Posted: September 14, 2020

fmdtools: A Fault Propagation Toolkit for Resilience Assessment in Early Design

Daniel Hulse[1, *], Hannah Walsh[1], Andy Dong[1], Christopher Hoyle[1], Irem Tumer[1], Chetan Kulkarni[2], and Kai Goebel[3]

[1]Oregon State University
[2]KBR, Inc., NASA Ames Research Center
[3]Palo Alto Research Center
[*]Corresponding author: hulsed@oregonstate.edu

Abstract

Incorporating resilience in design is important for the long-term viability of complex engineered systems. Complex aerospace systems, for example, must ensure safety in the event of hazards resulting from part failures and external circumstances while maintaining efficient operations. Traditionally, mitigating hazards in early design has involved experts manually creating hazard analyses in a time-consuming process that hinders one's ability to compare designs. Furthermore, as opposed to reliability-based design, resilience-based design requires using models to determine the dynamic effects of faults to compare recovery schemes. Models also provide design opportunities, since models can be parameterized and optimized and because the resulting hazard analyses can be updated iteratively. While many analysis frameworks have been presented for early hazard assessment, these frameworks are difficult to apply without reference implementations, and most currently-available fault modelling tools are meant for the later stages of design. This paper describes fmdtools, a Python-based resilience-based design and analysis environment that solves these problems by enabling the designer to represent the system in the early design process, simulate the effects of faults, and quantify corresponding resilience metrics. This toolkit is then demonstrated in the hazard analysis and architecture design of a multi-rotor drone.

1 Introduction

Resilience, the ability to prevent and mitigate hazards, is a key consideration in the design of complex engineered systems (Cottam et al., 2019; Yodo and Wang, 2016a). In the aerospace industry, for example, it is important that aircraft adapt and recover from hazards (Choi et al., 2010), airports reconfigure runways in the event of damage (Faturechi et al., 2014), and supply chains that mitigate disruptions (Treuner et al., 2014). Because resilience factors heavily into the risk, safety, and functional reliability of a system, it is important to proactively consider resilience in the early design phase, when there is the most freedom to consider alternatives and allocate PHM features (Yodo and Wang, 2016b; Youn et al., 2011). It is often helpful to support this consideration with a cost-benefit analysis that can then be used to provide a coherent business case for PHM system development (Banks et al., 2009).

Considering hazards in early design involves representing the system in a high-level function structure to identify hazards and develop resulting design requirements (Stone et al., 2005). In the design of aircraft, this process is called functional hazard assessment and follows the ARP4761 guideline (ARP, 1996; Allenby and Kelly, 2001). Model-based functional hazard assessment has been an active research area (Krus and Lough, 2009; Kurtoglu and Tumer, 2008; Noh et al., 2011), with many new methods focusing on how to model different aspects of the system, such as human errors (Irshad et al., 2019), dynamic behaviors, new flows resulting from failures (Jensen et al., 2009), and operational decision-making (Short, 2016). However, there has been less demonstration of how to use this information to compare design alternatives, and the research codes underlying these methods have not been shared within the research community. To enable this resilience-based design, then, there is an opportunity to develop a tool that enables one to assess the resilience of a model without re-implementing underlying data structures and fault propagation methods.

The goal of the fmdtools project is to aid in the application of risk and resilience-based design frameworks

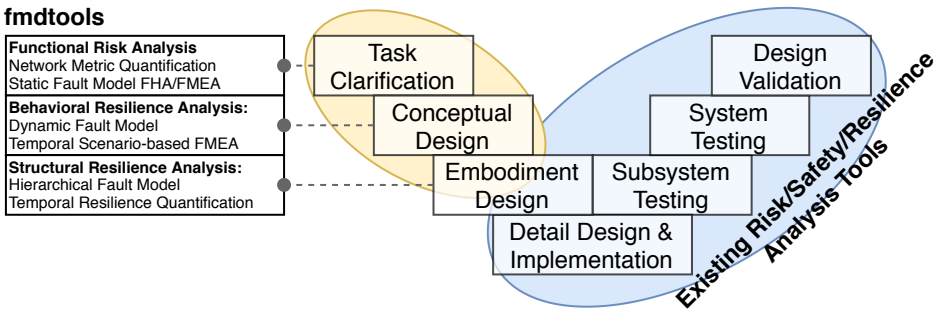

Figure 1: fmdtools is intended specifically to provide fault analysis methods that enable the consideration of risk in early conceptual design processes

by providing an open-source environment for modelling, simulation, and analysis of a new system. In doing so, it seeks to enable the practical application of early model-based functional hazard assessment frameworks while explicitly enabling the consideration of resilience. This toolkit has a number of potential applications to considering PHM in aerospace systems by supporting cost-benefit analysis (e.g., (Holzel et al., 2014)), which can used to allocate resources for PHM (Youn et al., 2011) and assess design alternatives (e.g. prevention or recovery schemes) (Hulse et al., 2019a). While most traditional risk assessment methods focus on how faults lead to system-level failures, the goal of the fmdtools project is to represent the full set of dynamic effects that result from a given fault scenario, so that one can compare the effect of different recovery actions. This requires advancing the modelling paradigm often used in early design.

For instance, one major difficulty in modelling failure propagation is representing the system in a way that captures the full set of effects which would be caused by a fault (Hönig et al., 2017). While a number of formalisms have been developed that enable this (e.g. modelica (Bunus et al., 2009), simulink/lustre (Joshi and Heimdahl, 2007), etc), these models are often computationally expensive and are more applicable for later design when one has the complete set of system behaviors. As a result, model-based functional hazard assessment methods are often defined directly in code in a base language (e.g. MATLAB/Python). In this setting, it is convenient to express the system using a *procedural*, *directed-graph* representation of system behaviors where each function is defined in a method and where the output flows of each function are used as inputs to the next function in the graph. The main problem with this representation is that it can only represent flow propagation that occurs in a single direction–from the source functions where flows originate to the sink functions where flows terminate. Defining a model in a base language also makes it difficult to then structure the model in a way that logically organizes faults, states, and behaviors of functions to be be easily understood, visualized, and modified. To improve on this approach, fmdtools uses a *object-oriented*, *undirected-graph* representation of system behaviors, where each function's states, behaviors, and faults are defined in a function class and a model class is used to connect adjacent functions so that behaviors can propagate in any direction through the model graph and one can easily parse the structure of the system model.

The rest of this paper is organized as follows: Section 2 discusses and compares fmdtools with existing fault propagation toolkits to show how it compares with the state-of-the-art as well as uses of fault injection in other fields. Section 3 the underlying model representation, fault propagation, and visualization algorithms provided in the toolkit. Section 4 provides an example of using fmdtools to model on a multirotor drone case study with increasing fidelity in the design process and using it to inform decision making. Finally, Section 5 concludes with some assessment of the usefulness of the tool and directions for future work.

2 Background

Fault propagation is widely used in a number of domains to assess the safety and resilience of a system of interest. As shown in Figure 1, while most current tools focus on modelling the system in the later

Toolkit	Behavioral/Causality Representation	Model Input/Platform	Availability	Use in design
HiP-Hops (Papadopoulos and McDermid, 1999)	Dynamic simulation with failure logic	Simulink, SimulationX, AADL, etc.	Commercial	Functional Hazard Assessment, Design Optimization (Papadopoulos et al., 2011)
Rodon (Bunus et al., 2009)	Behavioral constraint network with failure logic	Modelica-like Rodelica model	Commercial	Model-based engineering process (Lunde et al., 2006)
Modelica fault libraries (van der Linden, 2014; Minhas et al., 2014; Gundermann et al., 2019)	Undirected behavioral/failure logic	Modelica	Open Source	Design exploration (Lattmann et al., 2014)
OpenErrorPro (Morozov et al., 2019)	Probabilistic markov chain	Simulink/Stateflow, UML/SysML, and AADL	Open Source	Model-based reliable system design (Morozov et al., 2018)
SHyFTOO (Chiacchio et al., 2020)	Dynamic simulation with probabilistic hybrid fault tree	Simulink and toolkit-defined model	Open Source	Model-based design (Chiacchio et al., 2019)
OpenCossan (Patelli et al., 2018)	Probabalistic semi-markov transitions, external simulation	MATLAB toolkit-defined model	Open Source	Resilient, reliable, robust design under uncertainty (Patelli and Broggi, 2015)
IBFM (McIntire et al., 2016)	Bond graph with failure logic	Text-based model, Python	Open Source	Functional decomposition and design optimization (Hulse et al., 2019b)
fmdtools	Dynamic un-directed behavioral simulation with failure logic	Python toolkit-defined model	Open Source	Early resilience-based design, analysis, visualization and optimization

Table 1: Comparison of Model-based Fault Simulation Toolkits for Design

design stages and in the verification and validation process, when there are detailed models of the system, fmdtools is meant to support early design processes. To do this, it provides analyses that support each phase of the function-behavior-structure design process commonly used in engineering design (Howard et al., 2008). In this process, one creates a functional decomposition of the tasks the system is to perform, finds solution principles to achieve those tasks, and then synthesizes those principles into a realized design concept. These design processes are supported by static failure-logic functional hazard assessment and network models, dynamic behavioral models, and hierarchical fault models, respectively.

A number of modelling formalisms have been developed to assess the risk of hazards in engineered systems, including fault trees, bayesian networks, and stochastic petri nets (a type of discrete event simulation) (Chemweno et al., 2018). While the simpler methods (fault trees, bayesian networks) enable stronger proofs of system dependability (Chemweno et al., 2018), they do not express the system resilience–the behavior over time resulting from a fault. In these situations, a discrete event simulation or continuous dynamic model is used. Discrete event simulation (e.g. (Matloff, 2008)) has been used to assess and design resilience into systems, including maintenance (Wang et al., 2015) and recovery aspects (Miles, 2018). Similarly, Monte Carlo techniques are often used with continuous simulation models to quantify risk of hazardous states (Hu, 2005). While these approaches describe the underlying general approaches to simulating faulty behaviors in a system, the following sections describe specific toolkits that use these approaches to quantify risk and resilience.

2.1 Related Work

As shown in Table 1, there have been some prior efforts to develop generally-applicable toolkits for the design of resilience in a model-based engineering process. One major difference between toolkits is the way they represent causality in the system to determine the effects and propagation of faults, which has a number of potential aspects, including the types of failure paths able to be represented (through the system behavior, failure logic, state transitions or a hybrid) (Hönig et al., 2017), and the nature of causality (probabilistic or deterministic). To integrate with design activities, currently-available frameworks are built around either a standardized modelling language (e.g. AADL), or an existing systems modelling tool such as Simulink or Modelica. While this enables one to use a single unified model for multiple analyses through the process, it can also be limiting if certain aspects of fault propagation are difficult to represent in the given formalism. Finally, while each of these toolkits are claimed to support different design processes, the design being performed is later stage embodiment design–not the early conceptual design shown in Figure 1. The main exception to this is Hip-Hops (Hierarchically Performed Hazard Origin and Propagation Studies) and IBFM (Inherent Behaviors of Functional Models), which both target early design processes. As a result, the implementation of early functional risk assessment tools has been cited as a necessity to bridge the gap between model-based hazard assessment methods in literature and their adoption in industry (Grigoleit

et al., 2016).

Development of the fmdtools toolkit was initiated to address limitations with the previously-developed IBFM simulator (McIntire et al., 2016). While IBFM was developed for the early, high-level functional design of engineered systems and was able to determine the end-states of large sets of system faults, the behavioral representation was limited to finite states (Zero to Highest), a set of given possible operations on input and output flows, event-based dynamic propagation, and a given syntax for specifying failure logic. This limited its ability to express all possible behaviors that could occur in a system as a result of a given fault. Additionally, the text-based model representation made it difficult to parameterize models and simulate them iteratively in an optimization algorithm. In this work, these limitations are addressed by defining the model as a set of interacting Python classes with possible faults and (nominal or faulty) behaviors to simulate over a set of discrete time-steps. With this model representation, one can easily express and optimize the dynamic behavior of the system without being limited by expressiveness of the underlying modelling tool. fmdtools has additionally since expanded in scope to include not just fault propagation tools but the necessary analysis and visualization tools needed to interpret simulation results. In doing so, it comprises an early resilience-based design environment that enables quick, iterative analysis over a design model.

2.2 Other Fault Modelling Tools

In addition to the fault propagation toolkits mentioned above, there are additionally a number of toolkits used for fault propagation in safety assessment and for application-specific resilience assessment.

Safety assessment toolkits are used to verify that a given design meets desired safety and reliability criteria (Hönig et al., 2017; Joshi et al., 2006). Typically, these tools take a design model specified in a formal language (e.g. Simulink (Joshi and Heimdahl, 2005), Lustre (Joshi and Heimdahl, 2007), Modelica (van der Linden, 2014), AADL (Stewart et al., 2018), UML (Combemale et al., 2008)) and apply a model checker to the system description to find cases where the system does not work as intended. While this process can be performed in a nominal system model, model-based safety assessment tools extend this process to assess safety by including failure modes and faulty behaviors in the system description (Joshi and Heimdahl, 2005, 2007). However, because the intended purpose of these tools is to verify safety requirements post-design (see the right side of the v-model in Figure 1), they typically rely on detailed, fully-specified models of the design that are not available early in the design process (Grigoleit et al., 2016). Furthermore, these tools are not intended to assess resilience–the dynamic effects of failures due to faults–but instead assess the safety or reliability of the system (i.e. an overall fault tree or failure probability). As a result, they are not applicable to the design of resilience in the early design process when the system has not been fully specified and the designer is interested in assessing the system's dynamic response to faults.

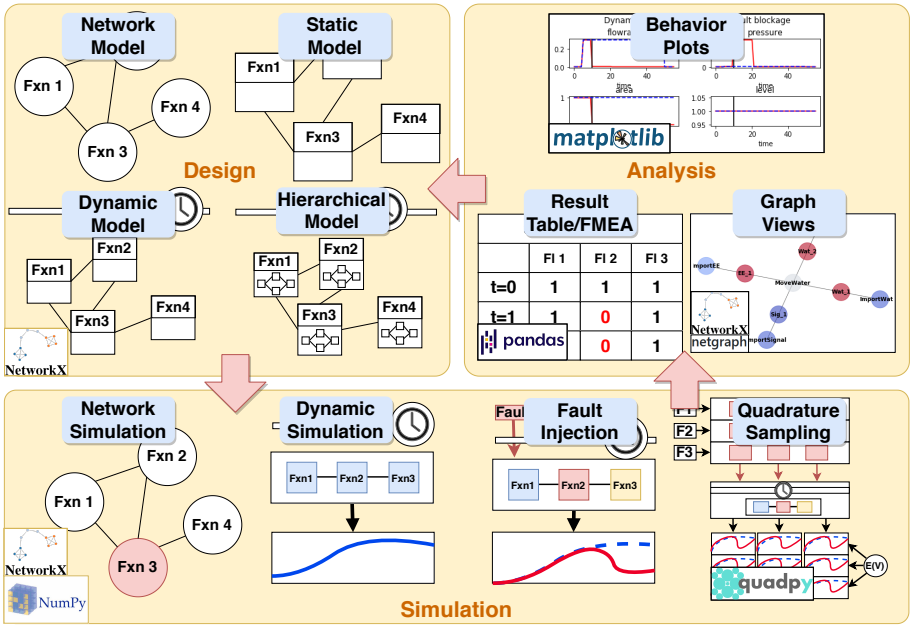

Figure 2: The fmdtools design, simulation, and analysis environment.

Fault injection is used widely in software and hardware engineering to assess a computer system's ability to manage the different types of faults. Existing simulators vary by domain (e.g. distributed systems (Martins et al., 2013), servers (Kollárová, 2014), stand-alone systems) and type of faults (hardware-originated (Georgakoudis et al., 2019; Schirmeier et al., 2015) or software components/algorithms (Goldstein et al., 2020; Arribas et al., 2018)), though generic simulators also exist (Winter et al., 2015). One of the advantages of software (as opposed to hardware/systems) engineering is that this testing can be iteratively performed on a running prototype of the system at a low computational cost, and as a result, these tools do not operate on models of the system as is needed in engineering design.

There are additionally a number of application-specific resilience assessment toolkits that interface with specific system simulators to model the resilience-related attributes for that domain. For example, integrated circuits have well-defined properties that have led to a number of specialized fault simulation algorithms, which have since been implemented in software (May and Stechele, 2012; Niermann et al., 1992). Infrastructure often has specific hazards to assess, which has resulted in tools to consider natural disasters for cities (Fraser et al., 2016; McKenna, 2011) and cyber-physical threats in smart grids (Wadhawan and Neuman, 2017). Finally, assessing the hazards of autonomous vehicles in a real system is both costly and hazardous (Gambi et al., 2019), which has led to the development of a number of simulators that enable one to try different policies to approaching hazards which the vehicle will encounter (Jha et al., 2018, 2019). While these toolkits can assess resilience in specific design contexts with high fidelity, they are not applicable to a generic systems design context.

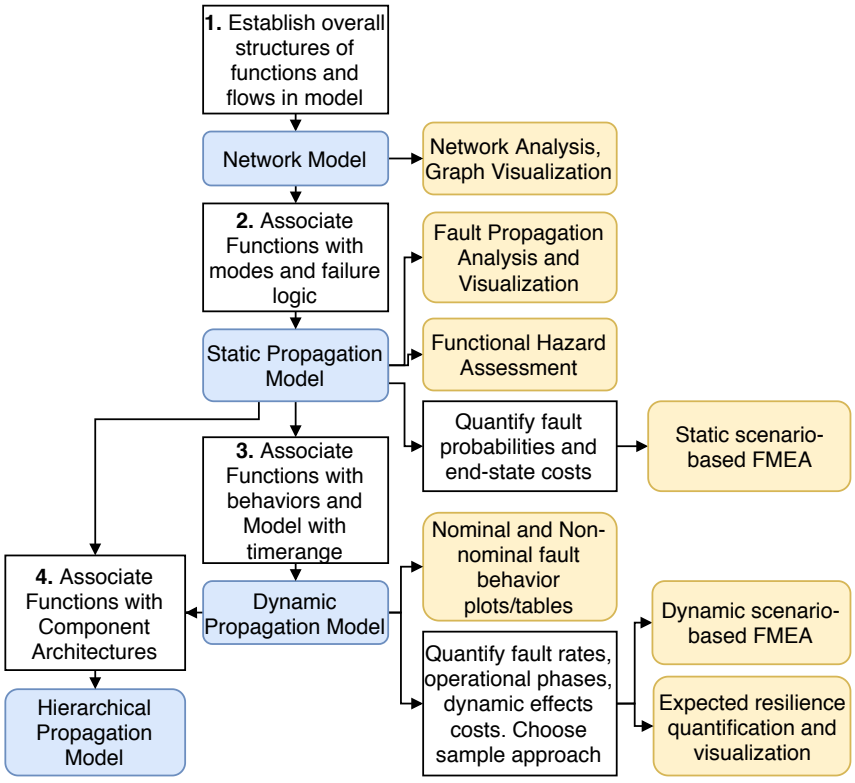

Figure 3: Fault model types and analyses enabled by fmdtools. Note that since model types build on each other, the analyses from less detailed model types (e.g. static propagation models, network models) can apply to more detailed model types (e.g. dynamic propagation models, hierarchical propagation models).

3 Methods and Algorithms

The fmdtools toolkit aims to provide a design, analysis, and simulation environment that enables the design of resilience into a system. To accomplish this, it provides a number of tools to represent, simulate, and analyze the system as it progresses through the design process, as shown in Figure 2. As a result, it can accommodate a number of modelling and analysis use-cases to progress from the early, abstract representations of the design (e.g. network and static propagation models) to more detailed representations of the

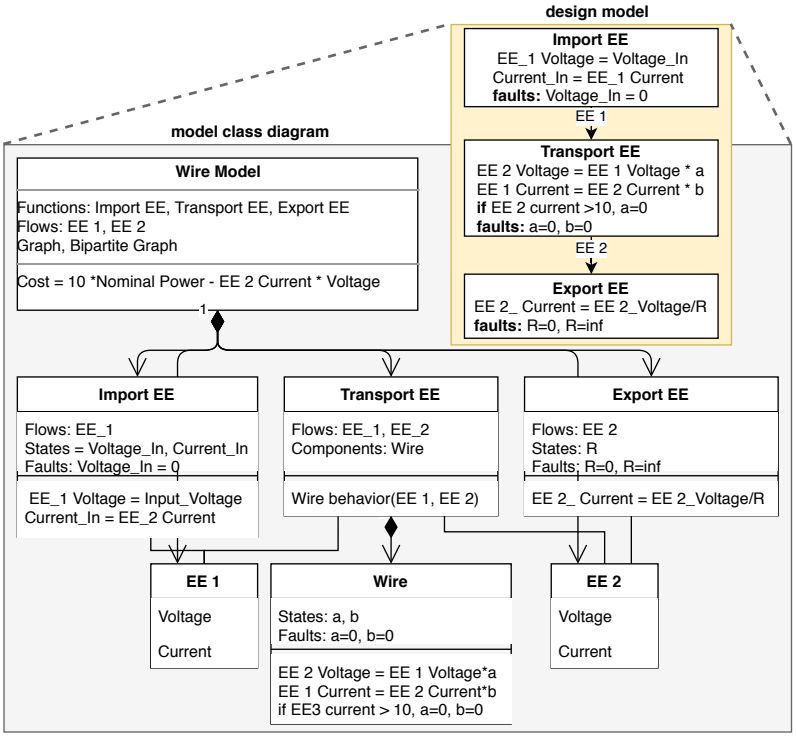

Figure 4: Example model representation. A Model is composed of function objects with internal states and faults, (optional) instantiating component objects and relationships to flow objects.

system structure (e.g. dynamic and hierarchical propagation models) and behaviors in the same modelling environment, as shown in Figure 3.

The full implementation of this work is provided in a publicly-available repository, along with examples and documentation at `github.com/DesignEngrLab/fmdtools` or (Hulse et al., 2020). While an exhaustive description of every method and class is out of the scope of this paper, this section will discuss some of the underlying concepts and structure of the toolkit. The fmdtools toolkit is organized into different modules used for model representation, simulation, and analysis. The `modeldef` module provides the classes to define a system model from functions, flows, and components as well as a fault sampling approach for resilience quantification. The `faultsim` module then provides methods for propagating faults in a model and quantify network metrics. Finally, the `resultdisp` module provides a number of methods to process and visualize simulations of the model, including behaviors over time, FMEA tables, fault graphs, and heatmaps.

3.1 Model Representation

In this work a system consists of functions (modules that perform a task), components (specific solutions to a function), and flows (variables) that are connected with each other in a bipartite graph. In a model (itself defined by a user-defined class), functions, flows, and components are represented by objects instantiated from user-defined classes that are connected by a graph, as shown in Figure 4 for a model of a wire. In this representation, each function (e.g., Transport EE) consists of its associated flow objects (e.g., EE 1, EE 2), internal state variables, set of faults, behavior methods, and constituent component objects (if a component representation is used). Flows are in turn defined by dictionaries of states with corresponding values and components are defined by internal state variables, a set of faults, and behavior methods. Model objects are then composed of their constituent functions, flows, and components as well as a graph object used to track the connections between functions and flows and a classification method used to quantify the costs of a fault scenario propagated in the model.

This model is constructed in fmdtools by defining a subclass that extends the `Model` class to represent a system of interest. This model class is constructed by defining the simulation parameters (e.g. units, timesteps, starting and ending simulation times, design parameters), adding each flow in the model, adding

each each function in the model and connecting them with flows to construct the model graph, and creating a classification method which determines the rate, cost, and expected cost of a given scenario given the results and parameters for that simulation–the end-state of functions and flows (e.g. values and faults present), the scenario properties (e.g. rates, faults, etc.), and the history of model states (values for flows over time). By instantiating this class, one can then use the resulting object to model the evolution of system states over time for a given set of defining parameters.

3.1.1 Functions

In design, functions represent the high-level tasks performed by the system (Pahl and Beitz, 2013). In fmdtools, the `FxnBlock` class is used to represent these functions, which constitute the main building block of the system model defining faults and system behaviors. To use this class, users define a subclass for each function which uses inherited `FxnBlock` attributes to represent the properties of the function. At its most basic, a user-defined function class is defined by the flows going in and out of the function, a behaviour method which relates values of input flows to values of output flows, and a set of faults which modify the input-output relationship defined in the behavior method. However, more attributes can be defined, including states (internal variables of the function tracked in the model history), conditional fault methods defining input/output flows or function states result in faults, timers that can be used to express delays in behaviors, and components, described in Section 3.1.3.

To enable expected resilience quantification, each fault to inject in the function can be associated with a probability model that expresses the likelihood of the fault occurring. This probability model is defined by a function-wide rate, the proportion (or rate/probability) of faults resulting from the mode, an opportunity vector expressing the relative likelihood of a fault occurring at a specific phase of operation, and a specification of whether these values are a rate (with units) or a probability. Each fault can additionally be given a cost of repair that can be used to calculate the costs of fault scenarios.

3.1.2 Flows

Flows represent the states (e.g. energy, material, or signal) with which the functions interact to achieve the overall goal of the system. In fmdtools, the `Flow` class is used to represent these states, which can either be extended in a user-defined subclass (if there are special properties the designer wishes to represent) or instantiated using a dictionary with the name of the flow, name of the values characterizing the state of the flow, and initial quantity for each value. Because of the undirected nature of the model graph and associative relationships between functions and flows, flows are accessible (i.e. values can be changed) in all connected functions.

3.1.3 Components

While functions represent the task a system performs, components can be used to represent realizations of that function. Often in risk or resilience-based design, one is interested in designing component architectures which will fulfill an overall function, even when an individual component fails (Youn et al., 2011). To represent this, fmdtools uses the `Component` class, which shares many attributes with the `Function` class, including internal states, a behavior method, and a set of fault modes. Similarly, to use the `Component` class in a model, one must define a subclass for the modelled component with its own states, fault modes, and behavior method. However, unlike the `FxnBlock` behavior method (which acts on `FxnBlock` attributes and does not return anything), `Component` behavior methods explicitly take the inputs of the component behavior as input and return outputs of the behavior as output. This is done to enable the designer to relate the inputs and outputs of the components in the architecture fulfilling a function with each other and function states in the behavior methods of its corresponding function.

3.2 Resilience Simulation

fmdtools has two main approaches for assessing the resilience of a model: quantifying network metrics of the system architecture and propogating faults in the system model. Network metric quantification, described in Section 3.2.1 and implemented in the `networks` submodule enables some consideration of the structural resilience of the system before specifying the fault logic in the system. Fault propagation, described in Sections 3.2.2 and 3.2.3 can then be used to assess the resilience of a model given the model has

faults to propagate in each function (or the function of interest) and the functions each have the necessary fault logic and/or behaviors.

3.2.1 Network Metric Quantification

The network metric quantification tools in `fmdtools` enable the early assessment of the failure tolerance and resilience of a design. Network analysis is an emerging early design methodology for predicting the likely failure tolerance of a design without the need for high fidelity models (Haley et al., 2016a; Mehrpouyan et al., 2013). In this approach, engineered systems are represented as networks in which nodes represent functions, parameters, or components depending on the specific network formalism. In short, network analysis enables analysis of the topology that emerges from the connectivity between system elements. The representation of connectivity between system elements is similar to that of a design structure matrix (DSM), and network theory enables visualization and powerful analyses of emergent network properties. Networks are used for both *a priori* assessment of resilience properties as well as for quantifying the network's response to simulated faults.

Resilience Properties of Networks Known relationships between structure and failure tolerance (Boccaletti et al., 2006) in complex networks enable the *a priori* assessment of a network's resilience properties prior to simulation. The structure of a network is intrinsically related to its functionality; structural vulnerabilities are therefore relatable to loss of functionality. In networks, failure tolerance is typically studied by attacking or removing nodes and measuring the resultant degradation of the network structure. That is, much as loss of functionality in one component in an engineered system leads to decreased overall performance, degradation or removal of one node in a network leads to an alteration of the network's topology. In this way, a network's resistance to failure is relatable to an engineered system's resistance to failure. Certain networks are more prone to degradation due to node removal than others. Likewise, certain nodes are more prone to causing degradation than others. In component networks, failure or removal of a component node implies loss of functionality in that component, and the analysis therefore relates to the effects of the failure of that function. Network analysis of the effects of function node failure is complementary to, for example, FMEA in that it enables an assessment of the effects of a failure and its severity.

A common method for characterizing a network is studying its degree distribution. In an undirected network, the degree of a node is its the number of connections (edges). Nodes with high degree (hubs) tend to be more critical in retaining a network's functionality. A network's degree distribution is a histogram of the degrees of all nodes in the network. Degree distributions that follow a bell-curve tend to be more vulnerable to random node removal, whereas degree distributions that follow a power law tend to be more vulnerable to targeted node removal (Barabási, 2009) (i.e. removal of a hub). High degree nodes are identifiable using `find_high_degree_nodes`. Degree distributions are provided using `degree_dist`.

The modularity and community structure of a network also have significant bearing on the network's failure tolerance. Modules, or communities, are tightly coupled groups of nodes. The modularity of a network, the degree to which a network exhibits a modular structure, is typically measured using Q-modularity (Newman, 2010). Nodes that connect modules – bridging nodes – are functionally important to a network's failure tolerance (Walsh et al., 2018). This is comparable to, for example, identifying high severity failures in FMEA. Networks with high modularity have been found to be less robust overall (Walsh et al., 2019). `find_bridging_nodes` is provided to identify bridging nodes in the network model. `calc_modularity` is provided to compute the modularity of the network model.

Average shortest path length (ASPL) is a measure of the efficiency of the spread of information through a network. Under attack, networks with low ASPL are more likely to retain short to moderate length paths between any given pair of nodes, whereas networks with high ASPL are more likely to disintegrate significantly under attack. ASPL is defined as the mean of the sum of all edge weights along the shortest path between each pair of nodes in a network and is available as `calc_aspl`.

Simulation-Based Analysis of Network Resilience Rather than using a network's structure to predict its response to failure, it is possible to use simulation-based approaches to network analysis. First, robustness coefficient simulates the effect of failure in the network. This approach, implemented as `calc_robustness_coefficient`, measures the changing size of the largest connected component of a network during successive node removal (Viana et al., 2009). A second simulation-based approach is an SFF (susceptible-failed-fixed) epidemic spreading model, which is able to explicitly represent node recovery (Mehrpouyan et al., 2013).

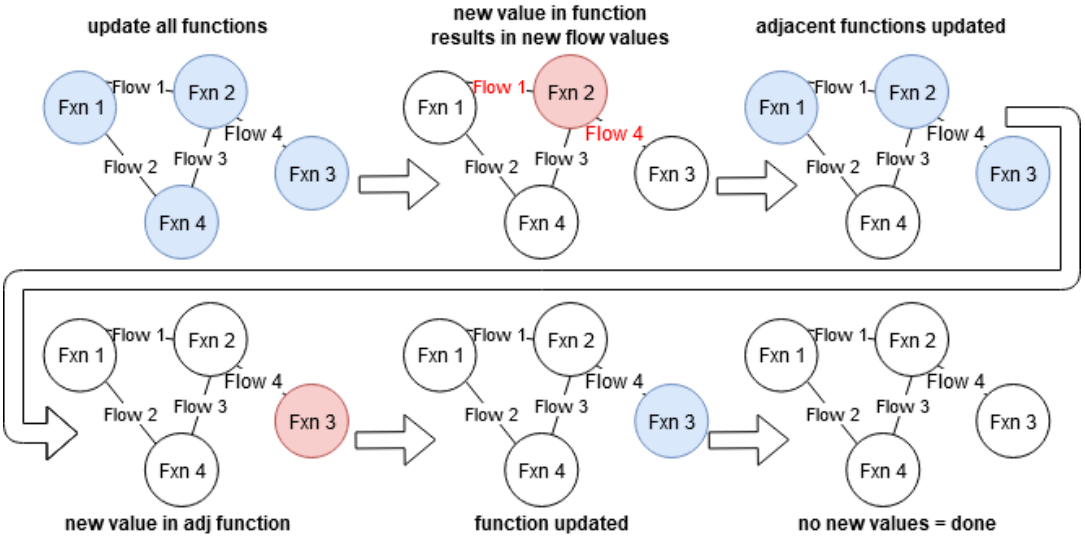

Figure 5: Illustration of static fault propagation. Function behavior methods are iteratively run in a list until the states of the system no longer change.

Rather than attacking or removing nodes as in the robustness coefficient, this model considers nodes to be in a susceptible, failed, or fixed state. Failed nodes may cause susceptible nodes to fail, similarly to infected persons spreading an epidemic. After a node is fixed, it is unable to fail again by the same cause (immunity). The SFF model is available as `sff_model`.

3.2.2 Fault Propagation

Propagation of faults in a model has two major aspects: static fault propagation and dynamic fault propagation. Static fault propagation is the process of determining the immediate effects of a fault in a system, as illustrated in Figure 5. First, all of the behavior methods are run. If a new value occurs in one of the functions (e.g., because of fault injection) or its associated flows, that function and functions adjacent to the changed flows are added to a list of functions to update. The behavior functions for those functions are then run and new functions to update are again added if they receive new input values. This process is run iteratively until the system reaches a stable end-state, if a stable end-state is possible. Thus, one necessary property for fault models is stable fault behavior–behaviors in one function should not change behaviors in other functions that will in turn change the original behaviors in the original function repeatedly, indefinitely.

As shown in Figure 4, functions have associative relationship with flows, meaning functions have full access to (and can change the values of) the states of both “input” and “output” flows. Because the propagation of system states is undirected, functions have the ability to propagate new system states to any other functions in the graph–not just the function that would be placed “next” in the sequence of tasks to perform. However, because of the undirected system representation, conflicts between function behaviors can occur when different functions specify different values for the same flow state, resulting in a non-convergent system state at a given time-step. This must be avoided in model setup, which can be achieved by representing flows with states that propagate forward through the model graph (i.e. “effort” variables such as voltage or potential in a bond graph representation) and states that propagate backwards through the model graph (i.e. “flow” variables such as current or rate).

Dynamic fault propagation is the evolution of states in the model over time necessary to quantify resilience as a time-dependent property of a system. The implementation of dynamic fault propagation used here is illustrated in Figure 6. As shown, the static propagation procedure is iteratively run over a set of time-steps from fault injection time to the end of the simulation time. To fully assess resilience, a history is kept of all of the states of the model (flow values, function state values, faults in functions, etc.) over the set of time-steps. Based on a simulation like this, one can then quantify metrics for the simulation such as dynamic costs, recovery time, or worst state over time.

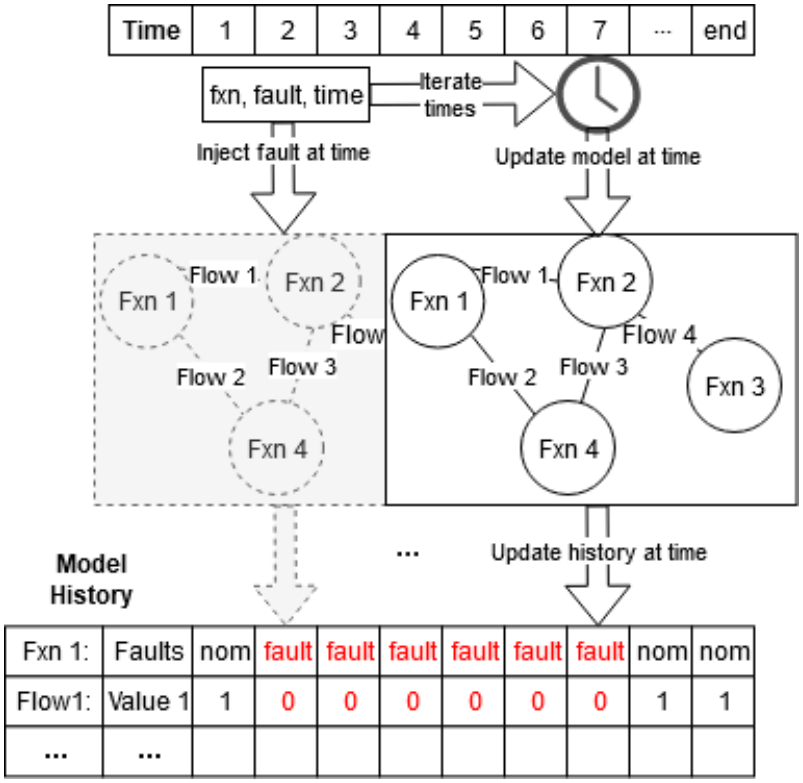

Figure 6: Dynamic fault propagation. A model is iteratively updated at each discrete time-step from fault injection to the end of simulation.

3.2.3 Fault Injection

Depending on the scope of the analysis, one might be interested in injecting faults in different ways. Before injecting faults, it is important to determine that the model performs as expected by simulating the system without any faults. Then, while setting up faults and fault behaviors (and in systems with single faults), one can propagate a single fault at a given simulation time to verify that the simulated behavior matches the expected behavior for that fault. Once faults are encoded, the list of faults can be propagated in the system at times defined in the model. While this approach lets one see the consequences of faults injected at set times, it may not be for calculating expected resilience metrics, since it neglects joint fault scenarios and when in time faults are most probable.

To quantify the mathematical expectation of fault-injection based resilience models, the `SampleApproach` class can be used to define the set of fault scenarios to propagate in the model, as illustrated in Figure 7. This class uses the dynamic probability model set up in the model, functions, and components, along with user-defined parameters to create a list of fault scenarios which will be used to represent the statistical expectation of the defined faults. This approach can be defined over the set of faults to include, the number of joint faults to inject and the probability model for the joint faults (e.g. assuming independence or a conditional probability), and the times to inject the faults at. The set of injection times is determined by two main parameters: the phases defined in the model (and opportunity vectors for each fault in the probability model), and the set of times within each phase. Within each phase, these times can be specified as every discrete timestep, an evenly-spaced approach with a set number of points, a randomly-spaced approach with a set number of points, or an approach using a given quadrature defined in the `quadpy` software package (Schlömer et al., 2020). Additionally, Sample Approaches can be refined post-hoc based on a set of simulation results to represent the behaviors with a small set of sample points. Using these approaches, one quantify expected metrics iteratively with as few fault simulations as possible.

3.3 Resilience Analysis and Visualization

Using the models defined in Section 3.1 and simulations in Section 3.2, one can then perform analyses on the results. fmdtools provides a number of different methods using existing Python libraries, including mat-

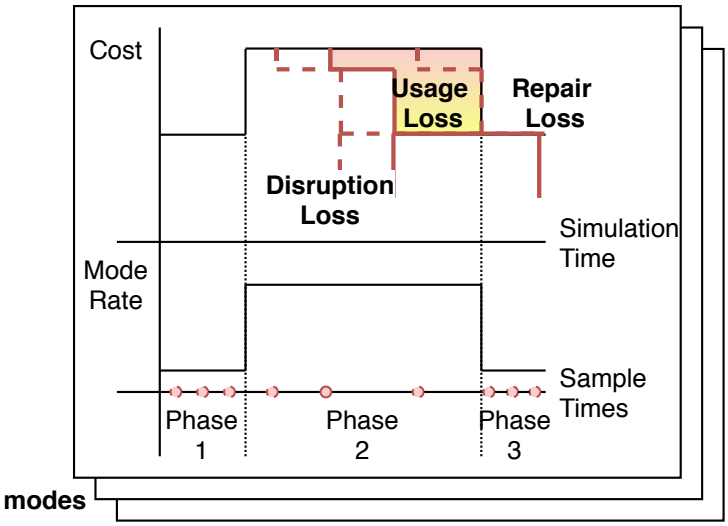

Figure 7: Injecting faults according to a fault sampling approach.

plotlib (Hunter, 2007), networkx (Hagberg et al., 2008), and pandas (pandas development team, 2020) to makes sense of the fault behaviors modelled in the system. To assist with this analysis and visualization, the results of the simulations are processed to summarize the state of different aspects of the system as nominal or faulty. This process results in three categorizations for functions, flows, and components: nominal, when the data structure behaves as it does in the nominal state; degraded, when the data structure has a different behavior than it does in the nominal state; and faulty, when a component or function has a fault. This approach to result processing enables high-level visualization of the status of model structures without the user defining bounds or conditions for each variable to be listed as faulty or degraded. Using this representation, one can make a number of plots of the model graph structure, system behaviors over time, and tabular summaries of results.

3.3.1 Graph Plots

To visualize the propagation of faults in the system, fmdtools provides a number of methods that display a graph view of the system using `matplotlib` (Hunter, 2007), `networkx` (Hagberg et al., 2008), and `netgraph` in the `graph` sub-module. Graph views of the system enable one to see the structure of the model as well as desired states or properties of the functions and flows. Two main graph representations can be plotted: a default graph representation where the flows are plotted as edges between function (which are nodes) and a bipartite graph representation where both functions and flows are nodes and edges are the associative relationships between them. To visualize the model graph in an intuitive layout, methods calling `netgraph`'s `InteractiveGraph` are provided which enable one to place nodes manually (rather than relying on an algorithmic layout). While the default representation is more intuitive to interpret–especially for simple systems–the bipartite representation often makes a better use of space–especially when a flow is connected to more than two functions–because there is less more freedom to ensure that edges do not overlap. Using the graph view, one can then visualize graph metrics as shown in Figure 8, the state of the model at the end-state or a particular time (or set of times) in the model history as shown in Figure 11, and various model run statistics defined by heatmaps (e.g. expected degradation time, maximum number of faults, etc.). These visualizations give one a view of how faults and behaviors propagate at the system level.

3.3.2 Dynamic plots

When modelling a dynamic system, it is often important and necessary to plot particular states over time in order to see how the behavior evolves over time. Methods in the `plot` sub-module use `matplotlib` (Hunter, 2007) to show the evolution of chosen states of the model over time, with (if the simulation was a run of a fault scenario) faulty states overlaid over the nominal states over time to aid assessment of the fault-induced behavior, as shown in Figure 12. In addition to modelling system behavior in a particular modelling scenario, time-based plots also have the ability to visualize the cost responses given by the

ASPL	Modularity	Robustness Coefficient
1.44	0.12	95.85

Table 2: Network metrics for default (function) network representation of example drone model.

simulations at each injection time. This plot, shown in Figure 13, can be used visualize how the sample approach defined in Section 3.2.3 approximates the expected resilience costs by showing the rate over time for a particular fault, as well as the modelled cost and quadrature weight for each sample point. Since these plots are plotted in `matplotlib`, well-known commands and interfaces can be used to edit and save the plots.

3.3.3 Tables

Finally, it is often helpful to be able to view the results of fault simulations in tabular form. While simulation results are typically returned as nested dictionaries, fmdtools provides methods to view this information as a `pandas` (pandas development team, 2020) table to enable results processing and display. Based on the processed results, one can also make tabular summaries of simulations, such as the number of functions and flows degraded over time or in a particular simulation. Tables are most helpful for summarizing the results of a set of simulations, where they can provide an FMEA-style assessment of the functions and flows affected as well as the rate, cost, and expected cost of each fault simulation, as shown in Table 3, which can be generated to delineate between or summarize fault effects over each phase. Since these tables are implemented in `pandas`, existing interfaces can then be used to display and/or save results (e.g. as a `.csv`).

4 Example - Drone Model

This section illustrates how one can use the fmdtools software package to aid the conceptual design of a real system by providing analyses that increase in fidelity and detail with the design process. A number of examples are provided in the repository, including a conceptual model of a pump, a dynamic modelling of virus spreading, a human-operated tank system, and a static model of an electric power system.

This example considers the design of a multi-rotor drone which must fly to a given location and return to its destination. The functional model of this system is shown in Figure 11, which includes the rotor lines, structure, electrical power source and distribution, and path planning of the system. In this example, we first quantify metrics about the system structure, then use a static representation to generate a high-level FMEA and visualize fault propagation, then create a dynamic version of the model to visualize fault behaviors over time and quantify the effects of injecting faults at different simulation times, and finally use a hierarchical model to compare the dynamic fault responses and resulting resilience of different component architectures.

4.1 Network Representation and Analysis

First, the model is analyzed using network metrics and algorithms. In `fmdtools`, it is possible to analyze various network representations of the model, although only one network representation will be shown in each step. Each network analysis function has options to analyze the default (function) network, the bipartite (function-flow) network, the parameter network, or the component network. The bipartite network is treated as a unipartite-like network for analysis (Haley et al., 2016b). Analysis of the various network perspectives provides a more complete understanding of the model's resistance to failure.

The network metrics for the function network are given in Table 2. Low (close to zero) modularity indicates a high degree of interconnectivity and has been correlated with high robustness (Walsh et al., 2019). High robustness coefficient and low ASPL indicate high robustness to random node failure. High degree nodes are highlighted in red in Fig. 8. Nodes with high degree, or hubs, are more likely to cause significant performance degradation if in a fault state. Based on this analysis of the system topology, we can conclude that the functions with the most opportunity to affect other functions at a topological level are path planning, control systems, and the structure of the drone, since these functions have the most connections with other subsystems. Identification of these vulnerabilities is similar to the identification of high severity failures within an FMEA. The degree distribution of the function network is presented in Fig. 9. The relatively

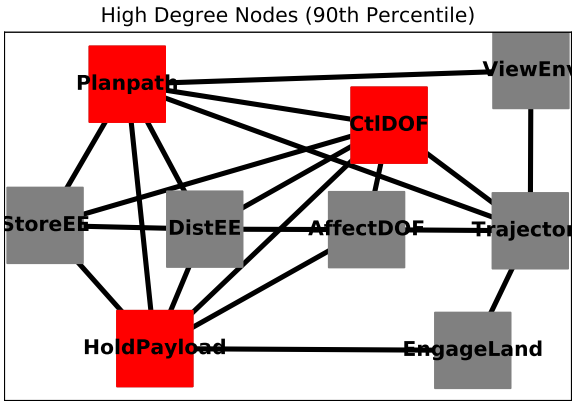

Figure 8: Visualization of high degree nodes in default (function) network representation of example drone model.

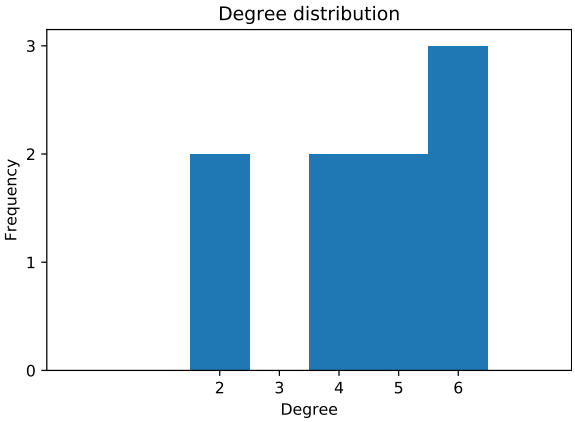

Figure 9: Degree distribution of default (function) network representation of example drone model.

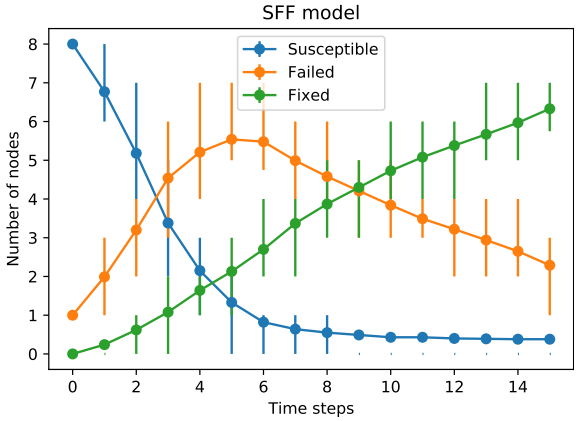

Figure 10: SFF model applied to function network representation of example drone model.

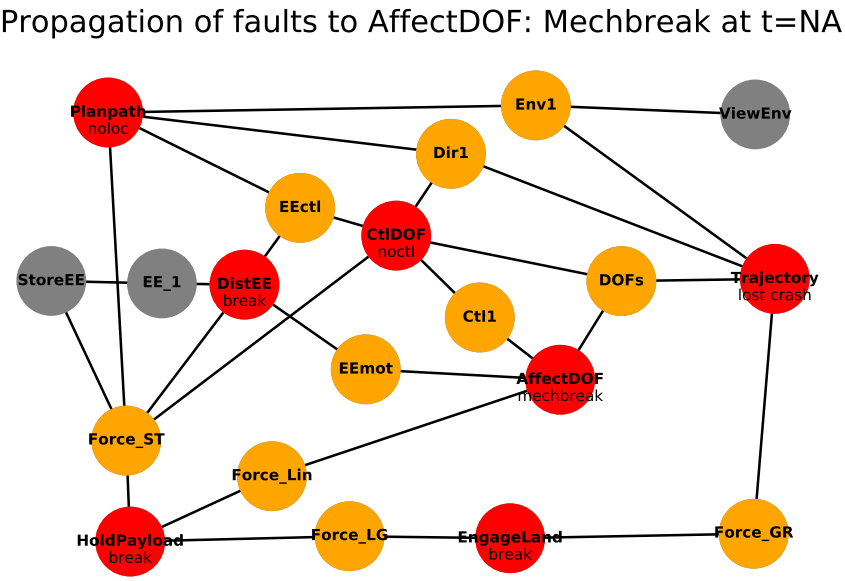

Figure 11: Static fault effects to the motor breaking: the drone crashes.

homogeneous degree distribution in Fig. 9 indicates that the network is not particularly robust to failure of critical nodes. The SFF model for the function network is provided in Fig. 10. This model demonstrates the system's resilience (based on parameter topology) to a cascading failure, given a user defined failure rate, fix rate, and start node (first node to fail). If various design alternatives are being considered, their relative resilience can be compared using the SFF model.

4.2 Static Representation and Analysis

To identify how faults lead to failures and begin to quantify risk in the system, the model is elaborated with flow attributes, function states, and failure logic, creating a static propagation model. As modelled in this system, deviations in the input of the Trajectory function block lead to a crash, which in turn propagates faults through the structure to initiate faults in the rest of the functions. Using this model (and an underlying fault probability model), one can create a FMEA-like table of fault effects, rates, and costs, as shown in Table 3, to show which faults have the highest impact on the design. As shown, most faults in this model lead to a crash, making the chosen rate the driving factor of expected cost. One of the faults under consideration in this model is a mechanical break causing the AffectDOF function to lose the ability to control the degrees of freedom of the drone. This fault is visualized in Figure 11, showing how fault leads to a crash and in turn faults in the other functions. This fault will be used to motivate analysis and design through the rest of this example.

Fault	Degraded Functions	Degraded Flows	Rate	Cost	Exp. Cost
StoreEE nocharge	StoreEE, DistEE, CtlDOF, Planpath, Trajectory...	Force_ST, Force_Lin, Force_GR, Force_LG, EE_1...	1e-05	183300	183300
Planpath degloc	DistEE, CtlDOF, Planpath, Trajectory, EngageL...	Force_ST, Force_Lin, Force_GR, Force_LG, EEmo...	8e-06	193000	154400
DistEE short	DistEE, CtlDOF, Planpath, Trajectory, EngageL...	Force_ST, Force_Lin, Force_GR, Force_LG, EEmo...	3e-06	186000	55800
AffectDOF ctlbreak	DistEE, AffectDOF, CtlDOF, Planpath, Trajecto...	Force_ST, Force_Lin, Force_GR, Force_LG, EEmo...	2e-06	184000	36800
AffectDOF ctlup	DistEE, AffectDOF, CtlDOF, Planpath, Trajecto...	Force_ST, Force_Lin, Force_GR, Force_LG, EEmo...	2e-06	183500	36700
DistEE break	DistEE, CtlDOF, Planpath, Trajectory, EngageL...	Force_ST, Force_Lin, Force_GR, Force_LG, EEmo...	2e-06	183000	36600
CtlDOF noctl	DistEE, CtlDOF, Planpath, Trajectory, EngageL...	Force_ST, Force_Lin, Force_GR, Force_LG, EEmo...	2e-06	183000	36600
AffectDOF short	DistEE, AffectDOF, CtlDOF, Planpath, Trajecto...	Force_ST, Force_Lin, Force_GR, Force_LG, EE_1...	1e-06	186200	18620
AffectDOF mechbreak	DistEE, AffectDOF, CtlDOF, Planpath, Trajecto...	Force_ST, Force_Lin, Force_GR, Force_LG, EEmo...	1e-06	183500	18350
AffectDOF openc	DistEE, AffectDOF, CtlDOF, Planpath, Trajecto...	Force_ST, Force_Lin, Force_GR, Force_LG, EEmo...	1e-06	183200	18320
Planpath noloc	Planpath, Trajectory	Ctl1, DOFs, Dir1	2e-06	60000	12000
CtlDOF degctl	CtlDOF	Force_GR, Force_LG, Ctl1, DOFs	8e-06	10000	8000
AffectDOF propbreak	DistEE, AffectDOF, CtlDOF, Planpath, Trajecto...	Force_ST, Force_Lin, Force_GR, Force_LG, EEmo...	3e-07	183200	5496
AffectDOF propstuck	DistEE, AffectDOF, CtlDOF, Planpath, Trajecto...	Force_ST, Force_Lin, Force_GR, Force_LG, EE_1...	2e-07	186200	3724
HoldPayload break	DistEE, CtlDOF, Planpath, Trajectory, EngageL...	Force_ST, Force_Lin, Force_GR, Force_LG, EEmo...	2e-07	183000	3660
ViewEnv poorview	ViewEnv		2e-06	10000	2000
EngageLand deform	EngageLand		8e-06	1000	800
HoldPayload deform	HoldPayload	Force_ST, Force_Lin	8e-07	10000	800
DistEE degr	DistEE	Force_GR, Force_LG, EEmot, EEctl, Ctl1, DOFs	5e-06	1000	500
EngageLand break	EngageLand		2e-06	1000	200
AffectDOF ctldn	AffectDOF	Force_GR, Force_LG, DOFs	2e-06	500	100
AffectDOF mechfriction	AffectDOF	EE_1, EEmot	5e-07	500	25
AffectDOF propwarp	AffectDOF	Force_GR, Force_LG, DOFs	1e-07	200	2

Table 3: Automatically-Generated Scenario-Based Static FMEA from model

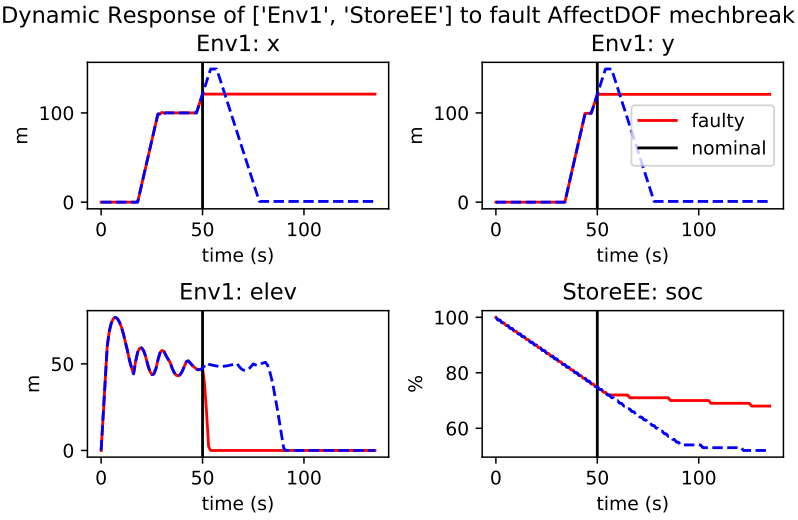

Figure 12: Dynamic behaviors of a motor breaking

4.3 Dynamic Representation and Analysis

In the dynamic model, the drone is given dynamic states and behaviors which iterate over time–in this case the position, velocity, charge, and perceived location of the system. This can then be used to model how the system behaves in faulty and nominal scenarios as shown in Figure 12 for the mechanical break fault. In the nominal case, the drone flies out to a location and returns to land, while in the faulty case the system crashes soon after the fault is injected, leaving it far from the landing location.

While this single-fault injection can help one understand how the system behaves, a fault injection approach can be used to quantify the expected effects of a fault which could occur at different points in the simulation. This is shown in Figure 13. As shown, the cost is high in the ascent and forward flight phase ($134K-$184K) since the fault then leads to a crash, and low in the descent phase ($500), since the drone has already landed. Additionally, the cost increases in the middle of the forward flight phase since the system crashes far from its landing location, which incurs additional cost in the model. While these results are consistent with the results of the static model at the point in time considered in the model (forward flight), they also show how a higher-fidelity dynamic model model can elicit a more nuanced consideration of fault effects.

4.4 Hierarchical Representation and Analysis

Given the effects of failures in this system, it is important to consider how they can be mitigated through the component architecture. In the drone model, for example, one has the ability to consider whether to realize the AffectDOF function with a quadrotor, hexarotor, or octorotor architecture. This example considers the quadrotor and octorotor architectures.

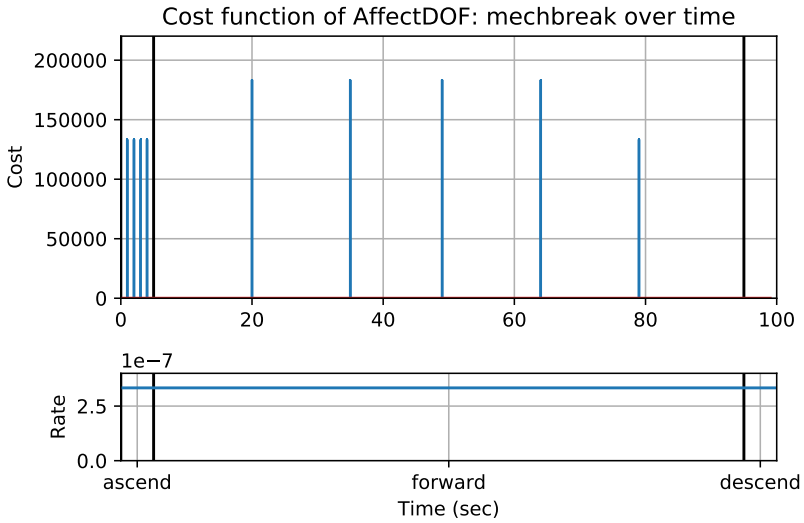

Figure 13: Modelled cost over time of the motor breaking

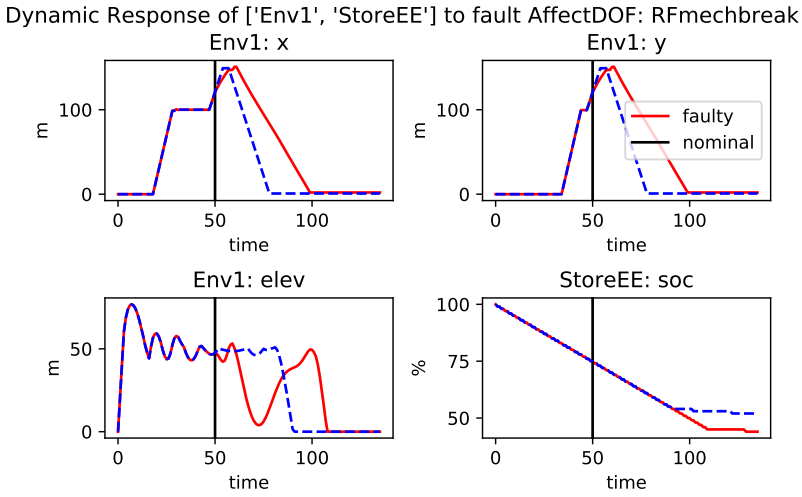

Figure 14: Fault behavior of octorotor

To compare how each architecture adapts to faults, the dynamic behaviors over time can be plotted. When considering the break fault, the quadrotor architecture reacts identically to the fault as in Figure 6, since losing one motor causes the system to lose stability. However, when the drone has an octorotor architecture, the system behaves as shown in Figure 14, faltering due to lost thrust but ultimately recovering and landing in the desired location. Thus the octorotor architecture is more resilient to this fault scenario.

However, to make a decision about component architectures on the basis of resilience, the expected cost of all fault scenarios for both architectures must be compared and weighted against the operational and implementation costs. To quantify this cost, each of the faults associated with the realized function (AffectDOF) are injected in the model according to a sampling approach. In this case, while the octorotor component architecture mitigates a number of scenarios due to the increased system redundancy, it does not mitigate every fault (e.g. control errors), and in fact increases the chance of other faults (e.g. electrical problems that propagate to the battery) because the larger number of components provides more opportunities for the fault to occur. Statistics from this fault approach are shown in Table 4. As shown, while the number of scenarios increases in the octorotor architecture, the number of scenarios which lead to a crash (and overall crash rate) is much lower, resulting in a lower overall resilience cost. This shows how fmdtools can be used to assist resilient design decision-making in the early design process.

	Quadrotor	Octorotor
Number of Scenarios	104	208
Number of Crashes	46	24
Crash Ratio	0.44	0.12
Crash Rate	2.4e-6	0.8e-6
Resilience Cost	45565	19359

Table 4: Cost of rotor faults in each architecture

5 Conclusions

The fmdtools project enables the consideration of resilience in the design process by providing a set of model classes, simulation methods, and analyses that enable the consideration of risk through the functional, behavioral, and structural stages of the design process. As demonstrated here, this enables one to analyze the system as the design becomes more detailed. fmdtools additionally has a number of features that make it relatively unique among fault simulation toolkits, including Python-based model definition, un-directed fault propagation, and simple model parameterization. This environment not only conducive to design, but may be easily modified and extended as needed to fulfill research needs, since all model attributes and simulation/analysis methods are defined in open-source Python code.

While the usefulness of this work is apparent from the demonstrations shown here, there are a number of possible extensions that would increase its practicality in design. First, safety is an important aspect of resilience that has specific requirements not explicitly taken into account in this work. That is, while this work is concerned with quantifying the costs of faults, safety procedures require one to quantify the overall cost of the entire set of failure scenarios, which often requires taking a deductive approach (ARP, 1996). Future work should address this by providing an approach to identify top-level failure events and quantify the risk of those events. Additionally, while the models used in this work are deterministic, failures can often have probabilistic effects that must be taken into account to accurately quantify overall risk. Future work should incorporate probabilistic state transitions into the model to enable non-deterministic fault propagation. Finally, while defining models directly in Python code increases model expressiveness, it forces one to use a stand-alone model and may make the toolkit difficult to use without the relevant programming knowledge.Future work should provide an interface for defining these models in an existing modelling tool-chain or model formalism (e.g. AADL) so the same model used by other design and analysis processes can be used to model resilience.

Acknowledgment

This material is based upon work supported by the National Science Foundation under Grant No. NSF CMMI #1562027. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the National Science Foundation.

The fmdtools package and scripts used to produce this work are archived at (Hulse et al., 2020) under open licenses.

References

Allenby, K. and Kelly, T. (2001). Deriving safety requirements using scenarios. In *Proceedings Fifth IEEE International Symposium on Requirements Engineering*, pages 228–235. IEEE.

ARP, S. (1996). 4761. *Guidelines and methods for conducting the safety assessment process on civil airborne systems and equipment*, 2.

Arribas, V., Nikova, S., and Rijmen, V. (2018). Vermi: Verification tool for masked implementations. In *ICECS*, pages 381–384. IEEE.

Banks, J., Reichard, K., Crow, E., and Nickell, K. (2009). How engineers can conduct cost-benefit analysis for phm systems. *IEEE Aerospace and Electronic Systems Magazine*, 24(3):22–30.

Barabási, A.-L. (2009). Scale-free networks: A decade and beyond. *Science*, 325(5939):412–413, ISSN: 0036-8075, DOI: 10.1126/science.1173299.

Boccaletti, S., Latora, V., Moreno, Y., Chavez, M., and Hwang, D.-U. (2006). Complex networks: Structure and dynamics. *Physics Reports*, 424(4):175 – 308, ISSN: 0370-1573, DOI: https://doi.org/10.1016/j.physrep.2005.10.009.

Bunus, P., Isaksson, O., Frey, B., and Münker, B. (2009). Rodon-a model-based diagnosis approach for the dx diagnostic competition. *Proc. DX'09*, pages 423–430.

Chemweno, P., Pintelon, L., Muchiri, P. N., and Van Horenbeek, A. (2018). Risk assessment methodologies in maintenance decision making: A review of dependability modelling approaches. *Reliability Engineering & System Safety*, 173:64–77.

Chiacchio, F., Aizpurua, J. I., Compagno, L., and D'Urso, D. (2020). Shyftoo, an object-oriented monte carlo simulation library for the modeling of stochastic hybrid fault tree automaton. *Expert Systems with Applications*, 146:113139.

Chiacchio, F., Aizpurua, J. I., Compagno, L., Khodayee, S. M., and D'Urso, D. (2019). Modelling and resolution of dynamic reliability problems by the coupling of simulink and the stochastic hybrid fault tree object oriented (shyftoo) library. *Information*, 10(9):283.

Choi, H. J., Atkins, E., and Yi, G. (2010). Flight envelope discovery for damage resilience with application to an f-16. In *AIAA Infotech@ Aerospace 2010*, page 3353.

Combemale, B., Crégut, X., Giacometti, J.-P., Michel, P., and Pantel, M. (2008). Introducing simulation and model animation in the mde topcased toolkit.

Cottam, B., Specking, E., Small, C., Pohl, E., Parnell, G. S., and Buchanan, R. K. (2019). Defining resilience for engineered systems. *Engineering Management Research*, 8(2):11–29.

Faturechi, R., Levenberg, E., and Miller-Hooks, E. (2014). Evaluating and optimizing resilience of airport pavement networks. *Computers & Operations Research*, 43:335–348.

Fraser, S., Simpson, A., Núñez, A., Deparday, V., Balog, S., Jongman, B., Murnane, R., Taillefer, N., Chauvin, N., Morel, O., et al. (2016). Thinkhazard!—delivering natural hazard information for decision making. In *2016 3rd International Conference on Information and Communication Technologies for Disaster Management (ICT-DM)*, pages 1–6. IEEE.

Gambi, A., Müller, M., and Fraser, G. (2019). Asfault: Testing self-driving car software using search-based procedural content generation. In *2019 IEEE/ACM 41st International Conference on Software Engineering: Companion Proceedings (ICSE-Companion)*, pages 27–30. IEEE.

Georgakoudis, G., Laguna, I., Vandierendonck, H., Nikolopoulos, D. S., and Schulz, M. (2019). Safire: Scalable and accurate fault injection for parallel multithreaded applications. In *2019 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pages 890–899. IEEE.

Goldstein, B., Srinivasan, S., Mellempudi, N. K., Das, D., Santiago, L., Ferreira, V. C., Solon, N., Kundu, S., and França, F. M. G. (2020). Reliability evaluation of compressed deep learning models. In *2020 IEEE 11th Latin American Symposium on Circuits Systems (LASCAS)*.

Grigoleit, F., Holei, S., Pleuß, A., Reiser, R., Rhein, J., Struss, P., and Wedel, J. v. (2016). The qsafe project–developing a model-based tool for current practice in functional safety analysis.

Gundermann, J., Kolesnikov, A., Cameron, M., and Blochwitz, T. (2019). The fault library-a new modelica library allows for the systematic simulation of non-nominal system behavior. In *Proceedings of the 2nd Japanese Modelica Conference, Tokyo, Japan, May 17-18, 2018*, number 148, pages 161–168. Linköping University Electronic Press.

Hagberg, A., Swart, P., and S Chult, D. (2008). Exploring network structure, dynamics, and function using networkx. Technical report, Los Alamos National Lab.(LANL), Los Alamos, NM (United States).

Haley, B., Dong, A., and Tumer, I. Y. (2016a). A comparison of network-based metrics of behavioral degradation in complex engineered systems. *Journal of Mechanical Design*, 138(12).

Haley, B., Dong, A., and Tumer, I. Y. (2016b). A comparison of network-based metrics of behavioral degradation in complex engineered systems. *Journal of Mechanical Design*, 138(12).

Holzel, N. B., Schilling, T., and Gollnick, V. (2014). An aircraft lifecycle approach for the cost-benefit analysis of prognostics and condition-based maintenance-based on discrete-event simulation. Technical report, DLR-German Aerospace Center Hamburg Germany.

Hönig, P., Lunde, R., and Holzapfel, F. (2017). Model based safety analysis with smartiflow. *Information*, 8(1):7.

Howard, T. J., Culley, S. J., and Dekoninck, E. (2008). Describing the creative design process by the integration of engineering design and cognitive psychology literature. *Design studies*, 29(2):160–180.

Hu, Y. (2005). *A guided simulation methodology for dynamic probabilistic risk assessment of complex systems*. PhD thesis.

Hulse, D., Hoyle, C., Goebel, K., and Tumer, I. (2019a). Using value assessment to drive phm system development in early design. In *Proceedings of the Annual Conference of the PHM Society*, volume 11.

Hulse, D., Hoyle, C., Goebel, K., and Tumer, I. Y. (2019b). Quantifying the resilience-informed scenario cost sum: A value-driven design approach for functional hazard assessment. *Journal of Mechanical Design*, 141(2).

Hulse, D., Walsh, H., and Zhang, H. (2020). Designengrlab/fmdtools: v0.5.3-revision2. DOI: 10.5281/zenodo.3759905, https://doi.org/10.5281/zenodo.3759905.

Hunter, J. D. (2007). Matplotlib: A 2d graphics environment. *Computing in Science & Engineering*, 9(3):90–95, DOI: 10.1109/MCSE.2007.55.

Irshad, L., Ahmed, S., Demirel, H. O., and Tumer, I. Y. (2019). Computational functional failure analysis to identify human errors during early design stages. *Journal of Computing and Information Science in Engineering*, 19(3).

Jensen, D., Tumer, I. Y., and Kurtoglu, T. (2009). Flow state logic (fsl) for analysis of failure propagation in early design. In *ASME 2009 International design engineering technical conferences and computers and information in engineering conference*, pages 1033–1043. American Society of Mechanical Engineers Digital Collection.

Jha, S., Banerjee, S. S., Cyriac, J., Kalbarczyk, Z. T., and Iyer, R. K. (2018). Avfi: Fault injection for autonomous vehicles. In *2018 48th Annual IEEE/IFIP International Conference on Dependable Systems and Networks Workshops (DSN-W)*, pages 55–56. IEEE.

Jha, S., Tsai, T., Hari, S., Sullivan, M., Kalbarczyk, Z., Keckler, S. W., and Iyer, R. K. (2019). Kayotee: A fault injection-based system to assess the safety and reliability of autonomous vehicles to faults and errors. *arXiv preprint arXiv:1907.01024*.

Joshi, A. and Heimdahl, M. P. (2005). Model-based safety analysis of simulink models using scade design verifier. In *International Conference on Computer Safety, Reliability, and Security*, pages 122–135. Springer.

Joshi, A. and Heimdahl, M. P. (2007). Behavioral fault modeling for model-based safety analysis. In *10th IEEE High Assurance Systems Engineering Symposium (HASE'07)*, pages 199–208. IEEE.

Joshi, A., Heimdahl, M. P., Miller, S. P., and Whalen, M. W. (2006). Model-based safety analysis.

Kollárová, M. (2014). Fault injection testing of openstack. *Ph. D. dissertation*.

Krus, D. and Lough, K. G. (2009). Function-based failure propagation for conceptual design. *AI EDAM*, 23(4):409–426.

Kurtoglu, T. and Tumer, I. Y. (2008). A graph-based fault identification and propagation framework for functional design of complex systems. *Journal of mechanical design*, 130(5).

Lattmann, Z., Pop, A., De Kleer, J., Fritzson, P., Janssen, B., Neema, S., Bapty, T., Koutsoukos, X., Klenk, M., Bobrow, D., et al. (2014). Verification and design exploration through meta tool integration with openmodelica. In *Proceedings of the 10 th International Modelica Conference; March 10-12; 2014; Lund; Sweden*, number 096, pages 353–362. Linköping University Electronic Press.

Lunde, K., Lunde, R., and Münker, B. (2006). Model-based failure analysis with rodon. In *Proceedings of the 2006 conference on ECAI 2006: 17th European Conference on Artificial Intelligence August 29–September 1, 2006, Riva del Garda, Italy*, pages 647–651. IOS Press.

Martins, R., Gandhi, R., Narasimhan, P., Pertet, S., Casimiro, A., Kreutz, D., and Veríssimo, P. (2013). Experiences with fault-injection in a byzantine fault-tolerant protocol. In *ACM/IFIP/USENIX International Conference on Distributed Systems Platforms and Open Distributed Processing*, pages 41–61. Springer.

Matloff, N. (2008). Introduction to discrete-event simulation and the simpy language. *Davis, CA. Dept of Computer Science. University of California at Davis. Retrieved on August*, 2(2009):1–33.

May, D. and Stechele, W. (2012). An fpga-based probability-aware fault simulator. In *2012 International Conference on Embedded Computer Systems (SAMOS)*, pages 302–309. IEEE.

McIntire, M. G., Keshavarzi, E., Tumer, I. Y., and Hoyle, C. (2016). Functional models with inherent behavior: Towards a framework for safety analysis early in the design of complex systems. In *ASME 2016 International Mechanical Engineering Congress and Exposition*. American Society of Mechanical Engineers Digital Collection.

McKenna, F. (2011). Opensees: a framework for earthquake engineering simulation. *Computing in Science & Engineering*, 13(4):58–66.

Mehrpouyan, H., Haley, B., Dong, A., Tumer, I. Y., and Hoyle, C. (2013). Resilient design of complex engineered systems against cascading failure. In *ASME 2013 International Mechanical Engineering Congress & Exposition*, volume 12: Systems and Design, page V012T13A063, San Diego. ASME.

Miles, S. B. (2018). Participatory disaster recovery simulation modeling for community resilience planning. *International Journal of Disaster Risk Science*, 9(4):519–529.

Minhas, R., De Kleer, J., Matei, I., Saha, B., Janssen, B., Bobrow, D. G., and Kurtoglu, T. (2014). Using fault augmented modelica models for diagnostics. In *Proceedings of the 10 th International Modelica Conference; March 10-12; 2014; Lund; Sweden*, number 96, pages 437–445. Linköping University Electronic Press.

Morozov, A., Ding, K., Steurer, M., and Janschek, K. (2019). Openerrorpro: A new tool for stochastic model-based reliability and resilience analysis. In *2019 IEEE 30th International Symposium on Software Reliability Engineering (ISSRE)*, pages 303–312. IEEE.

Morozov, A., Mutzke, T., Ren, B., and Janschek, K. (2018). Aadl-based stochastic error propagation analysis for reliable system design of a medical patient table. In *2018 Annual Reliability and Maintainability Symposium (RAMS)*, pages 1–7. IEEE.

Newman, M. E. J. (2010). *Networks*. Oxford University Press, New York, New York.

Niermann, T. M., Cheng, W.-T., and Patel, J. H. (1992). Proofs: A fast, memory-efficient sequential circuit fault simulator. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 11(2):198–207.

Noh, K.-W., Jun, H.-B., Lee, J.-H., Lee, G.-B., and Suh, H.-W. (2011). Module-based failure propagation (mfp) model for fmea. *The International Journal of Advanced Manufacturing Technology*, 55(5-8):581–600.

Pahl, G. and Beitz, W. (2013). *Engineering design: a systematic approach*. Springer Science & Business Media.

pandas development team, T. (2020). pandas-dev/pandas: Pandas. DOI: 10.5281/zenodo.3509134, https://doi.org/10.5281/zenodo.3509134.

Papadopoulos, Y. and McDermid, J. A. (1999). Hierarchically performed hazard origin and propagation studies. In *International Conference on Computer Safety, Reliability, and Security*, pages 139–152. Springer.

Papadopoulos, Y., Walker, M., Parker, D., Rüde, E., Hamann, R., Uhlig, A., Grätz, U., and Lien, R. (2011). Engineering failure analysis and design optimisation with hip-hops. *Engineering Failure Analysis*, 18(2):590–608.

Patelli, E. and Broggi, M. (2015). Uncertainty management and resilient design of safety critical systems. In *NAFEMS World Congress 2015*.

Patelli, E., Tolo, S., George-Williams, H., Sadeghi, J., Rocchetta, R., de Angelis, M., and Broggi, M. (2018). Opencossan 2.0: an efficient computational toolbox for risk, reliability and resilience analysis. In *Proceedings of the joint ICVRAM ISUMA UNCERTAINTIES conference*, volume 2018.

Schirmeier, H., Hoffmann, M., Dietrich, C., Lenz, M., Lohmann, D., and Spinczyk, O. (2015). Fail*: An open and versatile fault-injection framework for the assessment of software-implemented hardware fault tolerance. In *2015 11th European Dependable Computing Conference (EDCC)*, pages 245–255. IEEE.

Schlömer, N., Papior, N. R., Ancellin, M., and Arnold, D. (2020). nschloe/quadpy v0.14.7. DOI: 10.5281/zenodo.3752151, https://doi.org/10.5281/zenodo.3752151.

Short, A. R. (2016). *Design of autonomous systems for survivability through conceptual object-based risk analysis*. PhD thesis, Colorado School of Mines. Arthur Lakes Library.

Stewart, D., Liu, J. J., Whalen, M. W., Cofer, D., and Peterson, M. (2018). Safety annex for the architecture analysis and design language.

Stone, R. B., Tumer, I. Y., and Van Wie, M. (2005). The function-failure design method.

Treuner, F., Hübner, D., Baur, S., and Wagner, S. M. (2014). A survey of disruptions in aviation and aerospace supply chains and recommendations for increasing resilience. *Supply Chain Management*, 14(3):7–12.

van der Linden, F. L. (2014). General fault triggering architecture to trigger model faults in modelica using a standardized blockset. In *Proceedings of the 10th International Modelica Conference-Lund, Sweden-Mar 10-12, 2014*, number 96, pages 427–436. LiU Electronic Press.

Viana, M. P., Tanck, E., Beletti, M. E., and da Fontoura Costa, L. (2009). Modularity and robustness of bone networks. *Molecular BioSystems*, 5(3):255–261.

Wadhawan, Y. and Neuman, C. (2017). Bags: A tool to quantify smart grid resilience. In *FedCSIS Communication Papers*, pages 323–332.

Walsh, H. S., Dong, A., and Tumer, I. Y. (2018). The role of bridging nodes in behavioral network models of complex engineered systems. *Design Science*, 4(e8), DOI: 10.1017/dsj.2017.31.

Walsh, H. S., Dong, A., and Tumer, I. Y. (2019). An analysis of modularity as a design rule using network theory. *Journal of Mechanical Design*, 141(3):031102.

Wang, Z., Cui, Y., and Shi, J. (2015). A framework of discrete-event simulation modeling for prognostics and health management (phm) in airline industry. *IEEE Systems Journal*, 11(4):2227–2238.

Winter, S., Piper, T., Schwahn, O., Natella, R., Suri, N., and Cotroneo, D. (2015). Grinder: On reusability of fault injection tools. In *2015 IEEE/ACM 10th International Workshop on Automation of Software Test*, pages 75–79. IEEE.

Yodo, N. and Wang, P. (2016a). Engineering resilience quantification and system design implications: A literature survey. *Journal of Mechanical Design*, 138(11).

Yodo, N. and Wang, P. (2016b). Resilience allocation for early stage design of complex engineered systems. *Journal of Mechanical Design*, 138(9).

Youn, B. D., Hu, C., and Wang, P. (2011). Resilience-driven system design of complex engineered systems. *Journal of Mechanical Design*, 133(10).