tr>
<tr>
<td>2</td>
<td>O</td>
</tr>
<tr>
<td>3</td>
<td>O</td>
</tr>
<tr>
<td>…</td>
<td>O</td>
</tr>
<tr>
<td>10</td>
<td>O</td>
</tr>
</tbody>
</table>

Table 6 Result Of Missing Activity “Read Position” Fault In Two Different Designs
Table 6 continued.

| Case 2.1- Missing Activity "Read position" (Design 1: Level < L) |
|-------------|--------|--------|----------|---|
|            | O      | O      | O        | L  |
| 1           | O      | O      | O        | NA |
| 2           | O      | O      | O        | NA |
| 3           | O      | O      | O        | NA |
| …           | O      | O      | O        | NA |
| 10          | O      | O      | O        | NA |

| Case 2.2- Missing Activity "Read position" (Design 2: Level < L) |
|-------------|--------|--------|----------|---|
|            | O      | O      | O        | O  |
| 1           | O      | O      | O        | NA |
| 2           | O      | O      | O        | NA |
| 3           | O      | O      | O        | NA |
| …           | O      | O      | O        | NA |
| 10          | O      | O      | O        | NA |

Observations/Insights:

The results in Table 6 provide useful insights into advantages and disadvantages of different designs. These insights support a designer critical design decision, and design trade-offs.

The execution with nominal condition (i.e., the water level is within desired range, and inlet and outlet valves are open) exhibits no signs of failure. For both designs all the functions are operating including the critical function Valve Control Logic. However, the execution paths are different in design 1 and design 2. Design 1 exhibits an execution path containing (extracted from Figure 18) in all simulation runs. This execution path leads to opening of the inlet valve and outlet valve, and no failures are triggered. Design 2 exhibits an execution path D1-D2-D3-exit in all the simulation runs. Note, although the execution paths are different, the system-level outcome is the same i.e. both inlet and outlet valve stay open. As a result the holdup tank
system operates as desired. At the low-level execution the number of objects created are the same; however the number of messages passed is different for both the designs.

For a non-nominal case i.e. Level < L_L, failure is triggered in design 1 and not in design 2 (Table 6). The failure proves that missing activity Read Position is necessary for design 1, while for design 2 this activity is redundant. Another notable difference in design 1 and design 2 is the different execution paths. Design 1 exhibits an execution path containing (Figure 18) in all the simulation runs. This execution path leads to opening of the inlet valve and not the closing of outlet valve. Thus, “Tank dry” failure condition is triggered. Design 2 on the other hand follows a path D1-D5-D6-D7-Close outlet in the first simulation run, and during the second simulation run follows D1-D5-D6-D7-Exit. Thus, the outlet valve remains closed and no failure condition is triggered. The outlet valve opens in the subsequent runs when the water level in the tank is back in the desired range.

The results of the two designs for nominal condition i.e. the water Level is in the desired range, and the inlet and outlet valves open, are the same. Thus intuitively either design 1 or design 2 can be implemented. The choice will depend on the designer’s perspective, which may be either cost-oriented or sensors-oriented. However, to choose a better design, it is not enough to verify that the designs work under nominal conditions, but also under off-nominal conditions such as more extreme cases outside the desired range. Additionally, a comparison of trade-offs between the two designs must be made to support the decision. A sample comparison between design 1 and design 2 based on the results obtained from FPSA analysis is given in Table 7.
There is no “Store Pos” activity in *Valve Control Logic*. Thus, lesser message passing and memory usage will be observed.

Bad sensor will trigger a failure condition.

Since position is directly read from sensors, memory corruption or loss is not an issue.

<table>
<thead>
<tr>
<th>Design 1</th>
<th>Design 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>There is no “Store Pos” activity in <em>Valve Control Logic</em>. Thus, lesser message passing and memory usage will be observed.</td>
<td>There are several “Store Pos” activities in <em>Valve Control Logic</em>. Thus, a higher message passing and memory usage will be observed.</td>
</tr>
<tr>
<td>Bad sensor will trigger a failure condition.</td>
<td>The design is not sensor dependent.</td>
</tr>
<tr>
<td>Since position is directly read from sensors, memory corruption or loss is not an issue.</td>
<td>Since position is read from the memory, any memory corruption or loss will trigger a failure condition</td>
</tr>
</tbody>
</table>

Table 7 Comparison between Design 1 and Design 2

The comparison shows that one design may be safer and more reliable than other depending on the reliability of the hardware and the environment in which the system will operate. If the sensor has high reliability compared to the reliability of the memory, then design 1 will be a safer choice. However, if the sensor cannot operate in a harsh environment like high temperatures, or in the presence of radioactive materials, then design 2 will be a better choice. Another way to ensure system safety and reliability is through implementing design diversity. Design diversity can be achieved by adding redundant hardware elements and also software code to fulfill the same functional requirement. Suitable components for implementing a particular redundancy can be analyzed using the FPSA method. For example, redundant sensors could be added to ensure high sensor reliability or a backup memory, which records critical variables, could be employed, or a combination of both multiple sensors and backup memory with a voting logic can be employed. Such hardware-software diversity addresses the design diversity and consequently improves the system reliability.

*Case 3: Two simultaneously occurring faults Missing activity and Extra message*
Missing activity element “Read Position” in activity diagram and an Extra message element “getdata” in the sequence diagram, are considered here.

In case 3, missing activity fault is cancelled by the extra message fault. Even though the Read position activity was not defined in the high-level design, the low-level design includes the messages “getdata” that captures the sensor data as such cancels the effect of the missing activity fault. Many such harmless combinations of faults can be discovered, thus reducing the fault combinations of interest.
Chapter 5. Integrated System Failure Analysis

The Integrated System Failure Analysis (ISFA) technique analyzes the fault propagation paths and the functional impact of faults in software-driven electro-mechanical systems. It is developed from two principally similar techniques: Fault Propagation and Simulation Approach (FPSA), applicable to UML based software system; and Function Failure Identification and Propagation (FFIP), applicable to electro-mechanical systems. These techniques address the design aspects of different technical domains; hence syntactic as well as semantic considerations are required while integrating the models.

In this research, the integration of two different domains is based on principles of interface matching, input/output data matching, synchronization of events, and communication (in the form of correct messages and their timing). Except interface matching, the latter are related to the behavioral aspects modeled using the behavioral diagrams. The functional behavior model of FFIP captures the functional flow of the hardware system while the activity, use case, and sequence diagrams of UML capture the functional software aspect. Of these, only the activity diagram is capable of capturing the flow of the software functions, handling externally triggered activities, and including elements for sending signals to external entities. Therefore, the functional model of the
FFIP approach and the activity diagram of UML are integrated to study the integrated system-level function.

**Functional Failure Identification and Propagation**

The FFIP technique is an approach for evaluating and assessing the risk of functional failures during the conceptual design phase. The task of the FFIP technique is to estimate potential faults and their propagation paths under critical-event scenarios. FFIP was developed with a focus on modularity and with the intent of capturing the effect of complex system interactions [Jensen et al., 2008; Kurtoglu & Tumer, 2008; Jensen et al., 2009; Kurtoglu et al., 2010]. FFIP identifies the propagation and functional effect of component failures by identifying the function–component mappings from a database of generic components during the system-simulation process. The database includes qualitative, state-machine behavioral models for each generic component. These behavioral models capture both nominal and faulty behavior. During the system simulation process, different nominal and faulty behaviors are triggered. FFIP is formalized using the meta-modeling constructs of UML class diagram. Such formalization is necessary to seamlessly integrate with other domains.

**Formalization of FFIP**

For the formalization of FFIP, the modeling elements will first be expressed using a formal language such as MOF. MOF is widely used and provides the necessary constructs for expressing the conceptual modeling element. Figure 35 shows the FFIP domain modeling elements and their relationships. The FFIP modeling approach
represents a system in three different views: functional, behavioral, and in terms of components. Together, these views form the complete model of an electromechanical system. The FFIP modeling elements are divided into four packages: FunctionModel, ConfigurationFlowGraph, Flow, and BehaviorModel. The FunctionModel and ConfigurationFlowGraph import the same Flow package. Table 14 (in Appendix) provides an overview of the different models and the modeling elements.

Figure 35 Functional Failure Identification And Propagation Metamodel.
FunctionModel is composed of functions and sub-functions identified as the element HW_Function and different types of Flows between functions such as Signal, Material, and Energy. The association between hardware function and flows indicate that the HW_Function acts on the incoming Flow and transforms it to outgoing Flow. The FunctionLibrary is a library of predefined functions that can be updated with newly discovered functions. In FFIP, “function” is viewed as the actions that the design is supposed to perform not the subjective purpose of the design [Deng, 2002]. The hardware functions are selected from this function library. Similarly, the FlowLibrary is a library of predefined flow that can be updated with the newly discovered flows. Using the predefined functions and flows, and connecting them in a particular sequence, a function model is generated to achieve the actual or desired functionality of the system. Taxonomy, such as functional basis [Hirtz et al., 2002], can be used to standardize the naming convention of function and flow.

The component model, the Configuration Flow Graph (CFG), is composed of hardware components and subcomponents (collectively termed HW_Component); different types of flows such as Signal, Material, and Energy; and the variables depicted as Variable handled by the component. The CFG follows the functional topology. The relationship between hardware function and hardware component is such that one HW_Component can implement multiple HW_Function. The mapping between HW_Component and HW_Function is critical for the FFIP framework. The HW_Function acts on the input variables and transforms them into output variables. The overall component structure of the system is governed by the system functional model. CFGs and functional models are
similar to directed flow graphs where a “component” of CFG and a “function” of the functional model acts as a “node” and the “flow” acts as the “arc.” The output of one node is the input of another node. Because functions are mapped to components, the diagrams must maintain flow consistency between the functional and component views of the system. The flow in and out of a function or a group of functions is the same as the flow for the component(s) implementing the function(s). This constraint is specified by a set of “well-formedness” rules defined in the formal object constraint language. An example of this is Constraint C1 in table 14 (in appendix).

The system behavioral model follows a component-oriented approach. Qualitative behavioral models are defined for each component. The component behavior is depicted as BehavioralRules that include both nominal and faulty behaviors derived from the underlying first principles and the relationship between the input and output variables. The BehavioralRules are based on representing the physics of the component interactions at a conceptual stage. This is similar to the Qualitative Physics [DeKleer & Brown, 1984] behavioral descriptions except that state machines are used to represent discrete nominal and faulty behaviors rather than a continuous set of equations. For example, in qualitative physics, the spring equation \( F=k*x \) describes a proportional relationship between the variables \( F \) and \( x \). Qualitative reasoning indicates that the change in \( F \) is of the same sign and proportional in magnitude to the change in \( x \). In this way, qualitative physics uses "landmark" values for variables instead of continuous values. For example, landmark values might be the maximum and minimum \( F \) resulting from maximum and minimum positions \( x \). In this way qualitative physics can be applied when the precise variable range
is not known. While our approach to modeling behavior builds on this, in order to accomplish more precise reasoning about functional states, we have extended this to a state-based, qualitative interval model.

Our models describe discrete states of behavior using qualitative descriptions of the transformation of flows, where the flows variable is discretized into intervals. *BehavioralRules* are state machines composed of multiple *Nominal* states and multiple *Faulty* state definitions. A *Nominal* state can transition to another *Nominal* or *Faulty* state; similarly, a *Faulty* state can transition between other *Faulty* or *Nominal* states. The transitions are triggered by *events* that are environmental factors or control commands. The discrete behavioral rule approach would describe the above spring model as a few discrete states with their own behaviors such as “at rest”, “compressing and expanding.” Additionally we can include some failure states representing broken or misaligned springs. In all these states the output force is related to the input position. So that “Low” magnitude of the position flow results in “Low” value for the force flow under nominal behavior states. The same holds true for other discrete levels of the input flows.

Another important element of the FFIP framework is the Functional Failure Logic (FFL). It relates the component behavior to the operating state of system functions. The FFL evaluates the input and output flow levels as defined in the component’s behavioral model and relates those to the status of intended functions. This operational state is represented as the *HW_Function*’s attribute *status* and is classified as Lost, Operating, or Degraded. The *HW_Function.status* is identified as Lost when the intended function of that component is not achieved. The *HW_Function.status* is said to be Operating when
the intended function is achieved. Finally, the \textit{HW\_Function.status} is said to be 
\textit{Degraded} when the intended function is partially achieved but not as intended. Figure 36 is a representation of a valve component, its function, its behavioral rule, and FFL.

![Diagram of valve components](image)

\begin{align*}
\text{Nominal ON: } & \text{If } Q_{\text{out}} = Q_{\text{in}} \text{ AND } P_{\text{out}} = P_{\text{in}} \\
\text{Nominal OFF: } & \text{If } Q_{\text{out}} = \text{zero} \\
\text{Failed OPEN: } & \text{If } Q_{\text{in}} \neq \text{zero AND } Q_{\text{out}} = \text{zero} \\
\text{Failed CLOSED: } & \text{If } Q_{\text{out}} \neq \text{zero} \\
\text{Clogged: } & \text{If } Q_{\text{out}} < Q_{\text{in}} \\
\end{align*}

\begin{align*}
\text{IF} & \text{ mode } = (\text{Nominal ON } \text{||} \text{ Nominal OFF}) \\
\text{Then} & \text{ Guide Liquid = Operating} \\
\text{IF} & \text{ mode } = (\text{Failed OPEN } \text{||} \text{ Failed Closed}) \\
\text{Then} & \text{ Guide Liquid = Lost} \\
\text{IF} & \text{ mode } = \text{Clogged Then Guide Liquid = Degraded} \\
\end{align*}

Figure 36: (a) The valve component, its input–output variables, and the flow; (b) the valve function and flow; (c) valve behavioral rules in terms of input–output relationship; and (d) valve function failure logic

\textbf{Behavioral Simulation}

Behavioral simulation is a discrete-time simulation integrated with the automatic functional reasoning. To simulate a fault, the fault mode transition in a component behavioral state machine is triggered. This new state defines how the component in the failed mode will change the input–output flow relationship. For example, the “clogged” state of a valve component behavioral model changes the output flow of material from
nominal to zero. After a fault mode transition is triggered, the component state machines connected to that component (based on the CFG architecture) are also executed. Concurrently with the behavioral execution, the FFL evaluates the expected flow conversions. For example, the valve component is mapped to the function to regulate fluid flow. The FFL evaluates the input and output flows from the simulation and compares the expected change of implementing that function to the change observed in the simulation. The FFL then identifies the status of that specific function, as well as the status of all other functions in the model.

**Formalization of ISFA**

Hardware is integrated with software via interfaces. An interface is a component that communicates the send/receive information between physical hardware (HW) and software (SW) systems. Various types of input/output interfaces such as PCI buses, USBs, etc., can perform this task. Interfaces can be complicated and may consist of a number of electronic components such as integrated circuits, resistors, memory units, and capacitors. However, for high-level functional evaluation, the low-level component details are abstracted and the interface functions are defined based on the input/output data. The function of an interface is abstracted as a “transaction.” Basically, this is a signal type of data object. The success or failure of an interface is observed by analyzing the properties of the transaction. Important properties of the transaction include source, target, and timing information.

In the following discussion, the stereotype symbol, $<< >>$, refers to a particular instance of a class. Figure 37 shows the metamodel for the ISFA analysis and elements
used for integration. As shown in Figure 37(a), the structural elements (the configuration flow graph components of FFIP and the component diagram of UML) are integrated via *interface*, while the behavioral elements (functions of FFIP’s functional diagram and UML’s activity diagram) are integrated via *transaction*. The associations *interface* and *transaction* are implemented as association classes and are described in Figure 37(b).

Each transaction is associated with an *interface* and with a *TimingConstraint*. The *TimingConstraint* not only imposes timing constraints on hardware–software interactions but also keeps track of time during the behavioral simulation. At the conceptual level, the hardware represents the physical-system components while the hardware components specific to the software (such as buses, storage devices, and input/output devices) are outside the scope of this research. The attributes owned by <<interface>> represent the input/output data between the hardware and software components.

**Interface**

As mentioned earlier, an *interface* is an abstract concept that refers to a common object of interaction between two components. In the software domain, an interface is modeled as an abstract class that contains the method signature and attributes. Its implementation details are specific to the classifier implementing the interface. Similarly, in the hardware domain, an interface can be modeled as an abstract component capable of sending/receiving signals to a particular hardware component. The <<interface>> depicted in Figure 37(b) can be a component’s required or a provided interface (indicated by the attribute *component* of type “string”). For example, a “Sensor” component provides data via <<interface>>Isensor; therefore, <<interface>>Isensor becomes the
provided interface of “Sensor.” An “Alarm” component requires sensor data acquired via
<<interface>>Isensor; therefore, <<interface>>Isensor will be the required interface of
“Alarm.” The input/output data passed between components is captured by
attribute.value. The component that implements the interface acquires data by execution
of the two methods that an <<interface>> owns. These methods are subjected to the
following constraints.

Constraint: C5
Context: <<interface>>
Inv: If <<interface>>.required = true and <<interface>>.attribute.value != null
Execute <<interface>>.getdata()
Execute <<interface>>.setdata()

Constraint: C6
Context: <<interface>>
Inv: If <<interface>>.required = true and <<interface>>.attribute.value = null
Execute <<interface>>.wait()

Because an interface is the communication link between the hardware and software, a
malfunctioning interface can lead to failure of the complete system. To study the effect of
interface faults, we apply the failure reasoning of FFIP to interface modeling. Similar to
FFIP, each interface has a set of input/output-based behavioral rules and FFL. The
behavioral rules consist of nominal and faulty modes of interface; FFL defines the
functional effect in a particular mode of operation. Sample behavioral rules and FFL are
provided in Table 8.
Figure 37: (a) Integration of the functional failure identification and propagation (FFIP) and failure propagation and simulation approach (FPSA) metamodels (b) the relationship between associations “transaction” and “interface.”

FFL-function failure logic.
### Table 8: Sample behavioral rules and functional failure logic of an interface

**MODE** | **Behavioral Rules**<br>**NOM**<br>If \((<<interface>>\.provided\.value = <<interface>>\.required\.value)\) AND \((<<interface>>\.provided\.transaction = <<interface>>\.required\.transaction)\) AND \((<<interface>>\.transaction\.source ≠ empty)\)<br>**Faulty1**<br>If \((<<interface>>\.provided\.value ≠ <<interface>>\.required\.value)\) AND \((<<interface>>\.provided\.transaction = <<interface>>\.required\.transaction)\) AND \((<<interface>>\.transaction\.source ≠ empty)\)<br>**Faulty2**<br>If \((<<interface>>\.provided\.value = <<interface>>\.required\.value)\) AND \((<<interface>>\.provided\.transaction ≠ <<interface>>\.required\.transaction)\) AND \((<<interface>>\.transaction\.source ≠ empty)\)<br>**Faulty3**<br>If \((<<interface>>\.provided\.value ≠ <<interface>>\.required\.value)\) AND \((<<interface>>\.provided\.transaction ≠ <<interface>>\.required\.transaction)\) AND \((<<interface>>\.transaction\.source ≠ empty)\)<br>**Faulty4**<br>If \((<<interface>>\.provided\.value = <<interface>>\.required\.value)\) AND \((<<interface>>\.provided\.transaction = <<interface>>\.required\.transaction)\) AND \((<<interface>>\.transaction\.source = empty)\)<br><br>**Functional failure logic (FFL)**<br>If NOM, then \(<signal>\.transaction\.status = OK\)<br>If Faulty1, then \(<signal>\.transaction\.status = Degraded\)<br>If Faulty2, then \(<signal>\.transaction\.status = Unknown\)<br>If Faulty3, then \(<signal>\.transaction\.status = Lost\)<br>If Faulty4, then \(<signal>\.transaction\.status = Unknown\)<br><br>**Transaction**

A transaction is an instance of a signal and defines the communication details of the hardware–software interaction. Its function is to communicate that the HW function has generated the necessary data and is ready to send it, while the SW function is ready to...
receive the data and vice-versa. Each transaction is associated with an <<interface>>
where the provided interface will send the transaction and the required interface will
receive it. The transaction is also associated with a TimingConstraint. The details of the
transaction are stored in the following six attributes.

1. Source: Name of the function that initiates a transaction. The source can be either a
   
   HW_Function or a software Activity.

2. Target: The name of the function that receives a transaction. It is subjected to
   
   Constraint C7, indicating that the target function domain is different from the source
   of a transaction.

   **Constraint: C7**
   **Context: <<transaction>>**
   Inv: If <<transaction>>.source = Activity implies <<transaction>>.target = 
   HW_Function
   Inv: If <<transaction>>.source = HW_Function implies <<transaction>>.target = 
   Activity

3. isOrdered: Indicates that the transaction follows a particular order. The default value
   is “false.” The order is defined by the attribute “ordering.”

4. Ordering: Defines a sequence of transactions that should occur when the isOrdered is
   set to “true.”

5. Complete: A flag that indicates completion of a <<transaction>>. It takes a Boolean
   value. The default value is “false.”

6. Status: Indicates the status of the transaction. The transaction status is represented
   using a 2×1 vector. The first component indicates the status of the transaction as it
   relates to the physical condition of the interface. The second vector component
   indicates the status of the transaction resulting from the dynamic execution of the
HW–SW interaction. These two components of the component vector are independent. The first component of transaction status may take the values OK, Degraded, Lost, or Unknown. OK indicates the data has correctly transferred between HW and SW. Degraded indicates the data was corrupted while it was transferred between HW and SW. Lost indicates the data transfer did not take place. Unknown indicates transaction status cannot be determined based on the available input and output. The second component of the transaction status vector can take the values Active, Inactive, Complete, Incomplete, Never started, or Error. Active indicates the transaction was created. Inactive indicates the transaction was not created. Complete indicates the attribute transaction.complete is set to “true” (i.e., the transaction is complete). Incomplete indicates the transaction did not execute to completion. Never started indicates that the transaction was not allowed to start. Error indicates the execution of ISFA was faulty. The default status of the vector is (OK, Inactive). The second component is subjected to necessary conditions defined in terms of relevant TimingConstraint.start and TimingConstraint.finish states. Some of the combinations of start and finish states are unachievable.

**TimingConstraint**

In addition to ensuring completion of communication between the objects, timing is another important factor to determine the reliability and safety of a safety-critical system. Traditionally, the timing requirements are implemented by a watchdog timer and represented as TimingConstraint for each transaction. The TimingConstraint also handles the synchronization aspects of the hardware–software integration. Therefore,
TimingConstraint must be specified for each transaction. If <<transaction>> is unable to fulfill the TimingConstraint associated with it, the transaction.complete flag is set to “false.” This would indicate that the communication between the objects did not complete in a timely manner. The transaction.status would then be set to “Incomplete.”

TimingConstraint records logical temporal details of a transaction in the following five attributes.

1. Start: Marks the beginning of a transaction. The attribute has states: “-1” (unable to start), “0” (not started), and “1” (started). The default value is “0.”

2. Finish: Marks the end of a transaction. The attribute has states: “-1” (unable to finish), “0” (not finished), and “1” (finished). The default value is “0.”

3. Ts: Start time of the transaction.

4. Timevalue: The physical time during the execution.

5. Unit: The unit of time measurement of the system analysis; e.g., millisecond, second, hour, etc.

The attributes of each TimingConstraint are subjected to the following constraints (C8):

**Constraint: C8**
**Context:** TimingConstraint
**Inv:** self.start = “1” and self.finish = “1” implies transaction.complete = true

<table>
<thead>
<tr>
<th>Start</th>
<th>Finish</th>
<th>Status</th>
<th>Start</th>
<th>Finish</th>
<th>Status</th>
<th>Start</th>
<th>Finish</th>
<th>Status</th>
</tr>
</thead>
<tbody>
<tr>
<td>-1</td>
<td>-1</td>
<td>Impossible</td>
<td>0</td>
<td>-1</td>
<td>Impossible</td>
<td>1</td>
<td>-1</td>
<td>Incomplete</td>
</tr>
<tr>
<td>-1</td>
<td>0</td>
<td>Never started</td>
<td>0</td>
<td>0</td>
<td>Inactive</td>
<td>1</td>
<td>0</td>
<td>Active</td>
</tr>
<tr>
<td>-1</td>
<td>1</td>
<td>Impossible</td>
<td>0</td>
<td>1</td>
<td>Impossible</td>
<td>1</td>
<td>1</td>
<td>Complete</td>
</tr>
</tbody>
</table>

Table 9: All possible combinations of [Start, Finish] and corresponding interpretation of Status
Table 9 constitutes the second component of <<transaction>>.status that result from the dynamic execution of the IFSA model. Boolean logic of start and finish values constitute the interface’s behavioral rules while the “Status” constitutes Function Failure Logic (FFL). The default values of [start, finish] are [0, 0] and change dynamically during the model execution.

5.5 InstanceSpecification

InstanceSpecification is a class used to model additional constraints imposed by the data-transfer protocols. A computer is a discrete-time system that sends/receives data at specific instants of time to monitor/control a continuous physical process. The data-transfer process has to follow a specific communication protocol depending on the communication model selected; e.g., “polling system.” The communication model imposes additional restrictions such as when and how long a TimingConstraint on a particular transaction is valid and how often data is transferred. The InstanceSpecification class can be modified to adopt these requirements. For example, according to (Dasarathy, 1985), TimingConstraint on events occurring in real-time systems are classified into types: maximum, minimum, and durational. For demonstration purposes, we consider the following attributes of the InstanceSpecification:

1. Min: Defines the minimum time t, which must elapse before a transaction is activated.
2. Max: Defines the maximum time t, allotted for a transaction to complete.
3. Unit: The unit of time measurement of the system analysis; e.g., millisecond, sec., hr., etc.

The transaction status depends on the type of data transfer algorithm selected for communication. Different communication algorithms can be modeled and inserted into the ISFA execution model to determine their system-level functional impact. An example of a simple data-transfer model is expressed in algorithm format as “Algorithm_TStatus” (Figure 38). According to the “Algorithm_status” algorithm, data transfer by the transaction takes place within a time window of \([t_{\text{min}}, t_{\text{max}}]\). If the data is sent too early, i.e., before \(t_{\text{min}}\), then the data is rejected. This is indicated by the attribute \(<\text{signal}>\text{transaction.start} = -1<\text{signal}>\). If the data is sent/received too late, i.e., after \(t_{\text{max}}\), then the data is not transferred. This is indicated by the attribute \(<\text{signal}>\text{transaction.finish} = -1<\text{signal}>\).
Algorithm_TStatus: Algorithm to determine transaction status

Input: <<signal>>transaction
Output: <<signal>>transaction.status

Block 1:
1. Execute <<interface>> behavioral rules
2. Get the "<<interface>>.mode"
3. Execute <<interface>> FFL
4. Get <<signal>>transaction.status(1)
5. Get <<signal>>transaction.status(2)
6. If <<signal>>transaction.status(2) = "Active"
   7. TimingConstraint.start = "1"
   8. TimingConstraint.finish = "0"
7. ElseIf <<signal>>transaction.status(2) = "Inactive"
8. TimingConstraint.start = "0"
9. TimingConstraint.finish = "0"
10. Endif

Block 2:
13. Case 1: TimingConstraint.start = "0"
14. While(TimingConstraint.timevalue ≤ InstanceSpecification.max)
15. If TimingConstraint.start = "1"
17. Go to Case 2
18. Endif
19. Endwhile
20. TimingConstraint.start = "-1"
21. Case 2: TimingConstraint.start = "1"
22. If TimingConstraint.Ts > InstanceSpecification.max
23. TimingConstraint.start = "-1"
24. ElseIf(TimingConstraint.Ts ≥ InstanceSpecification.min and
   TimingConstraint.Ts ≤ InstanceSpecification.max)
25. While(TimingConstraint.timevalue ≤ InstanceSpecification.max)
26. Receive(X)
27. If X = “true"
28. TimingConstraint.finish = "1"
29. Go to Block 3
30. Endif
31. Endwhile
32. TimingConstraint.finish = "-1"
33. ElseIf TimingConstraint.Ts < InstanceSpecification.min
34. TimingConstraint.start = "-1"
35. Endif

Block 4:
36. If TimingConstraint.start = "1" and TimingConstraint.finish = "1"
37. Status = “Complete”
38. ElseIf TimingConstraint.start = "1" and TimingConstraint.finish = "-1"
39. Status = “Incomplete”
40. ElseIf TimingConstraint.start = "1" and TimingConstraint.finish = "0"
41. Status = “Active”
42. ElseIf TimingConstraint.start = "0" and TimingConstraint.finish = "0"
43. Status = “Inactive”
44. ElseIf TimingConstraint.start = "-1" and TimingConstraint.finish = "0"
45. Status = “Never started”
46. Else
47. Status = “Error”
48. Endif
49. <<signal>>transaction.status(2) = Status
50. return(<signal>>.transaction.status)

Figure 38: The Algorithm_TStatus
**ISFA Execution Model**

The execution model of the ISFA technique is expressed using the ESD notation. The hardware and software design execute in parallel. They communicate via transactions of the related interfaces. Both the hardware and software design execution include communication-related processes; for example, creation of a transaction and determination of the transaction status. These communication-specific processes ensure that data is transferred from one system (function) to the target system (function) in a timely manner. The execution model outputs the function statuses of the HW, SW, and Interface. These are input into the system function status identification process explained in Section 4.4.

To evaluate the design, the execution model is simulated over multiple time steps. In the context of the ISFA execution model, a “simulation step” is defined as one full execution of all the processes contained in the execution model, i.e., execution until the last component of the CFG and of the activity diagram. A “simulation run” is defined as a repetition of a simulation step until the point of interest (e.g., a pre-specified mission time defined in multiples of the simulation step) or point of failure.

![Simulation Step and Simulation Run](image)

**Figure 39:** A simulation step and simulation run
Figure 39 shows the concepts of simulation step and simulation run. For each simulation step $t_1$, $t_2$, $t_3$, …, $t_n$, the clock is reset. Therefore, the total time of simulation run can be calculated as $t = t_1 + t_2 + t_3 + \ldots + t_n$.

The ISFA simulation process involves a synchronized execution of the HW design and the SW design. The synchronization occurs via transaction. The HW design execution is driven by the FFIP’s configuration flow graph while SW design execution FPSA’s activity diagram. Each execution algorithms are detailed in Figure 40 and explained below.

The start of the execution leads to the process “Initialize the system”, which sets the initial conditions of the HW, the SW, and the transaction models of the system. After initialization, two concurrent paths p1 and p2 are created by the AND1 gate. Path p1 leads to HW design execution, while path p2 leads to SW design execution. Along each execution path transactions are created and read which ensure synchronization of the HW/SW design execution.
Figure 40: The integrated system failure analysis execution model at time step
**HW design execution**

Path p1 leads to an OR1 gate. The multiple-input-single-output OR1 gate creates a loop, which iterates over the HW components of CFG. OR1 leads to path p3 that points to the process “Identify the HW_Component(i),” where “i” is an index to identify a HW_Component. For each component identified, a multiple-input-single-output OR2 gate creates a loop which iterates over all the functions of the component in consideration. OR2 leads to the process “Read HW_Function (i,j),” where “j” is an index to identify a HW_Function of the i-th component. The outputs of this process i.e., function name, inflow, and outflow information are presented in the comment box. For each function identified, a multiple-input-single-output OR3 gate creates a loop which iterates over all the inflows of the functions. This iterative process over the inflows consists of a condition check “Is inflow (i,j,k) = <<signal>>transaction?” where “k” is an index to identify the inflow of j-th function of i-th component.

a. If the above condition is evaluated to “Y,” it means that an incoming transaction is necessary to execute the HW_Function(i, j). The transaction is created during the SW design execution while it is read during HW design execution by the process “Read <<signal>>transaction”.

The process returns the detail of the corresponding transaction, i.e., target and status. Next, the condition “Is HW_Function(i, j) = transaction.target?” checks if the target of the transaction is the function being executed. If “Y,” the process “Execute Algorithm_TStatus” evaluates the transaction status. The process leads to AND3, where the path branches into two parallel paths: p4 and p5. Path p5 ends with the
process “Update transaction(k).status.” Path **p8** leads to a condition check “Is inflow (i,j,k) = <<signal>>transaction?” If “N,” it leads to a condition check “Is k>#inflow?”

b. If the above condition is evaluated to “N,” the loop continues to execute until the condition “Is k>#inflow(k_max)?” is satisfied.

Once all the inflows of the function are identified, the execution path marked as **p6** leads to the process “Execute HW_Function (i, j).” This process will modify the output variable values and may cause a component mode change. So, the next process “Execute the Behavioral Rules” is executed to identify the mode of the component. Each mode definition comprises of a set of variables assembled in a mathematical equation. These variables are extracted by the subsequent process “Extract variables.” This process provides the name, value, and flow type as seen in the subsequent comment box further leading to **AND4 gate.** **AND4** branches the path **p6** into two parallel execution paths: **p7** and **p8.** Path **p7** leads to the process “Execute FFL,” which determines the function status and the process “Update HW_Function status” terminates this path. Path **p8** leads to **OR4** which iterates over all the output variables of the function being considered and checks if there is any outgoing transaction signal.

**OR4** leads to the condition check “Is outflow (i, j, m) = <<signal>>transaction,” where “m” is the index of outflow variable.

a. If the above condition is evaluated to “Y,” it leads to **AND6** which creates two parallel execution paths: **p11** and **p12.** Path **p11** increments the outflow-counter and leads to **OR4.** Path **p12** leads to the process “Create the <<signal>>transaction” that instantiates the necessary signal, its the attributes i.e., source, target, and status. The
transaction will be read during the SW design execution. The path p12 ends with the process “Update transaction(m).status.”

b. If the above condition is evaluated to “N,” it leads to AND5 which creates two parallel paths: p9 and p10. Path p9 checks the condition “Is m > #outflow(m_max)?” and increments the outflow if the condition evaluates to “N.” Path p10 leads to a condition check “Is j > #functions(j_max)?” which checks if the index of the current function is greater than the total number of functions of the i-th component. If “Y,” then the execution process leads a condition check “Last component?” If “Y,” last component of CFG is reached and the execution path p1 ends. If “N” the path leads to OR1 which iterates over the next component. If the condition “Is j > #functions(j_max)?” evaluates to “N,” then the execution process leads to OR2 and the iteration over next function continues.

**SW design execution**

SW design and its execution are fundamentally different from the HW design within which it operates. In an object-oriented SW design the structural diagrams do not capture the flow of the software execution. The flow of the SW execution is captured in the activity diagram.

The SW design execution begins with the process “Execute the main activity diagram”. The process leads to OR5 which iterates over all the activities of the main activity diagram. The output of OR5 marked as path A1 leads to the process “Read Activity (l)” returns the name of the activity(l) which further leads to process “Identify the component” which returns the name of the component that surrounds Activity (l), the
components inflows \((n_{\text{max}})\), outflows\((p_{\text{max}})\), where \(n_{\text{max}}\) and \(p_{\text{max}}\) are the maximum number of inflow and outflows as shown in the subsequent comment box. This leads to OR6 which iterates over all the inflows of the activity\((l)\). The iteration involves a condition “Is \(\text{inflow}(n) = \text{<<signal>>transaction}?”

a. If the above condition evaluates to “Y,” it leads to the process “Read \text{<<signal>>transaction.”

b. If the above condition evaluates to “N,” it leads to another condition “Is \(n > \#\text{inflow}(n_{\text{max}})?\)” If “Y,” it leads to path A4. If “N,” it leads back to OR6 to evaluate the next inflow.

The process “Read \text{<<signal>>transaction}” returns the details of the corresponding transaction, i.e., target and status, as presented in the subsequent comment box. Next, the condition “Is Activity\((l) = \text{transaction.target}?” checks if the activity being executed is the same as activity identified during the hardware design execution.

a. If the above condition is evaluated to “Y,” the process “Execute Algorithm\_TStatus” evaluates the transaction status. The process leads to AND7, where the path branches into two parallel paths: A2 and A3. Path A3 ends with the process “Update \text{transaction}(n).status.” Path A2 leads back to the condition “Is \(n > \#\text{inflow}(n_{\text{max}})?\)”

b. If the above condition is evaluated to “N,” it leads directly to the condition “Is \(n > \#\text{inflow}(n_{\text{max}})?\)”

Once all the inflows of the function are identified, the execution path marked A4 leads to the process “Execute Activity \((l)\)” which further leads to AND8. This process will modify
the output variable values that may or may not cause a component mode change. The
\textbf{AND8} gate divides the path A4 in two parallel paths: A5 and A6.

Path A5 leads to the process “Execute the Behavioral Rules” determines the component
mode, as presented in the subsequent comment box. Next, the process “Execute FFL” is
executed to determine the status of the Activity (l).

Path A6 leads to the process “Extract outflow variables” which returns the name of the
variables, their values, and their flow type. The process leads to \textbf{OR7} which iterates over
all the outflows. During the iteration, the condition check “Is outflow(p) =
\texttt{<<signal>>transaction}?” is performed.

a. If the condition evaluates to “Y,” it leads to \textbf{AND9}, creating two parallel executing
paths: A7 and A8. Path A7 increments the outflow counter and leads to \textbf{OR7} to
evaluate the next outflow. Path A8 leads to the process “Create the transaction” which
creates an object of the transaction and sets the transaction (p).status = “Active,” as
shown in the subsequent comment box. The path ends with the process “Update
transaction(p).status.”

b. If the condition evaluates to “N,” it leads to another condition “Is p> #outflow(p_{max})?”
If this condition evaluates to “N,” next signal is considered. If this condition evaluates
to “Y,” a check on end of main activity diagram is performed. If the end of main
activity diagram is reached the execution path p2 ends. Else the next activity is read.

\textit{Evaluation of System Function Status}

System functions are identified in the system requirements. These functions are
decomposed into hardware, software, and interface functions. Determination of system
function is dependent on the status of the decomposed functions. However, evaluation of system function status based on the decomposed functions status is not a matter of set theory union. A system failure can be defined in terms of critical physical variables that cross limiting conditions. These conditions—called “system failure criteria”—are deterministic and are predefined by the system analysts/designers. The state of the critical variables continuously changes during the ISFA design execution described in Section 4.3. The evolution of these critical variables is the result of HW Function, Activities, and transactions. At the end of each simulation step, the state of the critical variables must be evaluated to determine if a system failure has occurred. An overview of the complete process of system function status evaluation is summarized in Figure 41.

Figure 41: An evaluation of the system function status.
IFSA-integrated system failure analysis; HW-hardware; SW-software

Case Study

In this section we demonstrate the application of the ISFA method using a “holdup” tank system (Figure 15). The holdup tank in this case study is composed of an inlet valve with a position sensor, pipes, a tank with a pressure sensor, an outlet valve with a position sensor, and a software-based computer controller. The function of the holdup tank system
is to regulate the fluid flow from the tank to the output pipe while maintaining the desired water level in the tank. If the pressure is below a critical value, the output flow must be stopped and input flow must start so that the water level is within the desired range. The input and output valves operate according to the software controlled logic defined in the activity diagram (Figure 18).

The holdup tank system ensures a constant flow (say Q) of water to a nuclear core as the heating element. In this hypothetical example we assume that if the water supply from the holdup tank is lost for more than 5 units of time, the core may uncover leading to an accident. As a safety measure, a backup system pumps water from a limited capacity reservoir when the water level in holdup tank drops below the lower threshold limit. The availability of the backup system is limited for example let us assume that water can be pumped for up to 5 units of time and the reservoir can be re-filled every 20 units of time. Thus the backup system availability is one for only 5 units of time and zero for the remaining 15 units of time.

In this case study, we first describe the models that conform to the ISFA metamodel followed by the demonstration of the ISFA simulation process (Figure 40). The demonstration includes analysis of two different faults: 1) a tank leak that leads to fatigue failure of the outlet valve; 2) an incorrect software modification in the presence of a tank leak. How these faults propagate within the ISFA models and lead to the system failure will be discussed in detail.

System Models
Figure 42: ISFA functional model of the holdup tank system

Figure 43: ISFA component diagram
The system component model is shown in Figure 43. The component model is composed of:

1. **Physical Component Model**: It is described by the FFIP::ConfigurationFlowGraph. The components include the inlet valve with a position sensor, pipes, a tank with a pressure sensor, and an outlet valve with a position sensor. The flow between the components is *liquid* and that between the interfaces is *signal*.

2. **Interface Model**: Based on the ISFA metamodel, the <<interface>> are annotated as I1, I2, I3, I4, and I5 on the model. For example, <<interface>>I1 and <<interface>>I2 are *required* interfaces of software component **Sensor**. While <<interface>>I1 and <<interface>>I2 are *provided* interfaces of hardware components pressure, and position sensors respectively.
3. **Software Component Model**: Described by the combined UML::Deployment diagram and UML::Component diagram. This model conforms to the additional constraints imposed by the FPSA metamodel. The software components are ConfigurationManager, Sensor, and Valve Controller.

Figures 11, 13, 14, and 15 illustrate the system functional models. The three parts are:

1. **Physical Function Model**: Described in Figure 42 contains the FFIP::Functional model that conforms to the $HW_{Function}$–$HW_{Component}$ mapping relationship. The relationship is explicitly tabulated in the Table 10 for completeness.

<table>
<thead>
<tr>
<th>$HW_{Component}$</th>
<th>$HW_{Function}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>Holdup tank</td>
<td>Store fluid, supply fluid</td>
</tr>
<tr>
<td>Pressure sensor</td>
<td>Measure pressure</td>
</tr>
<tr>
<td>Position sensor</td>
<td>Measure position</td>
</tr>
<tr>
<td>Pipe</td>
<td>Transfer fluid</td>
</tr>
<tr>
<td>Inlet and outlet valve</td>
<td>Regulate fluid</td>
</tr>
</tbody>
</table>

Table 10: Mapping of hardware component to hardware function

2. **Transaction Model**: Based on the ISFA metamodel, the $<<signal>>$ transactions are annotated as T1, T2, T3, and T4. Figure 44 presents the details of a sample transaction T1. The attributes of the TimingConstraint associated with each transaction are set to default values, i.e., start = 0; finish = 0; timevalue = 0; units = s. These attributes change during the simulation process, as discussed in Section 4.2. Table 11 summarizes the transaction–interface association.
<table>
<thead>
<tr>
<th>Transaction</th>
<th>Provided Interface</th>
<th>Required Interface</th>
</tr>
</thead>
<tbody>
<tr>
<td>T1</td>
<td>I2. Pressure sensor</td>
<td>I2. Sensor</td>
</tr>
<tr>
<td>T2</td>
<td>I4. ValveController</td>
<td>I4. Outlet valve</td>
</tr>
<tr>
<td>T3</td>
<td>I4. ValveController</td>
<td>I4. Outlet valve</td>
</tr>
<tr>
<td>T4</td>
<td>I5. ValveController</td>
<td>I5. Inlet valve</td>
</tr>
<tr>
<td>T5</td>
<td>I5. ValveController</td>
<td>I5. Inlet valve</td>
</tr>
<tr>
<td>T6</td>
<td>I1. Position sensor</td>
<td>I1. Sensor</td>
</tr>
<tr>
<td>T7</td>
<td>I3. Position sensor</td>
<td>I3. Sensor</td>
</tr>
</tbody>
</table>

Table 11: Mapping of transaction to provided and required interfaces

3. **Software Function Model**: Described in Figures 17 and 18 are the activity diagrams conforming to the additional constraints imposed by the FPSA metamodel. Figure 17 clearly presents the component–activity mapping relationship.

The behavioral rules and Function Failure Logic (FFL) is presented in table 16. The acronyms in the table are: O = Operating, L = Lost, D = Degraded, U = Unknown, C = Complete, and NA = Not Applicable.

<table>
<thead>
<tr>
<th>Variable</th>
<th>Values</th>
</tr>
</thead>
<tbody>
<tr>
<td>$P_{in}$, $P_{out}$</td>
<td>$P_{val} = { P \mid P_{L_{Th}} &lt; P &lt; P_{U_{Th}}}$, $P_{L_{Th}}$ and $P_{U_{Th}}$ are defined in the design specification</td>
</tr>
<tr>
<td>Level</td>
<td>$L_{val} = { L \mid L_{L} &lt; L &lt; L_{U}}$, $L_{L}$ and $L_{U}$ are defined in the design specification</td>
</tr>
<tr>
<td>ControlCommand</td>
<td>${ \text{Open, Close, Null} }$</td>
</tr>
<tr>
<td>Pos</td>
<td>${ 1, 0 } \approx{ \text{Open, Close} }$</td>
</tr>
</tbody>
</table>

Table 12: Variable and design limitations associated with software component “valve controller”
**Case 1:** Illustrates a hypothetical scenario of how a holdup-tank-leak fault evolves, translates into valve failure and eventually leads to system failure. We assume that all the interfaces are in healthy condition, all transactions have \([\text{min, max}]\) time limit of \([0, 1^3]\), and the transactions are activated and completed within the time limit.

The simulation (Figure 40) begins and the initial conditions set are: no faults are injected, all hardware and software component exhibit nominal behavior, and all transactions are inactive i.e., \([\text{start, finish}] = [0, 0]\) (Table 9). Other initial conditions include *holdup tank* is half full (i.e., \(P_{L\text{th}} < P < P_{U\text{th}}\)), and the *inlet* and the *outlet valves* are Nominal ON (i.e., open). With these initial conditions the simulation is performed and results are recorded in table 13 under case 1. Some of the important steps and results of the simulation are discussed below.

Path p1 (in Figure 40) leads to execution of CFG (Figure 43) and path p2 (in Figure 40) leads to execution of the Activity diagram (Figure 17). The execution along path p1 and p2 is explained next.

**Path p1:** The first component of CFG i.e., *pipe1* is identified (Figure 40 along path p3). It has one function i.e., *transfer fluid* (Table 10) and has one inflow \((Q_{in}^1)\), one outflow \((Q_{out}^1)\) and none of them is a transaction (Table 16). This leads to execution of the *transfer fluid* function (Figure 40 along path p6). Since no faults are injected the execution of behavioral rules concludes the *pipe1* exhibits nominal behavior (Table 16). Along path p7 the execution of FFL indicates the *transfer fluid* function is *operating* (Table 16). Path p8 leads to paths p9 and p10 since there are no outgoing transactions.
Along path 10 since the pipe1 component has only one function the path leads to identification of the next component i.e., inlet valve.

Inlet valve has one function i.e., regulate fluid (Table 10), two inputs i.e., Q\textsuperscript{iv}\textsubscript{in}, and <<signal>>T4 or <<signal>> T5 (Table 16 and Figure 42), and one output i.e., Q\textsuperscript{iv}\textsubscript{out}. The transactions are not created by the software thus they have default values i.e., [start, finish] = [0, 0] which indicates the transactions’ status is inactive (Table 9). Next, we execute the Regulate fluid function (along path p6). Since no faults are injected the execution of behavioral rules concludes that the Inlet valve exhibits nominal behavior (Table 16). Further along path p7 execution of FFL indicates the Regulate fluid function status as operating (Table 16). Path p8 leads to paths p9 and p10 since there are no outgoing transactions. Along path 10 since the Inlet valve has only one function we move on to identify next component in CFG. After Inlet valve there are two components position sensor1 and pipe2. We select position sensor1 as next component and then pipe2, since parallel execution of components in CFG is not currently possible.

Position sensor1 has one function i.e., measure position (Table 10) and has one inflow i.e., Pos, two outflow i.e., Pos and <<signal>>T6 (Table 16). This leads to execution of the measure position function (Figure 40 along path p6). Since no faults are injected the execution of behavioral rules concludes the position sensor1 exhibits nominal behavior (Table 16). Along path p7 the execution of FFL indicates the measure position function is operating (Table 16). Path p8 leads to path p11 and path 12 since there is one outgoing transactions. Along path p12 a transaction <<signal>>T6 is created and its [start, finish] value changes to [1, 0] which indicates its status is Active (Table 9). Path p8 leads to path
p9 and path p10. Along path p10 since the position sensor1 component has only one function the path leads to identification of the next component i.e., pipe2, as mentioned earlier. Execution of pipe2 is identical to the execution of the pipe1 discussed before. Following pipe2 the next component is holdup tank.

Holdup tank has two functions i.e., store fluid and supply fluid (Table10) and has one inflow (Qm), one outflow (Qout) (Table 16). For the first function store fluid there is no incoming transaction, thus leading to the execution of the store fluid function (along path p6). Since no faults are injected the execution of behavioral rules concludes the holdup tank exhibits nominal behavior (Table 16). Along path p7 the execution of FFL indicates the store fluid function is operating (Table 16). Path p8 leads to paths p9 and p10 since there are no outgoing transactions. Along path 10 since the holdup tank component has another function the path leads to identification of the next function i.e., supply fluid.

There is no incoming transaction, thus leading to the execution of the supply fluid function (Figure 40 along path p6). Since no faults are injected the execution of behavioral rules concludes the holdup tank exhibits nominal behavior (Table 16). Along path p7 the execution of FFL indicates the supply fluid function is operating (Table 16). Path p8 leads to paths p9 and p10 since there are no outgoing transactions. Along path 10 since the last function of holdup tank is evaluated, the path leads to identification of the next component. Holdup tank is connected to two components i.e., pressure sensor and pipe3. Since the simulation procedure is not set for parallel execution, pressure sensor component is selected first and pipe3 as next component.
Pressure sensor has one function i.e., measure pressure (Table 10) and has one inflow i.e., $P_{in}$, and two outflow i.e., $P_{out}$ and $<<signal>>T_1$ (Table 16). This leads to execution of the measure pressure function (Figure 40 along path p6). Since no faults are injected the execution of behavioral rules concludes the pressure sensor exhibits nominal behavior (Table 16). Along path p7 the execution of FFL indicates the measure pressure function is operating (Table 16). Path p8 leads to paths p11 and p12 since there is one outgoing transaction. Along path 12 a transaction $<<signal>>T_6$ is created and its [start, finish] value changes to [1, 0] which indicates its status is Active (will be updated in while execution of path p2). Eventually path p8 leads to paths p9 and p10. Along p10 since the pressure sensor component has only one function the path leads to identification of the next component i.e., pipe3, as mentioned earlier. Execution of pipe3 is identical to execution of pipe1 discussed before. Following pipe3 the next component is outlet valve.

Outlet valve has one function i.e., regulate fluid (Table 10), two inputs i.e., $Q_{ov_{in}}$, and $<<signal>>T_2$ or $<<signal>>T_3$ (Figure 42), and one output i.e., $Q_{ov_{out}}$. The transactions are not created by the software thus they have default values i.e., [start, finish] = [0, 0] which indicates its status as inactive (Table 9). Next, we execute the Regulate fluid function (Figure 40 along path p6). Since no faults are injected the execution of behavioral rules concludes the outlet valve exhibits nominal behavior (Table 16). Along path p7 the execution of FFL indicates the Regulate fluid function status as operating (Table 16). Since, the outlet valve has only one function we move on to identify next component in CFG. After outlet valve there are two components position sensor2 and
pipe4. We select position sensor2 as next component and then pipe4, since parallel execution of components in CFG is not currently possible. Execution of the position sensor2 is identical to execution of the previously encountered position sensor1. Execution of pipe4 is identical to that of pipe1 discussed earlier. However note that pipe4 is the last component of CFG thus path p10 leads to the end of CFG. End of CFG indicates end of HW design execution, which leads the execution path p1 to the AND2 gate.

Path p2: It leads to execution of the main activity diagram (Figure 17). The first activity read is configure system (along path A1 in Figure 40). The corresponding component is configuration manager (Figure 17), which has no inflows, one outflow i.e., ConversionData (Table 16) and no transactions. Further the path leads to A4 along which the activity configure system is executed which will modify the output variables. Path A4 then leads to two concurrent paths A5 and A6. Along A5 the behavioral rule of the component configuration manager is executed. Since no faults were injected the component exhibits nominal behavior (Table 16). Next the FFL is executed which indicates the configure system is operating. Path A6 leads to the next activity since there are no outgoing transaction from configure system. In figure 17 we see that after configure system the control flow branches out into two parallel flows due to the fork. Since parallel execution of activities has not been setup yet in ISFA simulation process, we will execute the activities in the following order: Read pressure, Calculate Level, Read position, Store Pos.
Next activity read is \textit{read pressure} (along path A1). The corresponding component is \textit{Sensor} (Figure 17), which has three inflows i.e., \texttt{signal}\texttt{\textless\textgreater}T1, \texttt{signal}\texttt{\textless\textgreater}T6, and \texttt{signal}\texttt{\textless\textgreater}T7, and two outflow i.e., Level and Pos (Table 16). The transaction input \texttt{signal}\texttt{\textless\textgreater}T1 (created during HW design execution i.e., path p1) is read without any error since no faults are injected. Thus the transaction’s [start, finish] values are updated to [1, 1] which indicates the status is \textit{complete} (Table 9). Note that the target of \texttt{signal}\texttt{\textless\textgreater}T6 and \texttt{signal}\texttt{\textless\textgreater}T7 are not \textit{read pressure} activity (Figure 17), thus its status is not updated. Along A4 the activity \textit{read pressure} is executed which will modify the output variables. Path A4 then leads to two concurrent paths A5 and A6. Along A5 the behavioral rule of the component \textit{sensor} is executed. Since no faults were injected the component exhibits nominal behavior (Table 16). Next the FFL is executed which indicates the \textit{read pressure} is \textit{operating} (Table 16). Path A6 leads to the next activity i.e., \textit{Calculate Level}.

The next activity is \textit{Calculate Level} and the corresponding component is \textit{sensor} (Figure 17), which has three inflows i.e., \texttt{signal}\texttt{\textless\textgreater}T1, \texttt{signal}\texttt{\textless\textgreater}T6, and \texttt{signal}\texttt{\textless\textgreater}T7, and two outflow i.e., Level and Pos (Table 16). The target of the transactions are not \textit{Calculate Level} activity (refer Figure 42), thus their status is not updated. Along A4 the activity \textit{Calculate Level} is executed which will modify the output variable \textit{Level}. Path A4 then leads to two concurrent paths A5 and A6. Along A5 the behavioral rule of the component \textit{sensor} is executed. Since no faults were injected the component exhibits nominal behavior (Table 16). Next the FFL is executed which indicates the \textit{Calculate Level} is \textit{operating}. Path A6 leads to the next activity i.e., \textit{Read position}.
The next activity is *read position* (along path A1 in Figure 40) and the corresponding component is *Sensor* (Figure 17), which has three inflows i.e., $<<\text{signal}>>T1$, $<<\text{signal}>>T6$, and $<<\text{signal}>>T7$, and two outflow i.e., Level and Pos (Table 16). The transaction inputs $<<\text{signal}>>T6$ and $<<\text{signal}>>T7$ are read without any error since no transaction faults are injected. Thus the transaction’s [start, finish] value changes to [1, 1] which indicates the status is complete (Table 9). Note that target of $<<\text{signal}>>T1$ is not *read position* activity (Figure 42), thus its status is not updated. Along A4 the activity *read position* is executed which will modify the output variables. Path A4 then leads to two concurrent paths A5 and A6. Along A5 the behavioral rule of the sensor is executed. Since no faults were injected the component exhibits nominal behavior (Table 16). Next the FFL is executed which indicates the *read position* is operating. Path A6 leads to the next activity i.e., *Store Pos*.

The next activity is *Store Pos* (along path A1) and the corresponding component is *Sensor* (Figure 17), which has three inflows i.e., $<<\text{signal}>>T1$, $<<\text{signal}>>T6$, and $<<\text{signal}>>T7$, and two outflow i.e., Level and Pos (Table 16). The target of transactions are not *Store Pos* activity (refer Figure 42), thus their status is not updated. Along A4 the activity *Store Pos* is executed which will modify the output variable Pos. Path A4 then leads to two concurrent paths A5 and A6. Along A5 the behavioral rule of the component sensor is executed. Since no faults were injected the component exhibits nominal behavior (Table 16). Next the FFL is executed which indicates the *Store Pos* is operating. Path A6 leads to the next activity i.e., *Valve control logic*. *Valve control logic* is further decomposed into other activities (Figure 18).
The valve control logic is enclosed in the component *Valve controller* (Figure 18), which has two inflows i.e., Level and Pos (Table 16) and, five outflow i.e., ControlCommand, <<signal>>T2, <<signal>>T3, <<signal>>T4, and <<signal>>T5 (Table 16). The execution of *Valve control logic*, traces the path D1- D2-D3-Exit. Thus the output variables are not modified.

The transactions <<signal>>T2, <<signal>>T3, <<signal>>T4, or <<signal>>T5 are not created, thus their [start, finish] values remain [0, 0] which indicates that their status is *inactive* (Table 9). Path A4 then leads to two concurrent paths A5 and A6. Along A5 the behavioral rule of the component *Valve controller* is executed. Since no faults were injected the component exhibits nominal behavior (Table 16). Next the FFL is executed which indicates the *Valve control logic* is *operating*. Since *Valve control logic* is the last activity the end of main activity diagram (Figure 17) is reached. End of AD indicates end of SW design execution, which leads the execution path p2 to the AND2 gate.

Paths p1 and p2 are synchronized at AND2 gate, which further leads to end of first simulation step. For the next simulation step the above procedure is repeated. During the execution of each step, the function status is captured and tabulated (Table 13). Some of the important results and their interpretation are discussed below.

At step t = 1, all the hardware and software functions were operating, the transactions <<signal>>T1, <<signal>>T6, and <<signal>>T7 were complete. The transaction <<signal>>T2, <<signal>>T3, <<signal>>T4, and <<signal>>T5 were inactivate because the pressure was within the operating range \([P_{Lth}, P_{Uth}]\) and both the valves were in open position (Figure 18). Thus the system function *transfer fluid* is operating.
At step $t=6$ a tank leak fault is injected as such the tank level started decreasing and reached a lower acceptable limit at $t = 10$. During this period from $t = 6$ to 10, all the HW and SW function were in operating state, the water supply from the holdup tank was not interrupted thus the system function *transfer fluid* was operating.

At step $t = 10$, the *holdup tank* pressure dropped below the lower threshold value ($P_{L,Th}$) on account of the leak, and the backup system was started. The *holdup tank’s* mode changed from nominal to dry-out, thus its functions i.e., *supply fluid* and *store fluid* was inferred as *Lost* (Table 16). The software function, the *valve control logic* (Figure 18) followed the path (D1-D5-D6-D7) causing a transaction $<<\text{signal}>>T2$ i.e., close outlet valve to occur. Thus with only inflow and no outflow the water level rose to desired range and the holdup tank was available for the next time step. Note that the back system pumped water from the reservoir for one unit of time, thus system function *transfer fluid* was operating.

At step $t = 11$ the *holdup tank* functions were back to *operating* state accompanied by a transaction change from $<<\text{signal}>>T2$ to $<<\text{signal}>>T3$. At this step the system behaved similar to that at step $t = 6$ eventually leading to the *holdup tank* functions loss at step $t = 15$. At step $t = 15$ the system behaves similar to that at step 10. The backup system was switched ON, and the transaction $<<\text{signal}>>T2$ occurred.

The above system behavior continued up to step $t = 100100$. At step $t=100100$, the outlet valve was closed. At this point the valve reached its fatigue limit. From the next time step onwards the outlet valve’s failure mode Failed Open is triggered. The backup system
supplied water for the next 5 units of time (i.e., until \( t = 100104 \)) after which the system failure occurred at step \( t = 100105 \).

In case 1, the pattern of transaction and HW function status is worth noting. The transaction change from \(<\text{signal}>T2\) to \(<\text{signal}>T3\) and vice-versa occurs every 4 units of time due to the leak. In the absence of leak the outlet valve state would remain open. Thus the transaction pattern indicates a symptom of fatigue failure of the outlet valve. The HW function status pattern indicates a symptom of small leak.

**Case 2:** Illustrates a hypothetical scenario of how a classic software modification fault (a commission error) evolves, and translates into system failure. Before the software modification the variable \( Pos \) was set to values \{1 or 0\} corresponding to \{Open or Close\}. These values were stored in the computer memory during execution of valve control logic (Figure 18). Later the software design was modified in congruence with holdup tank design change i.e., decision to add a position sensor to each valve. So the software was modified to read the valve’s position data from the position sensor instead of from the computer memory. However during the new software modification the position sensor data was read and the variable \( Pos \) was erroneously set to \{0 or 1\} corresponding to \{Open or Close\} while the valve control logic was copied without any modification. The analysis of this commission error accompanied with the tank leak (same as case 1) is performed using the ISFA simulation process (Figure 40). Important results of the simulation are explained below.

At step \( t = 1 \) (same as case 1), all the hardware and software functions were operating, the transactions \(<\text{signal}>T1, <\text{signal}>T6, \) and \(<\text{signal}>T7\) were complete. The
transaction <<signal>>T2, <<signal>>T3, <<signal>>T4, and <<signal>>T5 were inactive because the pressure was within the operating range \([P_{L\text{th}}, P_{U\text{th}}]\) and both the valves were in open position (Figure 18). Thus the system function \textit{transfer fluid} is operating.

Step t=6 system behavior was also same as case 1 (step t=6) explained earlier. However at step t =10, when the pressure goes below lower threshold, the execution of activity diagram (Figure 18) takes a different path (D1-D5-D6-D8-Exit) than in case 1. As a consequence the transaction <<signal>> T5 i.e., open inlet valve was observed while <<signal>>T4 remained inactive. At the same time the backup was switched ON, so the system function \textit{transfer fluid} was operating. However the \textit{holdup tank’s mode} changed from nominal to dry-out, thus its functions i.e., \textit{supply fluid} and \textit{store fluid} were inferred as \textit{Lost} (Table 16).

At step t = 11, the holdup tank pressure was still below the lower threshold value since the outlet valve was open. Thus the water kept draining from the tank accompanied by the leakage. So, the backup system was ON. This condition continued till step t = 14, when the backup system’s limited reservoir was depleted. During this period the system function \textit{transfer fluid} was operating.

After step t =14 till step t =19, the water supply from the tank was less than required due to the leak. Thus for 5 units of time the nuclear core did not get the required amount of water, leading to core uncover and thus the system function was \textit{Lost}.

In case 2, the pattern of transaction and HW function status is worth noting. The transaction <<signal>>T5 is always activated on account of combined software
modification fault and tank leak fault. In the absence of leak the outlet valve would always be open and the transaction $signal_2$ and $signal_3$ would always remain inactive in the presence of the software fault. Thus the given software fault will have no impact on the system function under nominal behavior of the components giving an impression that the software modification was correct. Thus observing the transaction status pattern can give an insight into the type of fault (HW, SW or both).

Case 1 and Case 2 demonstrate that we can propagate hardware fault and software faults independently and simultaneously. Also we can evaluate the system-level functional impact of the combined faults. In general, the system function loss can occur due to component or function failure, and interaction failure. Impact analysis of all these failures requires an integrated domain model representation to enable seamless fault propagation.
Case 1: Valve failure; Fault Injected: Tank Leak @ t=6

<table>
<thead>
<tr>
<th>Simulation Time (t)</th>
<th>Hardware Components and Functions</th>
<th>Interface</th>
<th>Software Components and Functions</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Inlet valve Holdup Tank Pressure sensor Pipe¹ Outlet valve Position Sensor²</td>
<td>I1 I2 I3 I4 I5</td>
<td>Configuration Manager Sensor Control valve</td>
</tr>
<tr>
<td></td>
<td>Regulate fluid Store Fluid Supply fluid Measure pressure Transfer fluid Regulate fluid Measure position</td>
<td>T1 T6 T7 T2 (C) T3 (O) T4 (C) T5 (O)</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>O O O O O O O C C C IA IA IA IA</td>
<td>O O O O O O O</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>O O O O O O O C C C IA IA IA IA</td>
<td>O O O O O O O</td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>O L L L O O O O C C C C IA IA IA IA</td>
<td>O O O O O O O</td>
<td></td>
</tr>
<tr>
<td>11</td>
<td>O O O O O O O C C C IA IA IA IA</td>
<td>O O O O O O O</td>
<td></td>
</tr>
<tr>
<td>15</td>
<td>O L L L O O O O C C C C IA IA IA IA</td>
<td>O O O O O O O</td>
<td></td>
</tr>
<tr>
<td>20</td>
<td>O L L L O O O O C C C C IA IA IA IA</td>
<td>O O O O O O O</td>
<td></td>
</tr>
<tr>
<td>...</td>
<td>... ... ... ... ... ... ... ... ...</td>
<td>... ... ... ... ... ... ... ...</td>
<td></td>
</tr>
<tr>
<td>100100</td>
<td>O L L L O O O O C C C C IA IA IA IA</td>
<td>O O O O O O O</td>
<td></td>
</tr>
<tr>
<td>100105</td>
<td>O L L L O O O O C C C IA IA IA IA</td>
<td>O O O O O O O</td>
<td></td>
</tr>
</tbody>
</table>

**Case 2: Incorrect modification of software; Fault Injected: Tank Leak @ t=6**

<table>
<thead>
<tr>
<th>Simulation Time (t)</th>
<th>Hardware Components and Functions</th>
<th>Interface</th>
<th>Software Components and Functions</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Inlet valve Holdup Tank Pressure sensor Pipe¹ Outlet valve Position Sensor²</td>
<td>I1 I2 I3 I4 I5</td>
<td>Configuration Manager Sensor Control valve</td>
</tr>
<tr>
<td></td>
<td>Regulate fluid Store Fluid Supply fluid Measure pressure Transfer fluid Regulate fluid Measure position</td>
<td>T1 T6 T7 T2 (C) T3 (O) T4 (C) T5 (O)</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>O O O O O O O C C C IA IA IA IA</td>
<td>O O O O O O O</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>O O O O O O O C C C IA IA IA IA</td>
<td>O O O O O O O</td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>O L L L O O O O C C C IA IA IA IA</td>
<td>O O O O O O O</td>
<td></td>
</tr>
<tr>
<td>11</td>
<td>O O O O O O O C C C IA IA IA IA</td>
<td>O O O O O O O</td>
<td></td>
</tr>
<tr>
<td>14</td>
<td>O L L L O O O O C C C IA IA IA IA</td>
<td>O O O O O O O</td>
<td></td>
</tr>
<tr>
<td>15</td>
<td>O L L L O O O O C C C IA IA IA IA</td>
<td>O O O O O O O</td>
<td></td>
</tr>
<tr>
<td>19</td>
<td>O L L L O O O O C C C IA IA IA IA</td>
<td>O O O O O O O</td>
<td></td>
</tr>
</tbody>
</table>

**Table 13: Simulation Results. O = Operating, L = Lost, IA = Inactive, C = Complete**

Imagine \( Q_{in} = Q_{out} = 10 \) units; \( Q_{leak} = 2 \) units; \( L_L = 2 \) units; \( L_U = 20 \) units. Hence \( Q_{out} \) of tank is 12 units and Level decreases by 2 units in each simulation step.

¹Pipe' corresponds to for pipe1, pipe2, pipe3, and pipe4
²Position sensor corresponds to both the inlet and the outlet valve’s position sensors
Chapter 6. Mathematical Formulation

The mathematical formulation of the UML-based fault propagation analysis is a challenging task due to the varied nature of the diagrams, inter-dependency between different diagram elements, and lack of mathematical basis for several diagrams such as use-case, activity diagram, and class diagram. For graph-based software representations, some of the mathematical formalism that enables fault propagation analysis are:

1. Markov chains, for state-based representation – Suitable for fault propagation analysis of event based systems.
2. Petri-nets for place/transition based representation – Suitable for modeling the concurrent behavior of distributed systems. Petri-nets can be modified for fault propagation analysis
3. Event Sequence Diagram formulation for ESD based representation – Suitable for analysis of scenario-based event progression leading to failure.
4. First order logic and Predicate logic- Suitable for deductive analysis

These mathematical formulations are different in nature, suitable for a particular purpose, and not necessarily inter-operable. Developing a hybrid-integrated mathematical basis for the UML-based representation of the software-intensive systems to analyze fault propagation is thus non-trivial.
In my research, the proposed fault propagation approach is activity diagram driven. The activity diagram models the functional (or process) flow within the software system. Hence I choose to adopt, and develop a function-based mathematical approach. The central concept in this approach is called flat parts [Wei, 2006]. In this chapter, I introduce the concept of flat parts and its use in determining the fault propagation probability. I propose interval-arithmetic based rules to determine the flat parts. Seven unique rules are defined for the basic arithmetic operations, and three for advanced operators. These rules can be integrated into a tool along with a control-flow based algorithm, which determines the faulty variables and the sequence of functions. Flat part based fault propagation analysis can give quick and accurate results early on. Function based propagation is not a widely studied topic of research; however it could be an indispensable approach to design of software intensive systems.

A function is defined as a relationship between an input and an output. The relationship between the input and output can be continuous or discrete, linear or non-linear, and constant or non-constant. The input-output mapping could be one-to-one, one-to-many, many-to-one or many-to-many. As illustrated in figure 45 an input space can be decomposed into two parts: non-propagation region (FP1 and FP2) and propagation region (complement of FP1 and FP2). If the expected input (i.e. the input that would be provided to the function if there was no active fault on the path of execution) as well as faulty input (i.e. the input provided to the function if there is an active fault on the path of execution) fall in the non-propagation region, then the propagation probability of the
faulty input is zero; otherwise the propagation probability is one. Inputs that fall in the propagation region are used to define the probability of propagation.

![Graph showing flat parts FP1 and FP2 with points a, b, and c corresponding to the same output for different input values.]

Figure 45 Illustration of flat parts

A fault is said to be masked if multiple input values have a single output value. Such a set of inputs or a range of input is called the “fault masking area” a.k.a. “Flat part” (FP) [Wei et al., 2010]. For example, FP1 and FP2 correspond to the flat parts (Figure 45). In addition, points like a, b, c also corresponds same output for different input values. However, for simplicity and easier comprehension of proposed arithmetic, we will restrict the analysis to continuous flat parts of type FP1 and FP2. For discrete points (a, b, c) discrete arithmetic principles must be followed. Flat part is a property of a function. A flat part will either be preserved or generated when different functions interact with each other.

In this research, we address the questions: when a will a flat part will be preserved? and when will a flat part be generated? We do so by analyzing the fundamental elements that define a complex expression like a multinomial algebraic expression [Ramamoorthy, 1982]. The fundamental elements include the basic arithmetic operations like addition,
subtraction, multiplication, division, and advanced operations like integration, differentiation, and composition. These basic arithmetic operations and their impact on flat part modification will be explored. We will define the conditions and rules that generate, preserve or kill a FP. Further we propose to use the FPs to calculate the propagation probability of a function resulting from the combination of complex expressions.

Among these operations, function composition is the trickiest. Hence, analytical expressions of flat part determination are derived and discussed in section V. Function composition is also one of the most important operations emerging from the control-flow of the software. Control-flow analysis is a commonly used static analysis technique. It captures the software functional flow by connecting the function under consideration with the source functions and target functions. A control-flow analysis reveals different execution paths, which are composed of different functions. Each of these paths can be decomposed into a canonical form as presented in figure 46. We propose to exploit this canonical form to perform the fault propagation analysis.

![Figure 46 Canonical form of an execution path](image)

Thus, the flat part determination rules and their use to calculate the propagation probability is the main contribution and first step towards use of interval arithmetic for fault propagation analysis. We decompose a complex problem into its fundamental
elements; solve these fundamental problems, and eventually build a complete solution. We identify the fundamental concept underlying fault propagation as the fault masking area or flat part.

Since the flat part is an invariable property of the system, the approach can be used to determine the “maximum reliability a system can attain for a given input profile”.

The aim of this research is to lay the foundation for a function based approach to fault propagation, which can be used for reliability analysis.

**Basic Arithmetic Operations And Flat Part Determination**

The basic elements of an arithmetic expression are “operands” and “operation”. The basic operations are addition, subtraction, multiplication and division. More advanced operations are differentiation, integration and composition. An operand is an object on which the operation is applied. These basic and advanced operations and operands are the building blocks of the complex functions. Any complex function can be described by a series of operands on which one or more operations are performed. For example, consider \( f(x) = x^2 + 3 \); where \( x \) and \( 3 \) are called operands acted upon by multiplication (*) and addition (+) operations.

For the basic and advanced operations fundamental FP transformation rules, which are the useful formulae with extensive application, are developed in this research. Some rules are further extended as lemmas for completeness. These rules are the building blocks for FP analysis of complex functions. The fundamental FP transformation is expressed as proposition 1.
Proposition 1: FP transformation

The fundamental equation defining the flat part (FP) of any complex function is composed of two parts: i) FP newly generated from non flat parts, $F_P$; ii) FP generated from existing FPs. The fundamental equation is given by eq.1.

$$FP_{total} = FP_{generated} \bigcup FP_{preserved}$$

...(1)

The terms in eq. 1 are defined below and are applied in figures 47-50.

**FP generated**: FP is said to be generated when an original function does not contain a FP but on applying the basic or advanced arithmetic operators to the function a FP appears in the resulting function. Each operation has different FP generation rules.

**FP preserved**: FP is said to be preserved when an original function contain a FP and after applying the basic or advanced arithmetic operators, a part or the entire FP appears in the resulting function. For the four basic operations, FP preserved is given by eq. 2 below.

**Rule 1**: When ‘$n$’ different functions $f_i(x)$, where $i = 1..n$, are combined together with the basic arithmetic operations (addition, subtraction, multiplication, division) and each function $f_i(x)$ contains a flat part, $FP_{f_i}$, where $f_i$ is the $i^{th}$ function and $x$ is the input to the function, then the intersection of all the flat parts is the flat part preserved.

$$FP_{preserved} = \bigcap_{i=1}^{n} FP_{f_i}$$

...(2)

The complement of FP preserved is called FP killed. FP is said to be killed when an original functions contain a FP and after applying the basic or advanced arithmetic
operators a part or entire FP disappears in the resulting function. For the four basic
operations, FP killed is given by lemma 1 that is extended from rule 1.

**Lemma 1:** When ‘n’ different functions are combined together with the basic arithmetic
operations (addition, subtraction, multiplication, division), and each function contains a
flat part, then the union of all the flat parts minus the FP preserved is the flat part killed.

\[
FP \text{ killed } = \bigcup_{i=1}^{n} FP_{f_i} - FP \text{ preserved} 
\] ...

The minus sign in eq.3 is tricky to handle. FP killed can also be calculated
differently as discussed in Lemma 2.

**Lemma 2:** FP is killed when the FP of a function and a non-FP of other functions are
combined with a basic operation as presented in eq. 4.

\[
FP^f \cup \overline{FP}^g 
\] ...

where \( \cup \) denotes a basic operation

Then the FP killed is can be calculated by eq. 5,

\[
FP \text{ killed } = \left( \bigcup_{i=1}^{n} FP_{f_i} \right) \cap \left( \bigcup_{j=1}^{m} \overline{FP}_{g_j} \right) 
\] ...

Where, \( n \) and \( m \) are number of flat parts of functions \( f \) and \( g \) respectively.

Comparing equations 3 and 5 we can say that the minus sign is equivalent to the
intersection with the union of non-flat parts of a function.

The above proposition and rules are further developed for each basic operation and
advanced operations and demonstrated with the two piece-wise functions \( f(x) \) and \( g(x) \) to
demonstrate the rules for flat parts determination:
\[ f(x) = \begin{cases} 
 x + 3, & -8 \leq x \leq 5 \\
 1, & \text{otherwise} 
\end{cases} \]

\[ g(x) = \begin{cases} 
 x, & 1 \leq x \leq 8 \\
 5, & \text{otherwise} 
\end{cases} \]

For the above functions the flat parts are:

\[ FP_f = (-\infty, -8) \cup (5, \infty) \]

\[ FP_g = (-\infty, 1) \cup (8, \infty) \]

Piece-wise functions are common in electrical, electronics and software domain. For example in [Kehl et al., 2011], the latching probability of a transient is expressed as a piece-wise function with three parts: 0, 1, and a function of time. A logical function has if-then-else structure with each part describing different function for a specific range of input values.

A. Addition: \(f(x) + g(x)\)

FP resulting from an addition operation can be determined using the eq. 1, eq. 3 and the rule 2 given below. The rule is used to determine the FP generated.

**Rule 2:** Flat parts are generated when one function is negated and added to the other given the input \(x\) is same for both functions (eq. 6)

\[ g(x) = -f(x) \pm \text{constant} \quad \cdots (6) \]

The FP resulting from the addition of \(f(x)\) and \(g(x)\) is illustrated in figure 47.
8. **Subtraction: \( f(x) - g(x) \)**

FP resulting from a subtraction operation can be determined using the eq. 1, eq. 3 and the rule 3 given below. The rule is used to determine the FP generated.

**Rule 3:** Flat parts are generated when one function is subtracted from another equivalent function given the input \( x \) is same for both functions (eq. 7)

\[
g(x) = f(x) \pm \text{constant} \quad \text{...(7)}
\]

The FP resulting from the subtraction is illustrated in figure 48.
The subtraction operation generated a FP over the range (1, 5). Thus subtraction operation will have a greater region of non-propagation compared to the addition operation for the given operands \( f(x) \) and \( g(x) \).

The above result can used as a guiding principle in defining fault detection and diagnosis algorithms. For example, we can define a fault detection mechanism over the range (1, 5) by transforming \( f(x) \) or \( g(x) \).

C. **Multiplication: \( f(x) \times g(x) \)**

FP resulting from a multiplication operation can be determined using the eq. 1, eq. 3 and the rule 4 given below. The rule is used to determine the FP generated.

**Rule 4:** Flat parts are generated when one function is inversely proportional to the other given the input \( x \) is same for both functions (see eq. 8).
\[ f(x) \propto \frac{1}{g(x)} \]  

...(8)

The FP resulting from the multiplication is illustrated in figure 49.

\[
F_P \text{ generated} = \emptyset \\
F_P \text{ preserved} = (-\infty, -8) \cup (8, \infty) \\
\therefore F_P \text{ total} = (-\infty, -8) \cup (8, \infty) \\
F_P \text{ killed} = (-8, 1) \cup (5, 8)
\]

Figure 49: Multiplication

D. Division: \( f(x)/g(x) \)

FP resulting from a division operation can be determined using the eq. 1, eq. 3 and the rule 5 given below. The rule is used to determine the FP generated.

**Rule 5:** Flat parts are generated when one function is directly proportional to the other given the input \( x \) is same for both functions (see eq. 9).

\[ f(x) \propto g(x) \]  

...(9)

The FP resulting from the division is illustrated in figure 50.
Advanced Arithmetic Operations And Flat Part Determination

The advanced operators are commonly found in engineering applications. These operators include integration, differentiation, composition and others. FP rules for these operators are developed here.

A. Integration: $\int f(x) \, dx$

FP resulting from a division operation can be determined using the eq. 1 and rules 6 and 7 given below. The rules are used to determine the FP generated and FP preserved.

**Rule 6: Flat parts are never generated**

**Rule 7: Flat parts are preserved if and only if:**

$$f(x) = 0 \quad \text{...(10)}$$

The range of $x$ over which eq. 8 is satisfied gives the FP preserved.

```plaintext
FP generated = \emptyset
FP preserved = (\inf, -8) \cup (8, \inf)
\therefore FP total = (\inf, -8) \cup (8, \inf)
FP killed = (-8, 1) \cup (5, 8)
```

Figure 50: Division
Lemma 3: Flat parts are always killed for integration operation.

The FP resulting from the integration of $f(x)$ is illustrated in figure 51.

![Integration](image)

**Figure 51: Integration**

\begin{align*}
\text{FP generated} & = \emptyset \\
\text{FP preserved} & = \emptyset \\
\therefore \text{FP total} & = \emptyset \\
\text{FP killed} & = \emptyset
\end{align*}

**B. Derivative: $\frac{df(x)}{dx}$**

FP resulting from a division operation can be determined using the eq. 1 and rules 9 and 10 given below. The rules are used to determine the FP generated and FP preserved.

**Rule 8:** Flat parts are generated for all values of the input $x$ if the function is a constant ($\mathbb{S}$) or a linear function of $x$

\[ f(x) = \mathbb{S} \text{ or a linear function of } x \]  

**Rule 9:** Flat parts are always preserved for derivative operation.
Lemma 4: Flat parts are never killed for derivative operation.

The FP resulting from the differentiation of $f(x)$ is illustrated in figure 52 and calculated as below

$$\text{FP generated} = (-8, 5)$$
$$\text{FP preserved} = (-\infty, -8) \cup (5, \infty)$$
$$\therefore \text{FP total} = (-\infty, \infty)$$
$$\text{FP killed} = \emptyset$$

Figure 52: Differentiation

C. Composition: $g(x) \circ f(x)$

Composition is an important operation particularly for software evaluation. Software execution involves one function calling another in a specific order, hence can be expressed as a composition of functions. Function composition is a tricky operation, since it does not obey the commutative property. The commutative property allows independent execution of functions; lack of it induces dependencies in between the functions. The composition operation and the rules for FP determination are developed explicitly for different types of functions in the next section.
**Flat Parts Of Function Composition**

The FP determination of function composition is based on the following properties of composition:

1. The order of functions is important.
2. The domain of composition of $f_2 \circ f_1(x)$ is always a subset of domain of $f_1(x)$.
3. The range of composition of $f_2 \circ f_1(x)$ is always a subset of range of $f_2(x)$.

Hence, if $f_1(x)$ is a constant, $f_2 \circ f_1(x)$ is a constant over the corresponding domain of $f_1(x)$. In other words, we can say that FP is preserved and the FP preservation is not sensitive to the subsequent function type. This is not the case for the basic arithmetic operations seen earlier. The rule for FP preservation for composition can thus be given as rule 10.

**Rule 10**: If $f_i$ contains a FP, then the FP is preserved by subsequent composition operations given $f_i$ is not an integration operation.

Since the entire FP is preserved, new FP can be generated only from the non FP portion ($\overline{FP}$) of the function $f_i$. For composition $f_2 \circ f_1(x)$, $\overline{FP}$ can be calculated as follows:

\[
\overline{FP} \text{ of } f_2 \circ f_1(x) \text{ over } x = \text{Domain } f_1 \bigcap \text{Domain } f_2 \circ f_1(x)
\]

In general, consider the composition of “$n$” functions represented as follows:

\[
g_n(x) = f_n \circ ... \circ f_i \circ f_1(x); \forall i = 1..n \quad \quad \quad \quad \quad \quad \quad \quad (12)
\]
\( \overline{FP} \) of \( g_n(x) \) over the domain of \( x \) can be calculated as given in eq. 13 below:

\[
\overline{FP} \text{ of } g_n(x) \text{ over } x = \quad \ldots(13)
\]

\[
\text{Dom } f_1 \bigcap \left[ \text{Dom } f_2 \circ f_1(x) \right] \bigcap \ldots \bigcap \left[ \text{Dom } f_n \circ f_1(x) \right]
\]

Where, the domain \( [\text{Dom } f_n \circ f_1(x)] \) is calculated as follows (eq. 14):

\[
\alpha_L^{f_n} \leq f_{n-1} \circ \ldots \circ f_1(x) \leq \alpha_U^{f_n}
\]

\[
f_1^{-1} \circ \ldots \circ f_{n-1}^{-1}(\alpha_L^{f_n}) \leq x \leq f_1^{-1} \circ \ldots \circ f_{n-1}^{-1}(\alpha_U^{f_n}) \quad \ldots(14)
\]

Note, while calculating the domain, the order of the functions being composed must be strictly followed, since composition does not obey the commutative property. But while calculating the \( \overline{FP} \) the order of domain does not matter, since set operations obey commutative property.

As FP generation is dependent on the nature of the functions, we derive the FP generation equation for a constant, and linear function types. These function types are commonly encountered during approximation of system design equations. For instance, a complex equation involving non-linear functions, exponential functions etc. can be approximated as a linear function about point \( x \) or expressed using power series expansion.

A. Constant

Let us consider in \( g_n(x) \), the function \( f_i(x) \) is a constant \( \subseteq \) for all real values of \( x \):

\[
f_i(x) = \subseteq \quad \forall x \in \mathbb{R}
\]

Then the composition \( g_n(x) \) that includes function \( f_i(x) \) will preserve the FP of \( f_i(x) \)

\[
FPg_n = FPf_i
\]

154
B. Linear

We will now establish the $FP$ equations for cases where all the functions $f_n, \ldots, f_1$ in $g_n(x)$ are linear functions.

Case 1: Consider the case where $f_i$ in $g_n(x)$ is a linear function of the form eq. 15:

$$f_i(x) = a_i \cdot x + c_i \quad \forall i \in [1, n]$$

Composition of functions of the type $f_i(x)$ does not generate any flat parts.

$$g_n(x) \neq \emptyset \Rightarrow FP^{g_n} = \emptyset$$

Case 2: Consider $f_i$ in $g_n(x)$ is piecewise linear function of the form eq. 16:

$$f_i(x) = \begin{cases} 
  a_i \cdot x + c_i & \text{if } \alpha_L^{f_i} \leq x \leq \alpha_U^{f_i} \\
  \emptyset & \text{otherwise} 
\end{cases} \quad \ldots(16)$$

For the composition of two functions,

$$g_2(x) = f_2 \circ f_1(x)$$

The domain of $g_2(x)$ is given as:

$$Dom \ g_2(x) = [\alpha_L^{g_2}, \alpha_U^{g_2}]$$

where the limits $\alpha_L^{g_2}, \alpha_U^{g_2}$ are calculated using eq 14. as follows,

$$f_1(x) = \alpha_{L}^{f_2}$$

$$a_1 \cdot x + c_1 = \alpha_{L}^{f_2}$$

$$\therefore \alpha_{L}^{g_2} = x = \frac{\alpha_{L}^{f_2} - c_1}{a_1} \quad \ldots(17)$$
Similarly,

\[ \alpha_{U}^{g^2} = \frac{a_1 f_2 - c_1}{a_1} \] \quad \ldots(18)

For the composition of three functions,

\[ g_3(x) = f_3 o f_2 o f_1(x) \]

The domain of \( g_3(x) \) given as:

\[ \text{Dom } g_3(x) = [\alpha_{L}^{g_3}, \alpha_{U}^{g_3}] \]

where, the limits \( \alpha_{L}^{g_3}, \alpha_{U}^{g_3} \) are calculated using eq. 14 as follows,

\[ f_2(f_1(x)) = \alpha_{L}^{f_3} \]

\[ a_2 * (a_1 * x + c_1) + c_2 = \alpha_{L}^{f_3} \]

\[ \therefore \alpha_{L}^{g_3} = x = \frac{\alpha_{L}^{f_3} - (a_2 * c_1 + c_2)}{a_2 * a_1} \] \quad \ldots(19)

Similarly,

\[ \alpha_{U}^{g_3} = \frac{\alpha_{U}^{f_3} - (a_2 * c_1 + c_2)}{a_2 * a_1} \] \quad \ldots(20)

Hence, by induction, for the composition of \( n \) functions; the domain of \( g_n(x) \) is given as follows:

\[ \text{Dom } g_n(x) = [\alpha_{L}^{g_n}, \alpha_{U}^{g_n}] \]

\[ \alpha_{L}^{g_n} = \frac{\alpha_{L}^{f_n} - C_1}{C_2} \]

\[ \alpha_{U}^{g_n} = \frac{\alpha_{U}^{f_n} - C_1}{C_2} \]

Where, \( C_1, C_2 \) are constants:
\[ C_1 = \left( \prod_{i=2}^{n-1} a_i \right) c_1 + \left( \prod_{i=3}^{n-1} a_i \right) c_2 + \cdots + c_{n-1} = \sum_{j=1}^{n-2} \left( \prod_{i=j+1}^{n-1} a_i \right) c_j + c_{n-1} \]

\[ C_2 = \prod_{i=1}^{n-1} a_i \]

Observe that the domain of \( g_n \) may increase or decrease depending on the values of \( C_2 \) only, while \( C_1 \) displaces the upper limit and lower limit of domain of \( f_n \) equally in the same direction. The impact of \( C_1 \) will only be realized during the \( \overrightarrow{FP} \) calculation. If \( C_1 \) displaces the domain away from domain of \( f_1 \), then the intersection of domains will lead to NULL value.

**Limiting condition for the domain of \( g_n \):**

1. When \( C_2 \to \infty \)
   \[
   \lim_{C_2 \to \infty} (\alpha_L^{\theta_n}) = 0 \quad \text{and} \quad \lim_{C_2 \to \infty} (\alpha_U^{\theta_n}) = 0
   \]
   \[
   \therefore \lim_{C_2 \to \infty} (\overrightarrow{FP}^{\theta_n}) = 0
   \]

2. When, \( C_2 \to 0 \)
   \[
   \lim_{C_2 \to 0} (\alpha_L^{\theta_n}) = \inf \quad \text{and} \quad \lim_{C_2 \to 0} (\alpha_U^{\theta_n}) = \inf
   \]
   \[
   \therefore \lim_{C_2 \to 0} (\overrightarrow{FP}^{\theta_n}) = \inf
   \]

3. When value of \( C_1 \) is such that the domain of \( g_n(x) \) is outside the domain of \( f_1(x) \), then \( \overrightarrow{FP}^{\theta_n} = \emptyset \)

Thus depending on the values of \( C_1 \) and \( C_2 \), the FP of \( g_n(x) \) may or may not exist.

These limiting conditions are expressed as Lemma 5.
**Lemma 5**: *A composition of “n” functions (of the form \( f_i \) given by eq.14) may generate a flat part over the entire domain of \( x \), if the limit of \( C2 \) tends to infinite, where \( C2 \) is the product of coefficients of ‘n-1’ functions. On contrary, if \( C2 \) tends to zero, no new flat parts will be generated.*

The rules developed for the linear function can be used to demonstrate the impact of an input error on the FP. Consider a special case of eq 16, when the constant term \( c_i \) is zero, the function \( f_i \) is called a scaling function. The coefficient \( a_i \) is the scaling factor, which plays a critical role in FP generation. The importance of the scaling factor can be observed from figures 53 and 54. An incorrect scaling will lead to different fault propagation characteristics compared to the correct scaling factor.

Figure 53 shows that the result of function composition of \( g(x) \) with \( f(x) \). Notice that the FP of \( f(x) \) is entire preserved while FP of function \( g(x) \) is not entirely preserved after being composed with \( f(x) \).

If we scale \( x \) by 10 the FP of the function composition will be different from one observed in figure 53. The result in figure 54 shows that the FP resulting from scaling is larger than the FP obtained from *not scaling* \( x \) (figure 53). Similarly, scaling \( x \) by 1/10 produces different result. The FP resulting from down scaling is smaller than the FP obtained from *not scaling* \( x \).

Thus any fault in input values of \( x \), will change the FPs of the function composition. Different fault types will create or preserve FPs differently, thus displaying different propagation pattern [Hiller et al., 2002]. The FP rules can be used to analyze different
types of faults such as missing function, extra function, incorrect input etc, and their impact on FP.

Figure 53: Functions f(x) and g(x)

Figure 54 Composition g o f(x)

Figure 55 Composition g o f(10x)

Figure 56 Composition g o f(10x)

159
The composition of functions can be derived from the control flow diagram of a software system (figure 57). The control flow diagram imitates the execution of the software functions and the corresponding execution paths. These execution paths in presence of a fault can be derived using algorithm 1. Algorithm 1 implements a forward tracing program and a variable-state determination statemachine (figure 58). The statemachine models the dynamics of the variable state change.

Figure 57 Example function flow of software system with missing function fault MF

Where,

\[ \text{In}[f_i] \] the input variable of the function \( f_i \) eg. \( in1, x, y \)

\[ \text{Out}[f_i] \] output variable of the function \( f_i \) eg. \( x, y, out1 \)
The terms used in figure 58 are as follows,

\( d \) = “defined state” due to a non-faulty input

\( df \) = “faulty defined state” due to a faulty input

\( u \) = “used state” transitioned from a non-faulty defined state

\( uf \) = “faulty used state” transitioned from a faulty defined state

\( k \) = “killed state”

Use\([Fi]\) = Set of variables used to define the output variable of function \( f_i \)

Use\([Fi]\).state = State of variables used to define the output variable in function \( f_i \)

Var.Transition = State transition of the variable under consideration

out\([Fi]\).state = State of variable under consideration used in function \( f_i \)
Algorithm 1: Missing function propagation algorithm (MFPA)

1. Execute path finding algorithms
2. For all paths $P_m$ containing a missing function.
3. Initialize $in[f_i].state = \{d\}$ // No INPUT faults assumed
4. For all functions $f_i$ (i++) // Forward tracing to update the variable
   // states, both faulty and non-faulty
5. $in[f_i] = out[f_{i-1}]$ // Skip for first function i.e. $i = 1$
6. If $f_i \in Missing function$
7. $out[f_i].state = \{df\}$
8. $i = i + 1$ // Next function
9. Endif
10. For all $out[f_i]$
11. Select all the inputs used to define $out[f_i]$ into $use[f_i]$
12. executeState($use[f_i], in[f_i].state, out[f_i].state$) // See Fig 58
13. End for (j)
14. End for (i) // Path $m$ covered and input-output states updated

Forward Tracing
Identify all the defective variables and the corresponding functions using these variables

Missing Function Fault

In this section we will demonstrate the application of composition rules on the missing function fault. A missing function fault is an omission type of fault, which occurs when the system designer or coder omits a function due to oversight or missing requirement. This fault can be rectified by inserting the missing function in the desired location of the existing function flow. For this demonstration consider figure 59, where the faulty function composition and the composition after inserting the missing fault (MF) is presented graphically. It indicates the function MF should be between function F1 and F3; however it was missing in the original control-flow specification.
The function $f_1$, $MF$, and $f_3$ are defined as follows:

$$f_1(x) = f(x) \quad \text{... (as in section III)}$$

$$MF(x) = \begin{cases} x - 2, & 0 \leq x \leq 5 \\ 3, & \text{otherwise} \end{cases}$$

$$f_3(x) = g(x) \quad \text{... (as in section III)}$$

For the faulty composition, the $FP$ entails

$$g_{\text{faulty}} = f_3(f_1(x)) = g(f(x))$$

$Dom\ f_1(x) = [-8,5] \quad \text{...(given)}$

$Dom\ f_3 \circ f_1(x) = [-2,5] \quad \text{...(using eq. 17, 18)}$

$$:. \ FP\ of\ g_{\text{faulty}}\ over\ x = [-8,5] \bigcap [-2,5]$$

Hence, $FP$ of $g_{\text{faulty}}$ over $x$ is $[-2,5]$ (see figure 53)

For the correct composition, the $FP$ entails

$$g_{\text{correct}} = f_3(MF(f_1(x))) = g(MF(f(x)))$$

$Dom\ MF \circ f_1(x) = [-3,5] \quad \text{...(use eq.17,18)}$

$Dom\ f_3 \circ MF \circ f_1(x) = [0,7] \quad \text{...(use eq.19,20)}$
∴ \( \bar{FP} \) of \( g_{\text{correct}} \over x = [-8, 5] \cap [-3, 5] \cap [0, 7] \)

\( \bar{FP} \) of \( g_{\text{correct}} \) over \( x \) is [0,2]. Thus, in the correct version FP was generated in the interval \([-2,0] \cup [2,5]\) (figure 60), in which any input fault will not propagate. However, within this range for the faulty version, propagation will occur.

Figure 60 Composition of \( f_3 \circ MF \circ f_i \)

**Propagation Probability Calculation**

The propagation probability is a complement of non-propagation probability. Non-propagation probability will be calculated first, since the existence of a FP implies non-propagation. Non-propagation requires that correct input values (\( x_E \)) and faulty input values (\( x_A \)) fall in the same FP region. According to PIE theory [19], in the presence of software faults the three necessary and sufficient conditions to failure: fault propagation (P), fault infection (I), and fault execution (E). And the reliability is calculated as

\[
\text{Reliability} = 1 - \sum_{\text{all faults}} (P \ast I \ast E)_i
\]
Where, P, I and E are probabilities of propagation (P), infection (I), and execution (E) probabilities respectively. Thus, a lower propagation implies higher reliability value.

The propagation probability (eq. 22) is the complement of npp, where the non-propagation probability (npp) is calculated by integrating the distribution of x over all the flat parts (eq. 21).

\[
npp = \int_{x} (\text{distribution of } x) \, dx \quad \text{ ...(21)}
\]

\[
pp = 1 - npp \quad \text{ ...(22)}
\]

The distribution of x is called the operational profile of the input x. Figure 61 illustrates the concept of pp and npp when x has a normal distribution.

![Figure 61: NPP calculation](image)

For the missing fault example discussed in section V, the propagation probability of the faulty version and the corrected version can be calculated as follows:

\[
pp_{\text{faulty}} = 1 - \int_{-2}^{5} (\text{distribution of } x) \, dx
\]
\[ pp_{correct} = 1 - \int_{0}^{2} (\text{distribution of } x) \, dx \]

Thus, in the above case we can say, \( pp_{faulty} \) is greater than \( pp_{correct} \). Further, applying the PIE theory indicates the reliability of the correct version will be greater than the faulty version, if the infection probability and execution probability are assumed to be the same for both faulty and correct version.
Chapter 7. Conclusions and Discussion

Design stage fault propagation and reliability analysis was the focus of my research. The outcomes of the research are three-fold: an executable FPSA method; an executable ISFA method; and an initial mathematical basis for fault propagation and reliability calculation.

The FPSA method is an UML-based approach, is formalized using UML meta-modeling concepts, and includes executable elements of the communication models and action models of UML’s superstructure. The FPSA method has two levels of execution: high-level and low-level. The high-level execution predicts the possible high-level fault propagation paths through the activity diagram and the functional impact of faults. The low-level execution allows execution of events created during the high-level execution. The high-level execution includes execution of the Action Model ESD; while the low-level execution includes execution of the Interaction model ESD. It should be noted that the information gathered from the FPSA execution at the low-level could actually contradict the predictions made from the high-level execution. Such a contradiction was observed in case 1.1 (chapter 4).

The observations and insights obtained from the simulation of five different faults demonstrate the power of the proposed FPSA technique. The faulty cases show that interesting fault-failure patterns can be developed that may be further used for fault
detection or identification of locations to place safeguards. The technique has significant potential and can be transformed into a tool for early risk and reliability analysis of object-oriented systems designed using the UML specification.

The discussion on the potential number of fault combinations raises the issue of fault space explosion. The fault space could be reduced by considering the semantically similar UML elements. Three potential ways to reduce the fault space are identified. First, build dependency rules or constraint specifications for semantically similar diagram elements. Example dependency rules were developed in [Briand et al., 2009; Briand et al., 2006] for impact analysis. In [Briand et al., 2006], each rule is expressed formally in the object constraint language (OCL). These rules should be able to extract elements that have identical fault propagation characteristics. For example a “missing class” and “missing all objects of the class” will have identical fault propagation characteristics and as such only one of the two should be studied. These dependency rules will reduce the fault space significantly. Secondly the fault space could be reduced by considering the likelihood that a fault will affect a particular design element (using fault statistics similar to those discussed in chapter 4). Finally, rules to identify harmless fault combinations such as the one discussed in case 3 will further reduce the fault space.

The ISFA method is presented as a method to enhance traditional techniques such as Failure Modes and Effects Analysis (FMEA) and Fault Tree Analysis (FTA) by addressing some of the inherent difficulties of using these methods in complex systems. The ISFA method provides constructs for multiple-domain representation, thereby providing a unique system-level model. In addition, ISFA provides an execution model to
simulate the system model. This enables designers to understand fault propagation paths and the fault interactions that may lead to functional failures and help them improve the system quality at the earliest stages of the design process.

The propagation paths are a systematic outcome of each simulation step. In the case-study discussed, the severity of a fault is dependent upon the time to system failure. For example, the severity is very low if the system failure occurs after 100,000 time steps; medium if the failure occurs between time steps 10,000-100,000; high if failure occurs between time steps 1,000-10,000; very high if the failure occurs between time steps 1-1,000. Thus, the results (Table 13) indicate that the severity of an ‘incorrect software modification’ fault combined with a ‘tank leak’ fault (case 2) is very high since the system failure occurs at time step 19, while in case 1 the severity of the ‘tank leak’ fault alone is very low since the system failure occurs at time step 100105.

In ISFA, the fault propagation path identification is inductive. No *a priori* fault-propagation paths are defined; actually, it is an outcome of the simulation of faults. Existing fault-propagation analysis tools such as TEAMS (QSI Tool), SymCure (Kapadia, 2003), and the HFPG (Mosterman & Biswas, 1999), require designers to explicitly formulate a fault-propagation model by specifying paths of causal relationships. In contrast, ISFA only uses information available during the design stage to determine potential failures and their propagation paths. Further, this propagation is identified through component behavioral simulation rather than functional dependencies (Kruse & Grantham 2009).
Because software faults give rise to unexpected failures, the addition of software control increases the nonlinearity of the system. ISFA captures various nonlinear aspects of fault propagation. It is simplistic and often incorrect to assume that faults propagate by following the functional or structural connectivity of a system. For example, a “leak” in the tank should not impose any fault propagation to its neighboring components and functions. Similarly, the Store Pos software fault should only affect the component it controls. However, we see nonlinear behavior: the software failure does not immediately affect the physical system but as the fault persists, the Regulate liquid function is lost leading to total system failure. However, these two functions are (1) unconnected to the valve and software control and (2) not on the downstream path in the function model. With ISFA, a proper mapping between the system behavior, its physical state, and the system functions will enable the identification of these nontrivial, nonlinear, fault-propagation paths.

An additional feature of ISFA is its ability to identify functional failures that result from global component interactions, masked fault activation, and timing faults. In Case 2, the tank leak fault was initially active for some time but the simulation indicated that the software was able to maintain normal operation for a few time steps, thereby masking the leak fault. However, over a period of time the transaction frequency activated a valve failure, resulting in the loss of Store fluid, Supply fluid, and Regulate liquid. Therefore, even though the transactions occur normally, their timing and frequency can potentially lead to system failure.
The case study also demonstrated that the simulation can be performed directly on a high-level design without any implementation level details or model transformation. Different components, functions, and communication models can be inserted into the design and analyzed to develop an optimum design early in the design phase. The analysis is qualitative but powerful enough to identify areas of potential failures. Such failures would typically remain unnoticed in the early design phase only to be discovered later in the development process. At such a point, significant resources would have been committed, subsystems would have been fully defined and assembled, and levels of detail would have escalated precluding exhaustive analysis.

Fault propagation depends on the function characteristic called flat parts. Within the flat part, any active fault will not propagate to the output. Integrating the input over all the flat parts gives the non-propagation probability of a fault. The flat parts themselves will be generated or killed depending on the operation performed on the input by the function being considered. For basic and advanced operators, we define rules for flat parts generation and preservation. These rules are a simple yet powerful way to calculate flat parts and hence the fault propagation probability.

Among the advanced operators, function composition is of special interest for software applications. The complex software execution can be represented as control-flow, which can be seen as independent chain of functions composed with each other. Control-flow analysis is a commonly used static analysis technique for identification of execution paths, structural faults, and optimization of the software structure. The control-flow concept of reachability can be easily extended to focus only on the faulty variables.
and thus optimize the FP calculation. Work is in progress in this regard. In addition, research to extend FP calculation for multiple variables is required. This research is a starting point for future multi-variable theories.

The proposed fault propagation analysis approach can be applied to component-based systems. A component is composed of several functions interacting with each other. Thus a component’s FP will be a combinations of the rules described in chapter 6. An algorithm to determine the flat parts of a component is a part of future research. It is interesting to study component-based architectures since the existing component-based reliability analysis approaches can be combined with the proposed functional approach and the reliability of the system can be determined more accurately.

An argument can be made for identifying flat parts using the condition \( \frac{d}{dx} = 0 \) rather than set-based approach. Although this is true, the functions and chain of function can be complex and computational intensive. Additionally, expressing functions that contain predicates mathematically and solving such equations analytically becomes quickly impractical, as seen in the design optimization problems. Numerical techniques are often used in such cases.
References


174
<table>
<thead>
<tr>
<th>Reference</th>
<th>Title and Abstract</th>
</tr>
</thead>
<tbody>
<tr>
<td>Tumer et al., 2011</td>
<td>Tumer, Irem Y., and Carol S. Smidts. &quot;Integrated design-stage failure analysis of software-driven hardware systems.&quot; Computers, IEEE Transactions on 60.8</td>
</tr>
</tbody>
</table>


### Appendix - Table and Proofs

<table>
<thead>
<tr>
<th>Package</th>
<th>Description</th>
<th>Elements used</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>FunctionModel</td>
<td>Depicts a high-level functional description of the physical system</td>
<td>HW_Function</td>
<td>An intended function subjected to the following constraint:</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td><strong>Context:</strong> FFIP:HW Function</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td><strong>Inv:</strong> self.host → forAll (n:HW_Component</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td><strong>Inv:</strong> (self.inflow = n.inflow and self.outflow = n.outflow)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Flow</td>
<td>An entity modified by a function and passed between connecting functions</td>
</tr>
<tr>
<td></td>
<td></td>
<td>FunctionLibrary</td>
<td>Library of functions types</td>
</tr>
<tr>
<td></td>
<td></td>
<td>FlowLibrary</td>
<td>Library of flow type</td>
</tr>
<tr>
<td>Configuration</td>
<td>Depicts the component structure of the physical system</td>
<td>HW_Component</td>
<td>A high-level component type</td>
</tr>
<tr>
<td>Flow Graph</td>
<td></td>
<td>Flow</td>
<td>An entity passed between connecting components</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Variable</td>
<td>A parameter of the flow such as Temperature</td>
</tr>
<tr>
<td></td>
<td></td>
<td>ComponentLibrary</td>
<td>Library of component types</td>
</tr>
<tr>
<td></td>
<td></td>
<td>FlowLibrary</td>
<td>Library of Flow types (same as function)</td>
</tr>
<tr>
<td>BehaviorModel</td>
<td>Defines the behavior of each component in terms of its input–output relationship</td>
<td>BehavioralRules</td>
<td>A description of a single component nominal and faulty behavior based on the input/output variables</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Nominal</td>
<td>One or more intended operating states of a component</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Faulty</td>
<td>One or more failure modes of a component</td>
</tr>
<tr>
<td></td>
<td></td>
<td>FFL</td>
<td>Rules relating flow changes (caused by component behavior) to a function’s state</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Transition</td>
<td>Change from one state to another caused by an event</td>
</tr>
</tbody>
</table>

*Table 14 FFIP Modeling Elements*
<table>
<thead>
<tr>
<th>Component</th>
<th>Inputs</th>
<th>Outputs</th>
<th>Behavioral Rules</th>
<th>Function Failure Logic (FFL)</th>
</tr>
</thead>
</table>
| Configuration Manager | Conversion-Data         | Conversion-Data        | Mode = Nominal  
IF ConversionData ≠ {Null}  
Mode = Faulty1  
IF ConversionData = {Null}                                                   | IF mode = Nominal  
Then Configure system = O  
IF mode = Faulty1  
Then Configure system = L |
| Sensor             | Pressure signal, Position signal | P, Pos, Level         | Mode = Nom1  
IF Pressure ≠ NULL  
Mode = Nom2  
IF Pos ≠ NULL  
Mode = Nom 3  
IF Level ≠ NULL  
Mode = Faulty 1  
IF Pressure = NULL  
Mode = Faulty 3  
IF Pos = NULL  
Mode = Faulty 3  
IF Level = NULL | IF mode = Nom 1  
Then Read Pressure = O  
IF mode = Nom 2  
Then Read Position = O  
IF mode = Nom 3  
Then Calculate Level = O  
IF mode = Faulty 1  
Then Read Pressure = L  
IF mode = Faulty 2  
Then Read Position = L  
IF mode = Faulty 3  
Then Calculate Level = L |

Table 15 Behavioral rules and Function Failure Logic (FFL)  
(Extracted and modified from [Mutha et al. 2012]. FFL must be read from top to bottom for each component).  
O = Operational; L = Lost; U = Unknown  
Continued.
Table 15 continued.

<table>
<thead>
<tr>
<th>Valve Controller</th>
<th>P, Pos, Level</th>
<th>Control-Command</th>
</tr>
</thead>
<tbody>
<tr>
<td>Mode = Nom 1</td>
<td>IF Level ε Lvalid &amp; Pos = {NA, Open} AND Control command ≠ {NA, Close}</td>
<td></td>
</tr>
<tr>
<td></td>
<td>ElseIf Level ε Lvalid AND Pos = {NA, Close} AND ControlCommand = {NA, Open}</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Mode = Nom 2</td>
<td></td>
</tr>
<tr>
<td></td>
<td>IF Level&lt; L_L AND Pos = {NA, NA} AND Control Command ≠ {NA, Open}</td>
<td></td>
</tr>
<tr>
<td></td>
<td>ElseIf Level&lt; L_L AND Pos = {NA, NA} AND Control Command ≠ {Close, NA}</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Mode = Nom 3</td>
<td></td>
</tr>
<tr>
<td></td>
<td>IF Level &gt; L_u AND Pos = {NA, NA} AND Control Command ≠ {Close, NA}</td>
<td></td>
</tr>
<tr>
<td></td>
<td>ElseIf Level ε Lvalid AND Pos = {NA, Close} AND Control Command ≠ {NA, Open}</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Mode = Faulty 1</td>
<td></td>
</tr>
<tr>
<td></td>
<td>IF Level ε Lvalid AND Pos = {Open, Open} AND Control Command = {NA, Close}</td>
<td></td>
</tr>
<tr>
<td></td>
<td>ElseIf Level ε Lvalid AND Pos = {NA, Close} AND Control Command ≠ {NA, Open}</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Mode = Faulty 2</td>
<td></td>
</tr>
<tr>
<td></td>
<td>IF Level&lt; L_L AND Pos = {NA, NA} AND Control Command ≠ {NA, Open}</td>
<td></td>
</tr>
<tr>
<td></td>
<td>ElseIf Level&lt; L_L AND Pos = {NA, NA} AND Control Command ≠ {Close, NA}</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Mode = Faulty 3</td>
<td></td>
</tr>
<tr>
<td></td>
<td>IF Level &gt; L_u AND Pos = {NA, NA} AND Control Command = {NA, Close}</td>
<td></td>
</tr>
<tr>
<td></td>
<td>ElseIf Level &gt; L_u AND Pos = {NA, NA} AND Control Command ≠ {Close, NA}</td>
<td></td>
</tr>
</tbody>
</table>

**Value control logic**

- IF mode = Nom1 OR Nom2 OR Nom3 Then Valve control logic = O
- IF mode = Faulty1 OR Faulty2 OR Faulty3 Then Valve control logic = L
- Else Valve control logic = U
<table>
<thead>
<tr>
<th>Component</th>
<th>Inputs</th>
<th>Outputs</th>
<th>Behavioral Rules</th>
<th>Function Failure Logic (FFL)</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Mechanical components</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Holdup tank</td>
<td>Q&lt;sub&gt;in&lt;/sub&gt;</td>
<td>Q&lt;sub&gt;out&lt;/sub&gt;</td>
<td>Mode = Nominal IF Q&lt;sub&gt;out&lt;/sub&gt; = Q&lt;sub&gt;in&lt;/sub&gt; Mode = Dry-out IF P &lt; P&lt;sub&gt;LTh&lt;/sub&gt; Mode = Overflow IF P &gt; P&lt;sub&gt;UTh&lt;/sub&gt;</td>
<td>IF mode = Nominal Then Supply fluid = O Store fluid = O IF mode = Dry-out OR Overflow Then Store fluid = L</td>
</tr>
<tr>
<td>Pressure Sensor</td>
<td>P&lt;sub&gt;in&lt;/sub&gt;</td>
<td>P&lt;sub&gt;out&lt;/sub&gt;</td>
<td>Mode = Nominal IF P&lt;sub&gt;out&lt;/sub&gt; ≠ Null Mode = Faulty1 IF P&lt;sub&gt;out&lt;/sub&gt; = Null</td>
<td>IF mode = Nominal Then Measure Pressure = O IF mode = Faulty1 Then Measure Pressure = L</td>
</tr>
<tr>
<td>Position Sensor1, Position Sensor2</td>
<td>Pos</td>
<td>Pos</td>
<td>Mode = Nominal IF Pos ≠ Null Mode = Faulty1 IF Pos = Null</td>
<td>IF mode = Nominal Then Measure Position = O IF mode = Faulty1 Then Measure position = L</td>
</tr>
<tr>
<td>Pipe1, Pipe2, Pipe3, Pipe4</td>
<td>Q&lt;sub&gt;j&lt;/sub&gt;&lt;sup&gt;i&lt;/sup&gt; in j = pipe index</td>
<td>Q&lt;sub&gt;j&lt;/sub&gt;&lt;sup&gt;i&lt;/sup&gt; out j = pipe index</td>
<td>Mode = Nominal IF Q&lt;sub&gt;j&lt;/sub&gt;&lt;sup&gt;i&lt;/sup&gt; out = Q&lt;sub&gt;j&lt;/sub&gt;&lt;sup&gt;i&lt;/sup&gt; in Mode = Clogged/Leak IF Q&lt;sub&gt;j&lt;/sub&gt;&lt;sup&gt;i&lt;/sup&gt; out &lt; Q&lt;sub&gt;j&lt;/sub&gt;&lt;sup&gt;i&lt;/sup&gt; in Mode = Burst IF Q&lt;sub&gt;j&lt;/sub&gt;&lt;sup&gt;i&lt;/sup&gt; out = zero</td>
<td>IF mode = Nominal Then Transfer fluid = O IF mode = Clogged/Leak Then Transfer fluid = D IF mode = Burst Then Transfer fluid = L</td>
</tr>
<tr>
<td>Inlet valve (iv)</td>
<td>Q&lt;sub&gt;v&lt;/sub&gt;&lt;sup&gt;i&lt;/sup&gt; in &lt;&lt;signal&gt;&gt;T4 &lt;&lt;signal&gt;&gt;T5</td>
<td>Q&lt;sub&gt;v&lt;/sub&gt;&lt;sup&gt;i&lt;/sup&gt; out</td>
<td>Mode = Nominal ON IF Q&lt;sub&gt;v&lt;/sub&gt;&lt;sup&gt;i&lt;/sup&gt; out = Q&lt;sub&gt;v&lt;/sub&gt;&lt;sup&gt;i&lt;/sup&gt; in Mode = Nominal OFF IF Q&lt;sub&gt;v&lt;/sub&gt;&lt;sup&gt;i&lt;/sup&gt; out = zero Mode = Failed open IF (Q&lt;sub&gt;v&lt;/sub&gt;&lt;sup&gt;i&lt;/sup&gt; in ≠ zero AND Q&lt;sub&gt;v&lt;/sub&gt;&lt;sup&gt;i&lt;/sup&gt; out = zero) Mode = Failed close IF (Q&lt;sub&gt;v&lt;/sub&gt;&lt;sup&gt;i&lt;/sup&gt; out ≠ zero) Mode = Faulty1 IF Q&lt;sub&gt;v&lt;/sub&gt;&lt;sup&gt;i&lt;/sup&gt; out &lt; Q&lt;sub&gt;v&lt;/sub&gt;&lt;sup&gt;i&lt;/sup&gt; in</td>
<td>IF mode = Nominal ON or Nominal OFF Then Regulate fluid = O IF mode = Failed open OR Failed close Then Regulate fluid = L IF mode = Faulty1 Then Regulate fluid = D</td>
</tr>
<tr>
<td>Outlet valve (ov)</td>
<td>Q&lt;sub&gt;v&lt;/sub&gt;&lt;sup&gt;o&lt;/sup&gt; in &lt;&lt;signal&gt;&gt;T2 &lt;&lt;signal&gt;&gt;T3</td>
<td>Q&lt;sub&gt;v&lt;/sub&gt;&lt;sup&gt;o&lt;/sup&gt; out</td>
<td>Same as inlet valve</td>
<td>Same as inlet valve</td>
</tr>
</tbody>
</table>

Table 16 Behavioral rules and Function Failure Logic (FFL). O=Operational; L=Lost; U=Unknown.
Table 16 continued.

<table>
<thead>
<tr>
<th><strong>Software Components</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Configurati</strong></td>
</tr>
<tr>
<td>on Manager</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td><strong>Sensor</strong></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
</tbody>
</table>

Continued.
Table 16 continued.

<table>
<thead>
<tr>
<th>Valve Controller</th>
<th>Pos</th>
<th>Level</th>
<th>Control-Command</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>&lt;&lt;signal&gt;&gt;T2</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>&lt;&lt;signal&gt;&gt;T3</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>&lt;&lt;signal&gt;&gt;T4</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>&lt;&lt;signal&gt;&gt;T5</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Mode = Nom 1
IF Level ε Lvalid & Pos = {NA, Open} AND Control command ≠ {NA, Close}
Elseif Level ε Lvalid AND Pos = {NA, Close} AND ControlCommand = {NA, Open}
Mode = Nom 2
IF Level < L1 AND Pos = {NA, NA} AND Control Command ≠ {NA, Close}
Elseif Level < L1 AND Pos = {NA, NA} AND Control Command ≠ {Close, NA}
Mode = Nom 3
IF Level > L3 AND Pos = {NA, NA} AND Control Command ≠ {NA, Open}
Elseif Level > L3 AND Pos = {NA, NA} AND Control Command ≠ {Close, NA}
Mode = Faulty 1
IF Level ε Lvalid AND Pos = {Open, Open} AND Control Command = {NA, Close}
Elseif Level ε Lvalid AND Pos = {NA, Close} AND Control Command ≠ {NA, Open}
Mode = Faulty 2
IF Level < L4 AND Pos = {NA, NA} AND Control Command ≠ {NA, Open}
Elseif Level < L4 AND Pos = {NA, NA} & Control Command ≠ {Close, NA}
Mode = Faulty 3
IF Level > L5 AND Pos = {NA, NA} AND Control Command = {NA, Close}
Elseif Level > L5 AND Pos = {NA, NA} AND Control Command ≠ {Close, NA}
Mode = Faulty 3

IF mode = Nom1 OR Nom2 or Nom3
Then Valve control logic = O
IF mode = Faulty1 OR Faulty2 OR Faulty3
Then Valve control logic = L
Else Valve control logic = U
<table>
<thead>
<tr>
<th>Activity/Action</th>
<th>Valid trace</th>
</tr>
</thead>
<tbody>
<tr>
<td>Start (Initial trace)</td>
<td>Nil</td>
</tr>
<tr>
<td>Configure system</td>
<td>&lt;?getdata, !setdata, ?setdata&gt;</td>
</tr>
<tr>
<td>Read pressure</td>
<td>&lt;?getdata, !convert, ?convert&gt;</td>
</tr>
<tr>
<td>Calculate level</td>
<td>&lt;!calcLevel, ?calcLevel&gt;</td>
</tr>
<tr>
<td>Read position</td>
<td>&lt;?getdata, !convert, ?convert&gt;</td>
</tr>
<tr>
<td>Store Pos</td>
<td>&lt;!getdata, ?getdata, !store, ?store&gt;</td>
</tr>
<tr>
<td>Valve control logic</td>
<td>&lt;!getdata, ?getdata, !controlLogic, ?controlLogic&gt;</td>
</tr>
<tr>
<td>Open inlet valve</td>
<td>&lt;!getdata, ?getdata, !open, ?open&gt;</td>
</tr>
<tr>
<td>Close inlet valve</td>
<td>&lt;!getdata, ?getdata, !close, ?close&gt;</td>
</tr>
<tr>
<td>Open outlet valve</td>
<td>&lt;!getdata, ?getdata, !open, ?open&gt;</td>
</tr>
<tr>
<td>Close outlet valve</td>
<td>&lt;!getdata, ?getdata, !close, ?close&gt;</td>
</tr>
</tbody>
</table>

Table 17 Activity and corresponding trace specification
<table>
<thead>
<tr>
<th>Symbols</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><img src="image" alt="Process" /></td>
<td><strong>Process</strong>: Represents the execution of a part of the design specification. This special process symbol is used to capture possible design specification execution failures in the “No” path and to report them. The symbol <code>⊕</code> stands for the following failures modes: 1. Incomplete design specification 2. Design models do not conform to respective metamodel, e.g., undefined component-function mapping.</td>
</tr>
<tr>
<td><img src="image" alt="Comment Box" /></td>
<td><strong>Comment Box</strong>: Represents the information provided by the execution of previous process</td>
</tr>
<tr>
<td><img src="image" alt="Initiating Event" /></td>
<td><strong>Initiating Event</strong>: First event in the ESD that initiates a sequence</td>
</tr>
<tr>
<td><img src="image" alt="End State" /></td>
<td><strong>End State</strong>: Terminating point of an ESD scenario</td>
</tr>
<tr>
<td><img src="image" alt="Output OR gate" /></td>
<td><strong>Output OR gate</strong>: Models multiple mutually exclusive outcomes. This gate has one input and multiple outputs</td>
</tr>
<tr>
<td><img src="image" alt="Input OR gate" /></td>
<td><strong>Input OR gate</strong>: Models the selection of one of the multiple inputs that leads to a common process. This gate has multiple inputs and a single output.</td>
</tr>
<tr>
<td><img src="image" alt="Output AND gate" /></td>
<td><strong>Output AND gate</strong>: Models multiple concurrent processes. This gate has one input and multiple outputs</td>
</tr>
<tr>
<td><img src="image" alt="Input AND gate" /></td>
<td><strong>Input AND gate</strong>: Models synchronization of processes. This gate has multiple inputs and a single output.</td>
</tr>
<tr>
<td><img src="image" alt="Multiple input/output AND gate" /></td>
<td><strong>Multiple input/output AND gate</strong>: Models synchronization of input processes as well as multiple concurrent output processes. This gate has multiple inputs and multiple outputs.</td>
</tr>
<tr>
<td><img src="image" alt="Condition" /></td>
<td><strong>Condition</strong>: Used to model a condition, which evaluates to yes “Y” or no “N”</td>
</tr>
</tbody>
</table>

Table 18 Event Sequence Diagram syntax and semantics
A. Non-propagation probability.

Let, \( x \) be the variable and \( p(x) \) be the expected operational profile of \( x \). Let \( f(x) \) be the function of \( x \); and \( f_{\text{faulty}}(x) \) and \( f_{\text{correct}}(x) \) be the faulty and correct versions respectively. Then the non-propagation probability is given by

\[
\text{npp} = \int_{-\infty}^{\infty} p(x) * \delta(f_{\text{faulty}}(x) - f_{\text{correct}}(x)) * d(x)
\]

For all values of \( x \) such that,

\[
f_{\text{faulty}}(x) - f_{\text{correct}}(x) = 0
\]

It implies the difference between the functions has a flat part. Hence we can modify the above equation, and integrate only over the FPs:

\[
\text{npp} = \sum_{i=1}^{\#FP} \int_{FP_i} p(x) \ d(x) = \int_{\forall FP} p(x) \ d(x)
\]

Since all the FPs are a subset of \( x \), the integration will always evaluate to less than or equal 1. Hence \( \text{npp} \) is always less than or equal to 1, thus preserving the probability property. If \( f(x) \) is a composition of “n” function, then \( f_{\text{faulty}}(x) \) and \( f_{\text{correct}}(x) \) will be the faulty and correct compositions respectively and the npp equation will remain the same as above.

B. Composition of functions

Consider the composition of two functions - \( f(g(x)) \). For function \( g(x) \), the domain is defined over \( x \); while for \( f(g(x)) \) the domain is defined over \( g(x) \). Let, \( y = g(x) \), then
\[
\text{Domain in } y \text{ of } f(g(x)) = \text{Domain } f(y) \bigcap \text{Range of } y
\]

\[
\text{Domain in } x \text{ of } f(g(x)) = g^{-1}\left(\text{Domain } f(y) \bigcap \text{Range of } y\right)
\]

Assume \(f(x)\) and \(g(x)\) are monotonic functions then,

\[
\text{Domain in } x \text{ of } f(g(x)) = g^{-1}\left(\text{Domain } f(y)\right) \bigcap g^{-1}(\text{Range of } y)
\]

\[
\text{Domain in } x \text{ of } f(g(x)) = g^{-1}(\text{Domain } f(y)) \bigcap \text{Domain } g(x)
\]