


An ontology-based fault generation and fault propagation analysis approach for safety-critical computer systems at the design stage

Xiaoxu Diao , Mike Pietrykowski, Fuqun Huang, Chetan Mutha
and Carol Smidts

The Ohio State University, 201 W. 19th AVE, Columbus, OH 43210, USA

Research Article

Cite this article: Diao X, Pietrykowski M, Huang F, Mutha C, Smidts C (2022). An ontology-based fault generation and fault propagation analysis approach for safety-critical computer systems at the design stage. *Artificial Intelligence for Engineering Design, Analysis and Manufacturing* **36**, e1, 1–32. <https://doi.org/10.1017/S0890060421000342>

Received: 26 March 2021
Revised: 8 October 2021
Accepted: 1 November 2021

Key words:

computer systems; fault ontology; fault propagation analysis; functional failure analysis

Author for correspondence:

Xiaoxu Diao,
E-mail: diao.38@osu.edu

Abstract

Fault propagation analysis is a process used to determine the consequences of faults residing in a computer system. A typical computer system consists of diverse components (e.g., electronic and software components), thus, the faults contained in these components tend to possess diverse characteristics. How to describe and model such diverse faults, and further determine fault propagation through different components are challenging problems to be addressed in the fault propagation analysis. This paper proposes an ontology-based approach, which is an integrated method allowing for the generation, injection, and propagation through inference of diverse faults at an early stage of the design of a computer system. The results generated by the proposed framework can verify system robustness and identify safety and reliability risks with limited design level information. In this paper, we propose an ontological framework and its application to analyze an example safety-critical computer system. The analysis result shows that the proposed framework is capable of inferring fault propagation paths through software and hardware components and is effective in predicting the impact of faults.

Introduction

Computer systems generally consist of multiple hardware and software components with diverse functionalities. With the increasing number of task requirements of safety-critical systems, computer systems are widely used in safety-critical domains. The faults residing in computer systems have posed increasing threats to reliability and safety (Weichhart *et al.*, 2016; Isaksson *et al.*, 2018; Jiang *et al.*, 2018). And yet, challenges exist in assessing and migrating the risks of faults:

- 1) Fault properties are diverse and distinct in different domains (Avizienis *et al.*, 2001). A typical computing system consists of hardware platforms (HW) and multiple user software applications (SW) running on various operating systems (OS). The faults of the hardware platform are related to the environmental stress and the component degradation with time of service. On the other hand, software does not degrade physically, and the faults of the operating system and user programs are related to human errors, requirements, program structure, logic, and inputs (Park *et al.*, 2012).
- 2) The triggering conditions for faults are complex. System faults can be activated by multiple conditions such as the properties of components, components' inner structures, working environments, and timing aspects. For example, buffer overflow (Foster *et al.*, 2005), data race (O'Callahan and Choi, 2003), and other types of software faults (Durães and Madeira, 2006) may be created in an immature multitasking program and activated at a specific point in time with particular input data and hardware configurations.
- 3) The fault propagation paths are sophisticated, especially when the effects of the faults propagate across HW and SW domains (Shu *et al.*, 2016). This problem commonly exists in computer system architectures where user programs are usually assigned dynamically to unpredictable memory spaces or other physical resources.

In summary, faults in a computer system may occur under complex conditions (e.g., specific inputs and states) and pass through HW and SW components to cause functional failures of the entire system. The impacts of these potential faults on system reliability and safety are usually not fully considered and consequently lead to unexpected outages of services delivered by such systems.

An integrated approach is needed to describe the diverse features of the various faults of computer systems. Fault analysis at the design stage can effectively predict possible system failures before implementation. Fault analysis provides useful information to the system designer

© The Author(s), 2022. Published by Cambridge University Press. This is an Open Access article, distributed under the terms of the Creative Commons Attribution licence (<http://creativecommons.org/licenses/by/4.0/>), which permits unrestricted re-use, distribution and reproduction, provided the original article is properly cited.

for establishing a fault tolerance mechanism and increasing the reliability and robustness of the system (Gao *et al.*, 2008; Mutha *et al.*, 2013; Yang *et al.*, 2013). However, the following challenges prevent existing methods from use at the early design stage:

- 1) Many of the current fault analysis methods are specific to a fault type or system type (Yang *et al.*, 2015; Diao *et al.*, 2018; Dibowski *et al.*, 2017). To achieve a wide coverage of faults encountered in computer systems, various analysis methods need to be performed, which is time-consuming for system analysis.
- 2) The diversities of data and models cause difficulties in managing and reusing historical data. Each domain-specific analysis method uses its own approach to model and organize the knowledge of systems and faults. More general approaches are required when analyzing faults related to multiple domains.
- 3) Lack of automation requires significant manual effort. Also, fault generation and injection are usually based on expert experience and as such involve subjective evaluation and lack a systematic evaluation of systems.

These challenges are addressed in this paper by proposing an Integrated System Failure Analysis using an ONtological framework (IS-FAON) to model, generate, and analyze faults in computer systems including their activation, propagation, as well as their effects on functionality. Without details on the implementation of the system under analysis (SUA), the proposed framework allows system designers to observe system responses under nominal and faulty states at an early design stage and to effectively evaluate the robustness of the system under development before its implementation. In detail, the contributions of the present research are as follows:

- Proposed an integrated ontology framework that is capable of describing the features of both software and hardware faults in computer systems. The proposed ontology framework contains theories and prototype tools for predicting potential effects of faults at the design stage.
- Developed a series of domain-specific ontologies for representing the faults and their corresponding impacts in the fields of computer architecture, operating system, and user applications that enables handling diversity in data and model in a single framework. These ontologies can effectively reuse the existing knowledge of computer systems for fault analysis at the design stage when the target system has not been implemented.
- Defined a set of fault generation rules that can be applied to the models of the SUA to generate various types of faults and to automatically inject the generated faults into the SUA. The fault generation rules can maximumly cover the potential faults based on the known information of the target system. As such, effects of one or more faults (expected or unexpected by an expert) may be simulated and analyzed systematically.
- Designed an inference-based fault propagation analysis method based on logic inference that performs qualitative fault analysis based on the proposed ontological concepts. The proposed method can automatically predict the impacts of the potential faults. As such, a wide variety of individual faults or a combination of faults may be analyzed so that system designers and developers can improve the robustness of the target system.
- Conducted a case study on a safety-critical computer system to examine the correctness and effectiveness of the proposed approach. Although limited uncertainties exist in the analysis

result, the proposed fault analysis framework can effectively and correctly predict the effects of faults without detailed system implementation.

The paper is organized as follows: Section “Related work” reviews existing research devoted to fault propagation analysis and ontology. Section “Ontological framework for fault propagation analysis” introduces the ontological framework for fault propagation analysis. The ontology framework includes ontologies for system modeling, fault generation and injection, fault inference and analysis. The ontologies for system modeling are described in the section “System modeling.” The fault ontologies for fault generation are described in the section “Fault ontology and fault generation.” Section “Fault injection” focuses on injecting faults into the system models established in the section “System modeling.” Section “Fault propagation inference” focuses on the methodology related to fault inference and analysis using the system model, and faults. A case study to illustrate the application of the proposed ontological framework is described in the section “Case Study.” Furthermore, Section “Discussion” discusses the results of the case study, and finally Section “Conclusion” provides the conclusion and introduces future research.

Related work

Faults are caused by multiple factors across both HW and SW components and interactions between them. Faults derived from single components will possibly propagate through multiple types of components and may impact multiple tasks (Weichhart *et al.*, 2016; Isaksson *et al.*, 2018; Jiang *et al.*, 2018). Because of the diversity and complexity of faults in computer systems and the lack of information on a target system at the early design stage, researchers have attempted to solve the fault propagation and effect analysis problem without precise system models.

In the existing fault analysis approaches, Fault Tree Analysis (FTA; Lauer *et al.*, 2011) and Failure Modes and Effects Analysis (FMEA; Hecht and Baum, 2019) reuse modeling information of HW and SW components and trace the propagation paths of internal faults. The Fault Propagation and Transformation Calculus (FPTC; Wallace, 2005) uses architecture graphs to model the structure of HW and SW components and uses predefined symbols to model the component behaviors. It infers the system responses caused by single faults derived from software components in a single-task real-time system. The Functional Failure Identification and Propagation (FFIP; Papakonstantinou and Sierla, 2012) copes with faults that propagate over subsystems and cross the domain boundaries between electronics and mechanics. The improved signed directed graph (SDG) model (Yang *et al.*, 2013) describes the system variables and their cause–effect relations in a continuous process. It allows to obtain the fault propagation paths using the method of graph search. The Small World Network (SWN) model (Gao *et al.*, 2008) focuses essentially on the topological structure properties of the computer system network with several principles that are capable of assessing the safety characteristics of the network nodes. It uses the weight of the link between the nodes to define the fault propagation intensity considering the network statistical information. Subsequently, the critical nodes and the fault propagation paths with high risk are obtained through qualitative fault propagation analysis. Interface Automata (IA; Zhao *et al.*, 2016) gives a formal and abstract description of the interactions between components and the environment. It extends interaction models

on the system interface level with failure modes and provides automated support for failure analysis. Integrated System Failure Analysis (ISFA; Mutha *et al.*, 2013) uses views of functions and components to simulate the propagation of single or multiple faults in single process systems across software and hardware (mechanical) domains.

Table 1 compares the methods mentioned above with the one proposed in this paper in terms of the ability of each method to handle concurrent faults, perform multiple processes, reusability of models, handling of SW and HW behaviors, automation of analysis, and capability of fault injection and fault generation. The method proposed in this paper can generate new types of faults and infer the effects of faults for a SUA at the early design stage, which is an important contribution of this method.

To effectively collect and manage the knowledge of faults, ontological theories have been widely applied to various industrial systems, such as to diagnosis systems (Liu *et al.*, 2019) for rotating machinery (Chen *et al.*, 2015) and chemical processes (Natarajan and Srinivasan, 2010), to the fault management of aircrafts (Zhou *et al.*, 2009) and smart home services (Etzioni *et al.*, 2010), as well as to the fault propagation analysis of building automation systems (Dibowski *et al.*, 2017) and wireless sensor networks (Benazzouz *et al.*, 2014). These studies applied the ontological theories to different specific types of systems and created concepts and notations for modeling faults in their target systems. In this paper, we employ the ontological theories to express and analyze faults in HW and SW for computer systems. The proposed theories and tools are dedicated to model and infer the creation and propagation of faults and soundly infer the consequences caused by these faults. By taking advantage of ontologies, this paper provides fundamental concepts to solve the knowledge description and integration issues involved in fault analysis. In practice, we employed the Web Ontology Language (OWL; Allen and Unicode Consortium, 2007) as the modeling language for establishing the proposed ontologies and used the Semantic Web Rule Language (SWRL; Horrocks *et al.*, 2004) as a supplement for implementing related rules and constraints. We selected

Protégé (Musen, 2015) as the editor for creating and debugging the proposed ontologies.

Ontological framework for fault propagation analysis

“An ontology is an explicit specification of a conceptualization” (Gruber *et al.*, 2012). As an effective way for information standardization and sharing, ontologies have become increasingly valuable in the fields of computer science for their utility in enabling thorough and well-defined discourse as well as for building logical models of systems. In the proposed ontological framework, we employed ontologies to standardize knowledge related to fault propagation analysis and utilize the information associated with the SUA at the early design stage to maximumly cover various types of faults and to effectively infer their effects on the SUA. This section introduces concepts of fault propagation and fault analysis as used herein, while detailed ontologies are defined and discussed in detail in the sections “System modeling” and “Fault ontology and fault generation.”

Fault propagation

Figure 1 illustrates a fault propagation path through an SUA highlighted by bold lines originating from a fault and leading up to a failure. In the figure, *components* are the essential HW or SW objects that constitute a computer system (located at the bottom). Each component implements one or more *functions*. Normally, a component will interact with other components during system operation. These interactions are modeled by *flows*, which represent the travel of objects through components or functions. The traveling objects can be materials, energy, or signals. The relations between the input and output flows of a component are *behaviors* of such a component. A component's behaviors are related to its *states*. The set of components, related flows, and their states is defined as a *system configuration* (Avizienis *et al.*, 2004a).

In Figure 1, a *fault* (in the block with bold boundaries) is the cause of an *error*. The error, which is the deviation of the state of

Table 1. Comparison of fault analysis methods for computer systems

Approaches	Concurrent Faults ^a	Multiple Processes ^b	Model Reusable ^c	SW and HW Behaviors ^d	Analysis Automation ^e	Automated New Fault Generation ^f
FTA (Lauer <i>et al.</i> , 2011)	No	No	Yes	No	Semi	No
FMEA (Hecht and Baum, 2019)	No	No	Yes	No	No	No
FPTC (Wallace, 2005)	No	No	No	No	Yes	No
FFIP (Papakonstantinou and Sierla, 2012)	Yes	No	No	No	No	No
SDG (Yang <i>et al.</i> , 2013)	Yes	No	No	Yes	Yes	No
SWN (Gao <i>et al.</i> , 2008)	Yes	Yes	No	No	Yes	No
IA (Zhao <i>et al.</i> , 2016)	Yes	Yes	No	Yes	Yes	No
ISFA (Mutha <i>et al.</i> , 2013)	Yes	No	No	Yes ^g	Yes	No
IS-FAON	Yes	Yes	Yes	Yes	Yes	Yes

^a If the method can analyze the faults that occurred concurrently.

^b If the method can analyze the faults that occurred in the system with multiple processes.

^c If the method can reuse the existing models established for other systems.

^d If the method can model behaviors of software and hardware components.

^e If the method can perform the fault analysis automatically.

^f If the method provides abilities to generate new types of faults and inject faults into the target system model.

^g Mechanical Hardware is supported only.

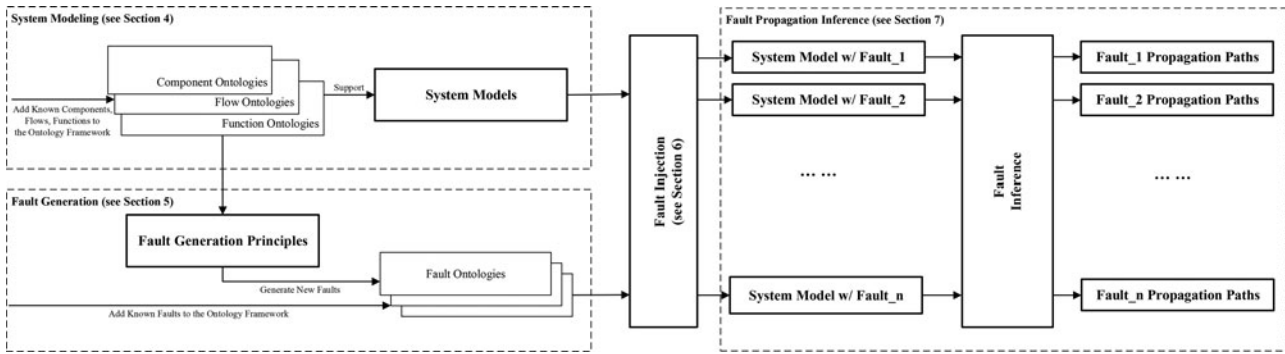


Fig. 1. A classic fault propagation path.

the system under analysis, would probably, in turn, activate another dormant fault and hence lead to another error. Consequently, this process will possibly trigger a function’s *failure* or *degradation*, which are the events that occur when the function deviates from the nominal states.

Fault analysis framework

Fault analysis is a process to identify the potential faults that may occur during the development and operation of the SUA, and to infer the impacts of a fault on the SUA. Figure 2 illustrates the main process of fault propagation analysis and the roles of the proposed ontologies in the analysis process. The fault analysis process starts with the system models based on component, flow, and function ontologies (see Section “System modeling”). Faults are modeled using fault ontologies. The framework provides the fault generation principles necessary to generate faults (see Section “Fault ontology and fault generation”) based on the component, flow, and function ontologies. The fault generation methodology will effectively improve the coverage of different types of faults. Then, the framework injects faults into the system models (see Section “Fault injection”). Based on the fault-seeded system models, the framework is capable of inferring the effects of faults and generating fault propagation paths (see Section “Fault Propagation Inference”).

System modeling

In ontology theories, *classes* and their hierarchical *links* represent objects and their classifications, respectively. An ontology uses *properties* (aka *predicates* in description logics) to represent the *attributes* of an object or the *relations* between objects. Properties can be categorized into *data properties* and *object properties*. Data properties use numbers or descriptive strings to represent an object’s attributes, such as the temperature of a computer processor. Object properties build the link between two or more objects. For example, the output of a memory unit (aka a component object) is the pressure data (aka a signal flow object) read by a sensor (aka a component object). Also, an ontology can define dependencies and constraints between classes and properties to represent natural laws or restrictions.

Ontologies for system modeling

Definition 1. *Component Ontology* is the foundation necessary to model the components of a computer system under analysis and defines how to model a new component of a computer system. The key process of modeling components is to abstract the generic attributes of concrete components by using the ontological concepts. Figure 3 shows the hierarchy of the classes created in the component ontology. We classified the components of a computer

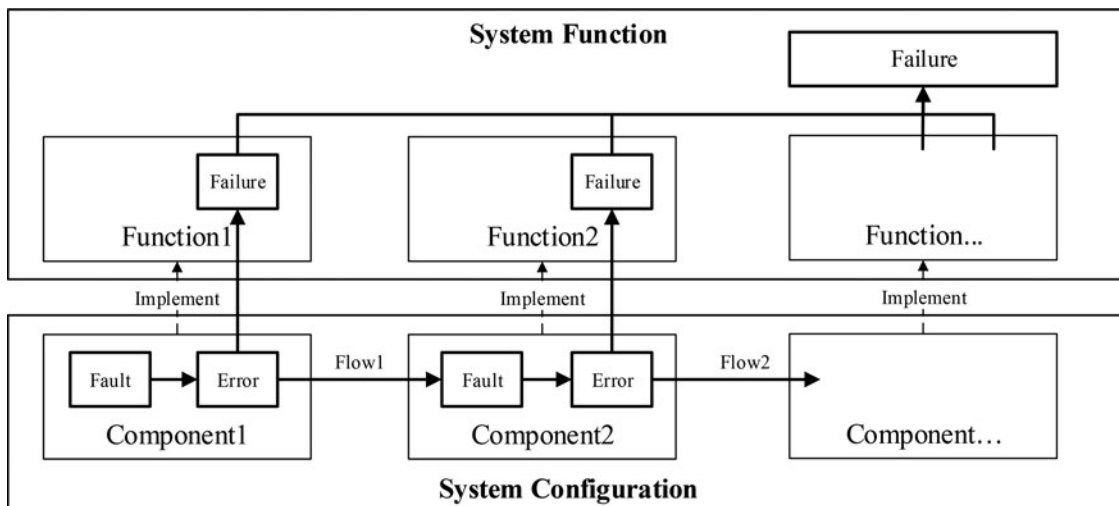


Fig. 2. Fault propagation analysis methodology.



Fig. 3. Hierarchy of the component ontology.

system into “Hardware Component,” “Software Component,” and “Operating System Component.”

Table 2 summarizes the properties related to the classes of components. The properties “ComposedOf” and “Location” define the relations between components. These relations decide on the occurrence of some types of faults. The properties “Inputs” and “Outputs” participate in the fault inference process since the effects of faults will propagate through the input and output flows of components. The property “purpose” links components to the functions they implement whose states will be inferred during fault inference. “Qualities” are measurable properties representing the component’s attributes in nominal and faulty states. Faults may be activated when these measurable properties change. At last, the property “States” organizes behaviors of components in different states. These behaviors are evidence used in inferring functional states during fault inference, detailed in the section “States and behavioral rules.”

To clarify these concepts further, a multi-core processor is used as an example component and is characterized using the component ontology derived from these concepts, see Table 3.

Definition 2. The Flow Ontology defines the classes related to the transition of objects between components and functions, which are involved in the propagation of the effects of a fault. The flow can track the transit of an object from its source position to its final destination as it weaves through the various components of the system. An example of flows would be the travel of a mouse click signal from a fingertip, into the universal serial bus (USB) port in the rear of the computer, into the system

bus, and finally reaching the processor. Figure 4 shows the hierarchy of the flow ontology. It is worthy to note that new types of flows can be added to the flow ontology if required.

Table 4 lists the properties defined in the flow ontology. In the table, “Qualities” are the properties of a flow that will be specified as data properties with constants or dynamic values. For example, a Command Flow (CF) records the information that a processor requires from a software program to operate. A CF has three important qualities: (1) the “command type” represents what actions the processor should perform, for example, reading data from a memory unit; (2) the “target address” represents the address of the memory unit or the I/O devices; and (3) the “operation data” represents the data that corresponds to the “command.” Table 5 details the properties of the active command flow as an example.

By using component and flow ontologies, we can model the structure of the SUA and the attributes of system components and flows. Some properties of components or flows can be missing when building the system model at the early design stage. For instance, we can use the concepts of a multi-core processor without defining its speed or other specification when designing a system. Along with the evolution of system design and development, the rough system model, built at the early design stage, will be more and more concrete and detailed. With the increasing concreteness of the components and flows in the model, more precise system behaviors can be emulated and analyzed, and more types of faults can be covered and analyzed by the proposed method.

Table 2. Properties defined in the component ontology

Property Name	Notations	Property Type	Description
ComposedOf	<i>ComposedOf</i> (\cdot)	Object (related to the Component Ontology)	<i>ComposedOf</i> specifies the constitution relation between components. The set of composites contains the subcomponents that constitute an integrated component.
Location	<i>Location</i> (\cdot)	Object (related to the Component Ontology)	<i>Location</i> represents the position relative to other components.
Inputs	<i>Inputs</i> (\cdot)	Object (related to the Flow Ontology)	<i>Inputs</i> represent the flows received from outside of a component.
Outputs	<i>Outputs</i> (\cdot)	Object (related to the Flow Ontology)	<i>Outputs</i> represent the flows sent out from the current component.
Purpose	<i>Purpose</i> (\cdot)	Object (related to the Function Ontology)	<i>Purpose</i> describes the goal of a component, which usually is its function.
Qualities	<i>Qualities</i> (\cdot)	Data (decided by specific qualities)	<i>Qualities</i> are the measurable properties that express particular characteristics of the current component.
States	<i>States</i> (\cdot)	Object (related to the State Ontology)	<i>States</i> are objects containing the behaviors and triggering conditions. See Section “States and behavioral rules” for details.

Definition 3. The *Functional ontology* describes functional knowledge pertaining to the corresponding components or systems. In this paper, functions are classified based on the taxonomy provided by the reference (Hirtz et al., 2002). Figure 5 shows the hierarchy of the function ontology.

Table 6 summarizes the properties defined for the functional ontology. It is worth noting that we use the same terms in the component and function ontologies, such as the “ComposedOf” and “States.” But these terms do not represent the same object. For example, a state of a component cannot be a state of a function. Similarly, a function cannot be composed of subcomponents. As an example class of the functional ontology, Table 7 shows the “Execute Command” function defined for a processor.

The function and flow ontologies allow system developers to model functionalities of the SUA and establish the mapping relations between components and their functions. With the functions and flows, the fault inference can predict the impact of faults on components and system functions.

Individuals and system models

A system model is a combination of *individuals* which are *instantiated* from the classes and properties defined by ontologies. For example, a class “multi-core processor” can be defined using the component ontology (see Section “Ontologies for system

Table 3. An example multi-core processor (MCP) component defined using the component ontology

Component: Multi_Core_Processor (MCP)		
Properties	Notations	Property Values
Composedof	<i>ComposedOf</i> (MCP)	{Core, RegA, RegB...} (a set of Hardware Components)
Location	<i>Location</i> (MCP)	{Connected to a Command Bus (a Hardware Component), Running a program (a Software Component)}
Inputs	<i>Inputs</i> (MCP)	{Command Flow, Memory Return Flow, IO Return Flow,...} (a set of Flows)
Outputs	<i>Outputs</i> (MCP)	{Memory Bus Flow, IO Bus Flow,...} (a set of Flows)
Purpose	<i>Purpose</i> (MCP)	{Execute Commands} (a set of Functions)
Qualities	<i>Qualities</i> (MCP)	{Clock Frequency, ...} (a set of Data Properties)
States	<i>States</i> (MCP)	{IdleState, ReadIOState, ReadMemState, WriteIOState, WriteMemState, ...} (a set of States)

modeling”) with the properties of generic inputs (e.g., I/O buses), outputs, etc. When establishing a system model with this type of processor, the abstract concept (i.e., the multi-core processor class) will be instantiated as a component individual (e.g., a processor named as “CPU_0”) in the system model. As a result, a system model is a super set of individuals and their properties which are instantiated from the classes defined by the ontologies in this section.

According to the type of included individuals, a *system model* is composed of *component models* and *functional models*. A *component model* is a structural model with the individuals of components and associated flow, representing system configurations. Whereas a *functional model* contains the individuals of functions and associated flows, representing the functions and their relations the target system needs to implement. Table 8 shows the composition of different types of models. An individual of flow in a component model may represent the same flow in the real world as the one in a functional model. An example can be the “Flow 1” and “Flow 2” in Figure 1. This implies that the functions “Function1” and “Function2” share the same relation as the one existing between the component “Component1” and “Component2.” These relations will be used when inferring the state of functions based on the behaviors of components, or vice versa. Component models and functional models can be integrated into synthetic models seamlessly through the dependencies introduced in the section “Dependencies and restrictions.”

States and behavioral rules

A state describes the combination of triggering conditions and behaviors of components, flows, functions, and faults. A transition of state describes the evolution of an object in terms of events or time sequence. The hierarchy of states considered in the proposed ontology is shown in Figure 6. For instance, the states of a component can be categorized into nominal states or faulty states; the states of a flow can be normal or abnormal; the states of a function can be Operating, Degraded, Lost, or Unknown. Figure 6 also classifies the states of a fault as dormant states, activated states, and terminated states. Since the host entity of a fault

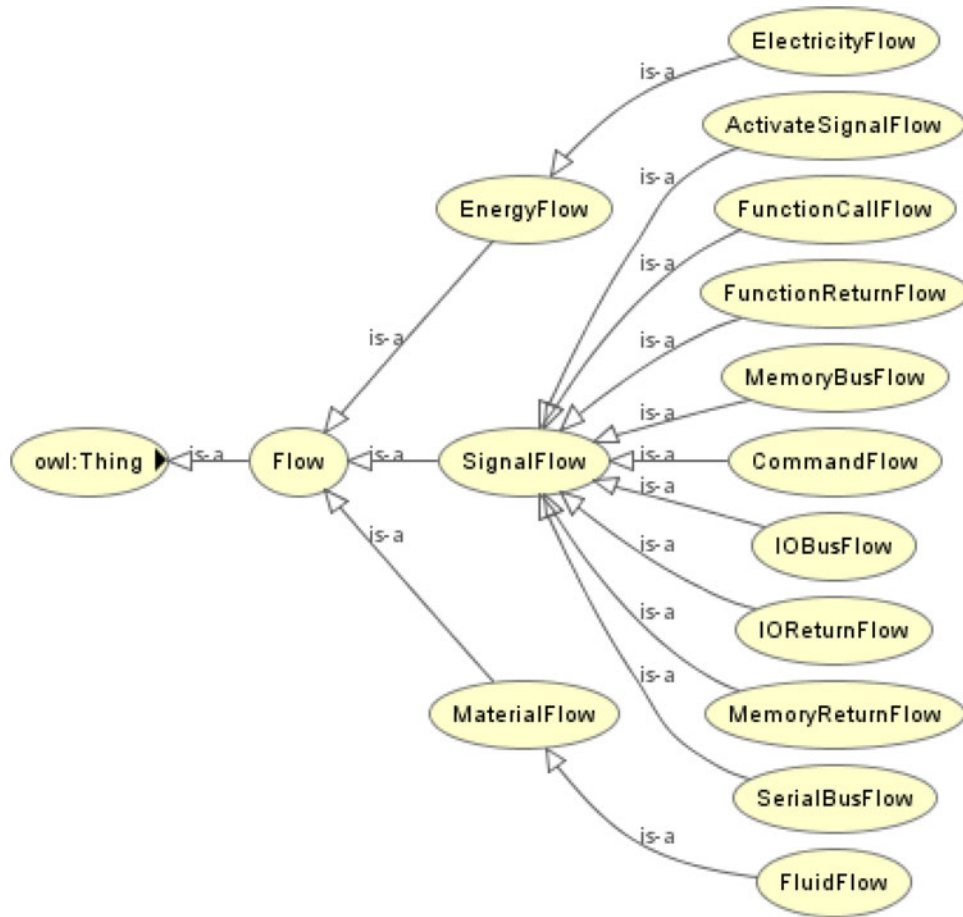


Fig. 4. Hierarchy of the flow ontology.

Table 4. Properties defined in the flow ontology

Property Name	Notations	Property Type	Description
Source	<i>Source</i> (.)	Object (related to the Component Ontology)	<i>Source</i> is the component which sends out the current flow.
Sink	<i>Sink</i> (.)	Object (related to the Component Ontology)	<i>Sink</i> is the component which receives the current flow.
Carrier	<i>Carrier</i> (.)	Object (related to the Component Ontology)	<i>Carrier</i> is the components which the current flow goes through.
Qualities	<i>Qualities</i> (.)	Data (decided by specific qualities)	<i>Qualities</i> are the properties that express particular characteristics of the current flow.
States	<i>States</i> (.)	Object (related to the State Ontology)	<i>States</i> are objects containing the behaviors and triggering conditions. See Section “States and behavioral rules” for details.

is designated as a component, the host component will be in a faulty state once the state of its associated fault becomes “activated.” See Section “Fault ontology” for details.

Table 5. An example command flow defined using the flow ontology

Flow: Command_Flow(CF)		
Properties	Notations	Property Value
Source	<i>Source</i> (CF)	{Program} (a software component)
Sink	<i>Sink</i> (CF)	{Processor} (a hardware component)
Carrier	<i>Carrier</i> (CF)	{Command Bus} (a hardware component)
Qualities	<i>Qualities</i> (CF)	{CommandType (an enumerate of commands), TargetAddress (a positive integer), OperationData (a Hex value)}
States	<i>States</i> (CF)	{Normal, Abnormal} (a set of States)

For different types of components, more specific states are defined to express specific behaviors under such state. Table 9 lists the properties defined for states in the ontological concepts. The content and format of these properties are detailed in the following definitions.

As an example, the nominal states of a processor (defined in Table 3) can be specified as Read Memory State (i.e., transferring data from a memory unit to its register), Add Memory State (i.e., performing addition on its register data and memory data and storing the result in its register), etc., as shown in Figure 7. In the middle of the figure, a state named “IdleState” is defined, which is the default state. We ignore some states because of the

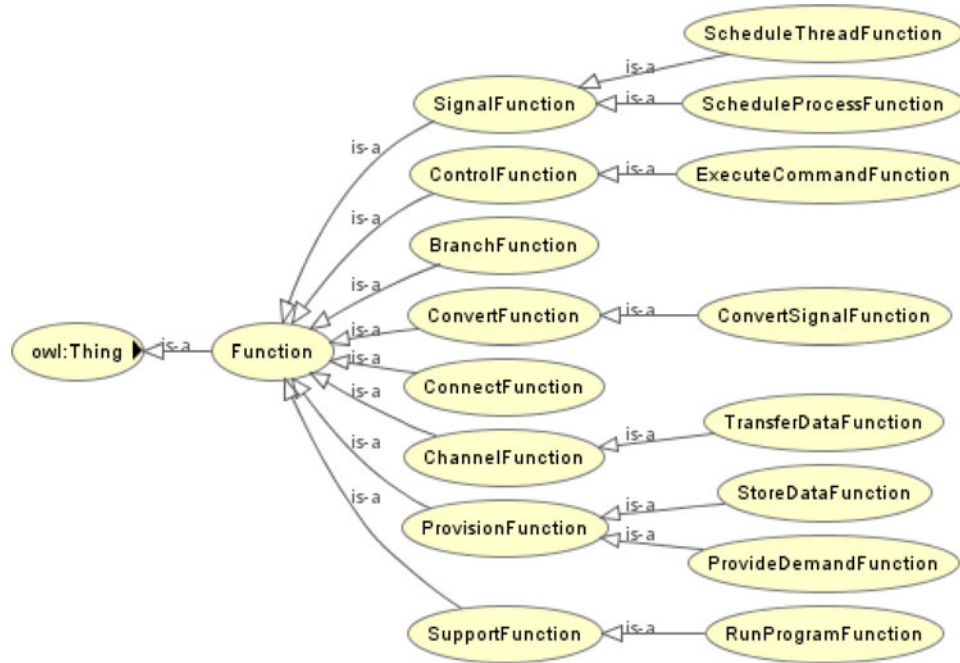


Fig. 5. Hierarchy of the functional ontology.

Table 6. Properties defined in the functional ontology

Property Name	Notations	Property Type	Description
ComposedOf	<i>ComposedOf</i> (·)	Object (related to the Function Ontology)	<i>ComposedOf</i> defines the subfunctions of the current function.
Host Entity	<i>HostEntity</i> (·)	Object (related to the Component Ontology)	<i>Host Entity</i> is the component that implements such function.
Inputs	<i>Inputs</i> (·)	Object (related to the Flow Ontology)	<i>Inputs</i> represent the flows received.
Outputs	<i>Outputs</i> (·)	Object (related to the Flow Ontology)	<i>Outputs</i> represent the flows sent out.
Purpose	<i>Purpose</i> (·)	Data (related to the requirement documents)	<i>Purposes</i> usually is mapped to a statement in system requirements. This property is used when tracing the impact of a fault to system requirement documents.
Qualities	<i>Qualities</i> (·)	Data (decided by specific qualities)	<i>Qualities</i> are the measurable properties that express particular characteristics of the current function.
States	<i>States</i> (·)	Object (related to the State Ontology)	<i>States</i> provides all possible states of the current function associated with triggering conditions. See Section “States and behavioral rules” for details.

Table 7. An example of execute command function defined using the function ontology

Function: ExecuteCommandFunction (EC)		
Properties	Notations	Property Value
ComposedOf	<i>ComposedOf</i> (EC)	{ } (not considered)
Host Entity	<i>Host_Entity</i> (EC)	{Processor} (a Hardware Component)
Inputs	<i>Inputs</i> (EC)	{Command Flow} (a set of Flows)
Outputs	<i>Outputs</i> (EC)	{Memory Bus Flow, I/O Bus Flow} (a set of Flows)
Qualities	<i>Qualities</i> (EC)	{Execution_Time (ms, us, ...)} (a set of Data Properties)
Purpose	<i>Purpose</i> (EC)	{Requirement Statement x.x } (a statement in the system requirement document)
States	<i>States</i> (EC)	{Operating, Degraded, Lost} (a set of States)

Table 8. Model composition

Model type	Composed of
System model	{Component Models, Functional Models}
Functional model	{Individuals of Function, Individuals of Flow}
Component model	{Individuals of Component, Individuals of Flow}

limitation of space. The activated state will change dynamically during fault inference. The inference uses behaviors as evidence to infer the activated state. The triggering conditions and behaviors of each state, appearing in Figure 7, will be explained in the following content.

Definition 4. Behavioral Variables (BVs) denote the qualities involved in a behavior. The expression of a BV usually contains three sections, as shown in the following formula. Assume that the current state is S and the host entity of S is $H = HostEntity(S)$, then a BV can be expressed by:

$$[H].[Inputs(H)|Outputs(H)].[Qualities(Inputs(H)|Outputs(H))].$$

In the formula, the symbol $[H]$ denotes the name of the host entity H ; the second section $[Inputs(H)|Outputs(H)]$ represents the inputs or outputs of the host entity. For example, we know that the inputs and outputs of a component are associated with the component's behaviors. Therefore, a BV of a component can be defined by using its inputs and outputs which are represented by flows. According to the definitions of these properties, this section includes the name of the flow which is an input or an output of a component. The third section $[Qualities(Input(H)|Output(H))]$ is the name of the qualities of the flow defined in

the second section. The following formula defines an example of a BV.

$$MCP.CommandFlow.CommandType.$$

In the formula, the symbol MCP is the name of a multi-core processor defined in Table 3. The symbol $CommandFlow$ defines the input of MCP which is a flow defined in Table 5. Then, the symbol $CommandType$ is one of the qualities of $CommandFlow$.

Definition 5. Behavioral Rules (BRs) are expressions used to describe the relations between BVs. In practice, these expressions are logic expressions containing an equal symbol and/or several operators and BVs. In a behavioral expression, a variable usually appears in the following format. A BR is composed of Boolean expressions (EXP) connected by logical operators (LO). A Boolean expression is composed of terms (TRM) connected by comparison operators (CMP). Furthermore, a term is composed of BVs that are connected by algebraic operators (OP) or bit operators (BO).

$$BR := [EXP] = [EXP][LO][EXP],$$

$$EXP := [TRM][CMP][TRM],$$

$$TRM := [BV][OP][BV],$$

$$LO := [AND|OR|NOT],$$

$$CMP := [> | < | = | \leq | \geq | ! =],$$

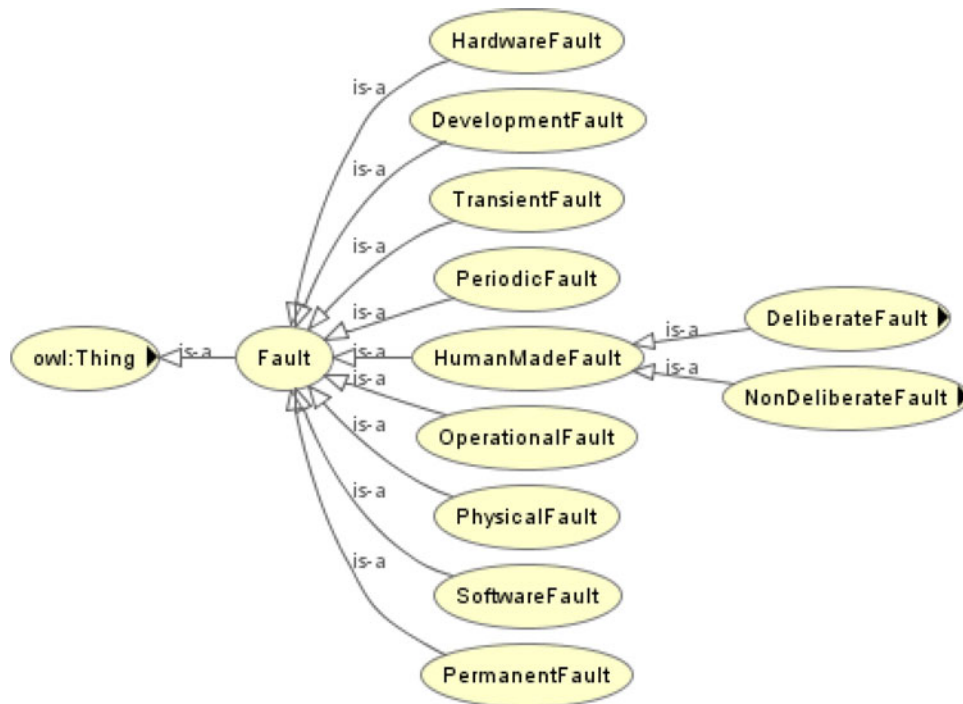


Fig. 6. Hierarchy of the state ontology.

Table 9. Properties defined for states

Property Name	Notations	Property Type	Description
Triggering Condition	<i>TriggeringCondition</i> (·)	Data (Strings representing Triggering Conditions)	<i>Triggering Condition</i> represents the condition for entering such state.
Host Entity	<i>HostEntity</i> (·)	Object (related to the Component/ Flow/Function Ontology)	<i>Host Entity</i> is the object that possesses such state.
Behaviors	<i>Behaviors</i> (·)	Data (Strings representing behavioral rules)	<i>Behaviors</i> contain the actions that the host entity takes to generate outputs in relation to its inputs and states. See the definition of behavioral rules for detail.

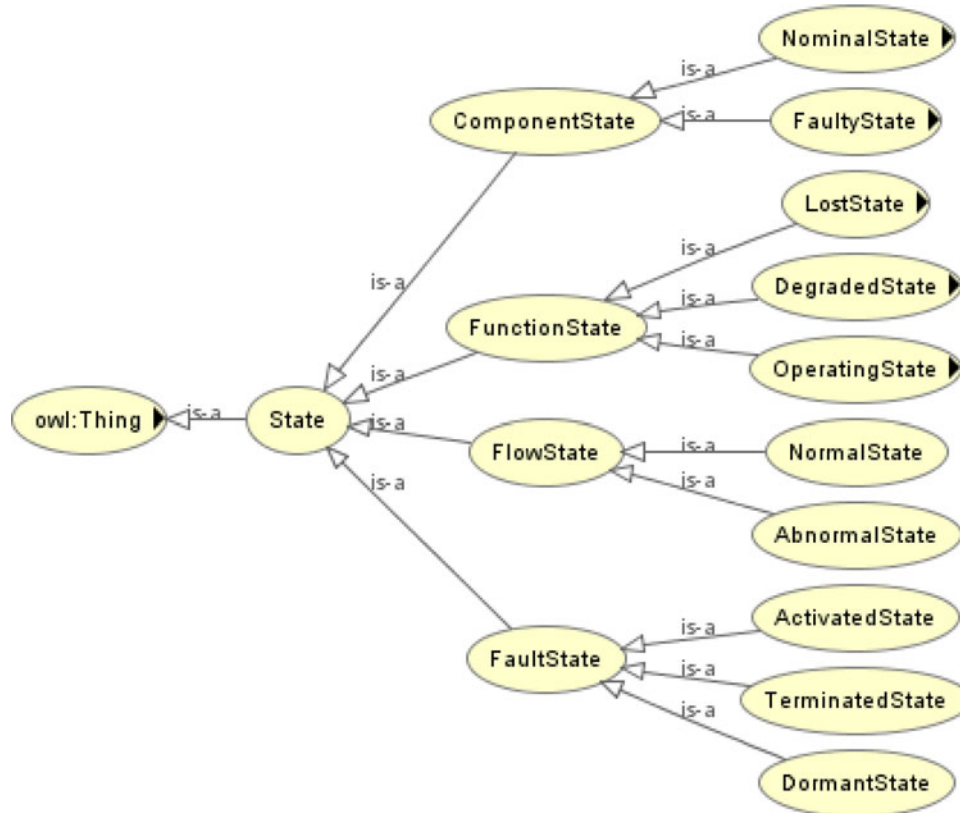


Fig. 7. Partial nominal states of a processor.

$$OP := [+ | - | \times | \div | BO],$$

$$BO := [BIT_AND|BIT_OR|XOR].$$

An example of BR is shown below. This example BR belongs to the “ReadMemoryState” of the multi-core processor class (MCP) defined in Table 3. The “Memory Bus Flow” is one of the outputs of the MCP and the “Command Flow” is one of the inputs of the MCP. Generally, the “Command Flow” is the flow that is sent from a software component to manipulate the action of a processor. The “Memory Bus Flow” is the flow that the processor sends to a memory bus to execute reading or writing operations. From Table 5, we can see that the “Command Type,” the “Target Address,” and the “Operation Data” are the qualities of a “Command Flow.” The “Memory Bus Flow,” whose definition

is not explicitly provided, has the same qualities as the “Command Flow.”

$$\begin{aligned} &MCP.MemoryBusFlow.CommandType \\ &= CMD_READMEM \text{ AND } MCP.MemoryBusFlow. \\ &\quad TargetAddress \\ &= MCP.CommandFlow.TargetAddress \text{ AND} \\ &\quad MCP.MemoryBusFlow.OperationData \\ &= MPC.CommandFlow.OperationData. \end{aligned}$$

According to the expression, the “Command Type” of the “Memory Bus Flow,” which is an output of the component “MCP,” is equal to a command “CMD_READMEM,” which is a predefined constant. Also, the “Target Address” of the “Memory Bus Flow” is equal to the “Target Address” of the

“Command Flow.” The “Operation Data” of the “Memory Bus Flow” is equal to the “Operation Data” of the “Command Flow.” During the early phase of system design, no detailed implementation of functions, components, and flows, or mathematical models representing their behaviors will be available.

Definition 6. Behavioral Rules with Time (BRTs) are time-labeled expressions representing the relations between BRs. The time dimension is added to enable fault inference and study the evolution of the system over time. In BRTs, the time-labeled BVs will be used, which add a time label t_n to the end of the BV expressions. The variable n denotes the current time step. For instance, the example BR with time labels can be defined below.

$$\begin{aligned}
 &MCP.MemoryBusFlow.CommandType.t2 \\
 &= CMD_READMEM \text{ AND } MCP.MemoryBusFlow. \\
 &\quad TargetAddress.t2 \\
 &= MCP.CommandFlow.TargetAddress.t1 \text{ AND } \\
 &\quad MCP.MemoryBusFlow.OperationData.t1 \\
 &= MPC.CommandFlow.OperationData.t1.
 \end{aligned}$$

According to the expression, the “Command Type” of the “Memory Bus Flow” at time step 2 is equal to a type of command “CMD_READMEM.” The “Target Address” of the “Memory Bus Flow” at time step 2 is equal to the “Target Address” of the “Command Flow” at time step 1. The “Operation Data” of the “Memory Bus Flow” at time step 2 is equal to the “Operation Data” of the “Command Flow” at time step 1.

The expression above denotes the relation between two flows at a concrete time step (e.g., t_1 and t_2). However, we usually use BRTs to define the relation at a general level (not for a concrete time step). In that case, we define a time variation expression to represent the time relations. We use the expression $\{[\pm N]\}$ for denoting the time relation. The expression $\{[0]\}$ or $\{[.]\}$ represents the current time step. The expression with a positive number, such as $\{[+1]\}$, means the time step after the current step with the number of steps. An expression with a negative number represents the time step that occurs before the current step with the number of steps. Hence, the example BR can be further defined as below.

$$\begin{aligned}
 &MCP.MemoryBusFlow.CommandType.\{[+1]\} \\
 &= CMD_READMEM \text{ AND } MCP.MemoryBusFlow. \\
 &\quad TargetAddress.\{[+1]\} \\
 &= MCP.CommandFlow.TargetAddress.\{[.]\} \text{ AND } \\
 &\quad MCP.MemoryBusFlow.OperationData.\{[+1]\} \\
 &= MPC.CommandFlow.OperationData.\{[.]\}.
 \end{aligned}$$

Definition 7. Triggering Conditions (TCs) are predicates that map the states and the BVs to a space of true or false. The result of these conditions is used in “if-then” rules to trigger a state transition. Similar to BRTs, the predicates of TCs are logic statements with operators and BVs, except that the TCs generally have a consequent state if the predicate is identified as True. The following formula defines an example of the TCs.

$$\begin{aligned}
 &IF \text{ } MCP.CommandFlow.CommandType.\{[.]\} \\
 &= =CMD_READMEM \text{ AND } MCP.State.\{[.]\} \\
 &= =MCP.State.IdleState \text{ THEN } MCP.State.\{[+1]\} \\
 &= =MCP.State.ReadMemState.
 \end{aligned}$$

According to the formula above, if the “Command Type” of the “Command Flow” is equal to a constant “CMD_READMEM,” and the current state of “MCP” is “Idle,” then the state of “MCP” in the next time step will be the state “ReadMemState.”

Dependencies and restrictions

Dependencies and restrictions exist between the attributes of the proposed ontologies. These dependencies reflect the relations between the ontological concepts. The restrictions defined here allow the framework to automatically detect incorrectness in the model by using ontology solvers. Such automatic check can be important in complex systems to ensure that components are interconnected to achieve the desired system functionalities. Dependency and restriction rules and the corresponding explanations are listed in Table 11 by using the notations defined in Table 10.

Table 11 interprets the constraints applied to the ontological concepts and the dependencies between them. Compliance to these relations guarantees the integrity of the system models. In addition, these rules will play a critical role in fault propagations which will be detailed in the section “Fault propagation inference.”

Modeling example

In this section, we use a simplified module of a computer system with SW and HW components as an example to illustrate the model construction using the ontological concepts. The function of the example subsystem is to provide a demand value to a control system. In this example, we created three individuals of component, four individuals of flow, and three individuals of function, as shown in Figure 8.

The specified values of the properties related to these individuals are detailed in Table 12. Individuals define concrete objects existing in the SUA, which are different from classes that are abstract concepts with constraints and rules. These individuals are subject to the predefined constraints and rules.

When we have the system model with components, flows, and functions, the next step is to generate and inject faults based on the ontological concepts related to faults.

Fault ontology and fault generation

The proposed ontology framework provides fault ontologies to manage the known faults discovered in historical accidents or events and infer new types of faults that have not been discovered.

Table 10. Notations for dependency rules

Notations	Description
CP	A class of component
FL	A class of flow
FC	A class of function
FA	A class of fault (see Section “Fault ontology”)
X, Y, Z	Free variables that could be a class of function, component, or flow
Isa(·)	The type of a class usually is the parent class of the current class

Table 11. Dependencies and restrictions in ontological concepts

No.	Essential Rules	Description
BR01.	$Y \in ComposedOf(X) \rightarrow X \neq Y, (Isa(X) = Component \text{ and } Isa(Y) = Component) \text{ or } (Isa(X) = Component \text{ and } Isa(Y) = Flow) \text{ or } (Isa(X) = Function \text{ and } Isa(Y) = Flow) \text{ or } (Isa(X) = Function \text{ and } Isa(Y) = Function);$	A component may contain several components and flows. A function may contain several subfunctions and flows.
BR02.	$Isa(X) \in \{Component, Function\} \rightarrow Isa(Inputs(X)) = Flow \text{ and } Isa(Outputs(X)) = Flow;$	The inputs and outputs of a component or a function should be flows.
BR03.	$Isa(Source(FL)) \in \{Component, Function\}, Isa(Sink(FL)) \in \{Component, Function\};$	The source and sink of a flow should be functions or components.
BR09.	$\forall CP, Isa(Purpose(CP)) = Function;$	The purpose of a component should be a function.
BR04.	$\forall FL, Isa(Carrier(FL)) = Component;$	The carrier of a flow should be a component.
BR05.	$\forall CP, \forall FC, FC \in Purpose(CP) \rightarrow Inputs(CP) = Inputs(FC) \text{ and } Outputs(CP) = Outputs(FC);$	A component will have the same inputs and outputs with the function that is the purpose of such component or flow.
BR06.	$X = Source(FL) \rightarrow FL \in Outputs(X);$	The source of a flow is the object whose outputs should contain the flow.
BR07.	$X = Sink(FL) \rightarrow FL \in Inputs(X);$	The sink of a flow is the object whose inputs should contain the flow.

The new fault inference is implemented by applying the fault generation principles introduced in the section to the properties defined by the component ontologies. This section details the concepts and properties related to fault ontologies and fault generation.

Fault ontology

Fault ontology allows the ontological framework to represent and generate various sorts of faults that may be introduced at design, development, and operation phases. In the perspective of system engineering (Avizienis et al., 2004b), an error is “the state of the system that deviates from the correct service state.” A fault is defined as: “An adjudged or hypothesized cause of an error.” System failure is “an event that occurs when the delivered service deviates from correct service.” A fault can arise from any phase of the life cycle of a product and can lead to erroneous states that may culminate into failures.

In prior research, faults have been classified through various perspectives, such as dependability (Avizienis et al., 2004a, 2004b), scientific workflow (Lackovic et al., 2010), and service-oriented architecture (Brüning et al., 2007; Hummer, 2012). In this paper, we synthesize the existing taxonomies for faults and establish the hierarchy of fault ontology shown in Figure 9. It is worth noting that the child nodes of the “fault” node in Figure 9 may not be defined in terms of the same perspective. For example, we defined the nodes of “software fault,” “hardware fault,” “development fault,” and “operational fault” as the children of the “fault” node, but the software and hardware faults are distinguished by domain, whereas the development and operational faults are classified by the phase during which the fault was introduced. A specific fault class will be linked to the corresponding node when building fault ontologies. For example, a “bit flip fault” of a processor register can be designated as a child node of the hardware fault and the operational fault.

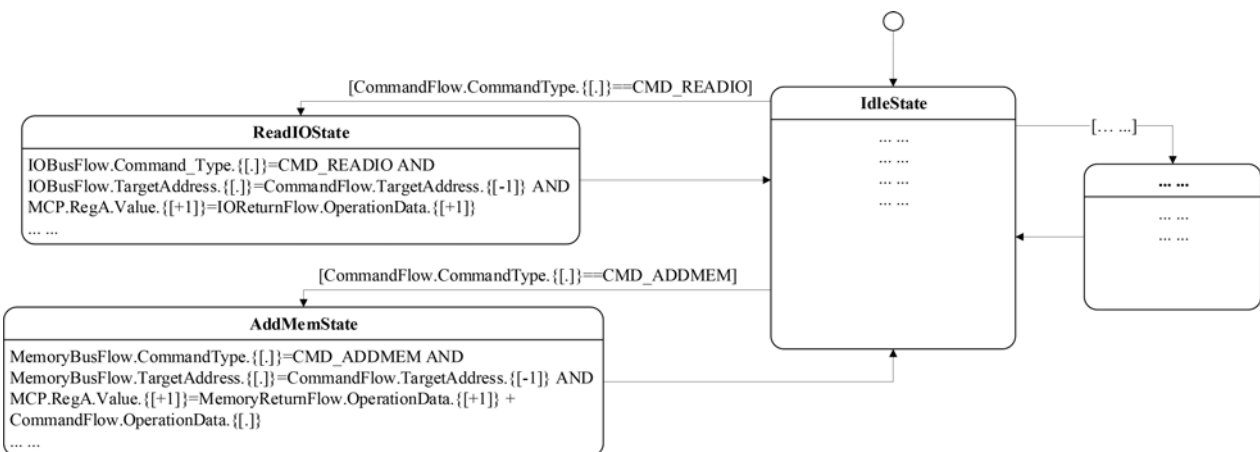


Fig. 8. System model of the example system.

Table 12. Property values of components and functions in the example system

Individuals	Classes	Properties and Values
Provide_Demand (RDF)	TransferDataFunction (is a Function)	ComposedOf(RDF)={...} HostEntity(RDF)={RDP} Inputs(RDF)={ReadDemand_CommandReturnFlow} Outputs(RDF)={ReadDemand_CommandFlow} Purpose(RDF)={"Req.XX.XX"} ^a Qualities(RDF)={...} States(RDF)={RDFOperatingState, RDFLostState, RDFUnknownState}
Execute_Command (ECF)	TransferDataFunction (is a Function)	ComposedOf(ECF)={...} HostEntity(ECF)={CPU_0} Inputs(ECF)={ ReadDemand_CommandFlow, ReadDemand_IOReturnFlow } Outputs(ECF)={ReadDemand_CommandReturnFlow, ReadDemand_IOBusFlow} Purpose(ECF)={"Req.XX.XX"} ^a Qualities(ECF)={...} States(ECF)={ECFOperatingState, ECFLostState, ECFUnknownState}
Store_Demand (SDF)	StoreDataFunction (is a Function)	ComposedOf(SDF)={...} HostEntity(SDF)={DUD} Inputs(SDF)={ReadDemand_CommandReturnFlow} Outputs(SDF)={ReadDemand_CommandFlow} Purpose(SDF)={"Req.XX.XX"} ^a Qualities(SDF)={} States(SDF)={SDFOperatingState, SDFLostState, SDFUnknownState}
ReadDemand_Program (RDP)	Program (is an SW Component)	ComposedOf(RDP)={...} Location(RDP)={Running_on(CPU_0)} Inputs(RDP)={ReadDemand_CommandReturnFlow} Outputs(RDP)={ReadDemand_CommandFlow} Purpose(RDP)={RDV} Qualities(RDProgram)={} States(RDProgram)={RDPRunningState, RDPIdleState}
CPU_0	MCP (is an HW Component)	ComposedOf(CPU_0)={...} Location(CPU_0)={...} Inputs(CPU_0)={ReadDemand_CommandFlow, ReadDemand_IOReturnFlow} Outputs(CPU_0)={ReadDemand_CommandReturnFlow, ReadDemand_IOBusFlow} Purpose(CPU_0)={ECF} Qualities(CPU_0)={...} States(CPU_0)={CPU0IdleState, CPU0ReadMemState, CPU0WriteMemState, CPU0ReadIOState, CPU0WriteIOState, CPU0AddMemState, ...}
DiskUnit_Demand (DUD)	DiskUnit (is an HW Component)	ComposedOf(DUD)={...} Location(DUD)={...} Inputs(DUD)={ReadDemand_IOBusFlow} Outputs(DUD)={ReadDemand_IOReturnFlow} Purpose(DUD)={SDF} Qualities(DUD)={...} States(DUD)={DUDIdleState, DUDReadState, DUDWriteState}

^a The requirement of the function is not shown in this example.

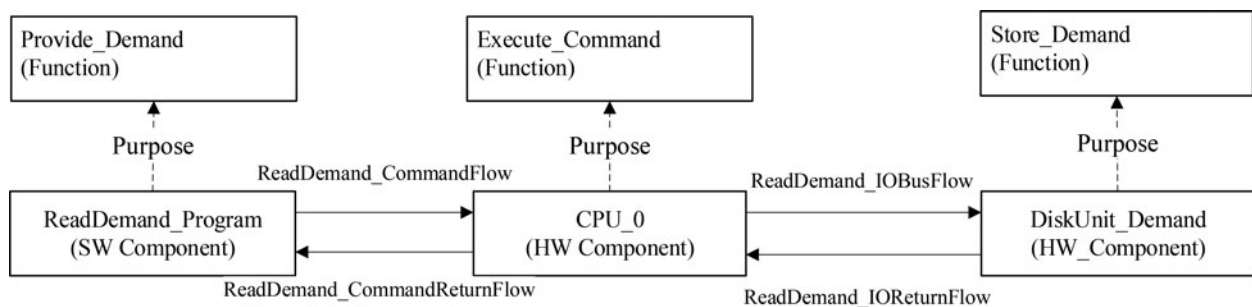


Fig. 9. Hierarchy of fault ontology.

Table 13. Properties defined in the fault ontology

Property Name	Notations	Property Type	Description and Possible Values
Host Entity	<i>HostEntity</i> (·)	Object (related to the Component Ontology)	<i>Host Entity</i> is the component where the current fault is located.
Origin	<i>Origin</i> (·)	Data (Strings used by the fault injection algorithm)	<i>Fault Origin</i> identifies the cause of the fault, which can assist in the identification of whether the current fault can be applied to the SUA. Generally, a fault can be introduced due to human errors or natural conditions, such as technologies, materials, or facilities used to create the product, as well as the physical environment interacting with the product during system operation. The existence of such factors allows the proposed framework to generate appropriate faults based on a knowledge base.
Phase of Introduction	<i>POI</i> (·)	Data (Strings used by the simulation engine)	<i>Phase of Introduction</i> is the phase when a fault was introduced into the system. It can be “development” or “operation.”
Occurrence	<i>Occurrence</i> (·)	Data (Strings used by the simulation engine)	<i>Occurrence</i> defines the time characteristics of a fault. Faults can be categorized into transient faults, periodic faults, and permanent faults. Transient faults occur unpredictably at random moments within the components of a system. Periodic faults occur repeatedly with the same time intervals. Permanent faults are the faults that usually occur one time and lead to permanent errors. This type of fault will change the states or behaviors of a component immediately and thus are apt to be detected relatively easily.
Triggering Condition	<i>TCond</i> (·)	Data (Strings of triggering condition expressions)	<i>Triggering Condition</i> denotes the ways to activate a fault. The faults, whether introduced at the requirement, design, or development phase, can be activated during system testing, manufacturing, or operation. The triggering condition encompasses three important ingredients: (1) the specific configuration(s) that the system can be in for the fault to be triggered; (2) the operation(s), i.e., the series of behaviors that the system can perform for the fault to be triggered; and/or (3) which dependencies and other events must occur for the fault to be triggered.
Impact Direction	<i>IDir</i> (·)	Data (Strings used by the simulation engine)	<i>Impact Direction</i> is categorized into upstream, downstream, and self. This property determines the impacted property of the host entity. Faults with “upstream” impact direction will change the inputs of their host entities; “downstream” impact direction faults will change the outputs of their host entities. An impact direction of “self” means that faults will change the behaviors of their host entities’ sub-entities.
Effects	<i>Effects</i> (·)	Object (related to State Ontology)	<i>Effect</i> of a fault is that the host entity is in an erroneous state. Abnormal behaviors of the host entity will be defined for the erroneous state. For fault inference, the information provided by the properties “Effect” and “Triggering Condition” will be combined with the property “state” of the host entity to mimic the activation and propagation of such fault.
States	<i>States</i> (·)	Object (related to Fault State Ontology)	<i>States</i> of a fault can be predefined as dormant, activated, or terminated. The meaning of the above states can be taken literally. Dormant faults are faults residing in a system or component that have not been activated; activated faults are faults that continually or periodically affect the working states of components or systems. The state of a fault may change to a terminated state when the fault is isolated or fixed.

Due to the complexity of fault causes and effects, several properties are defined to represent the factors involved in fault generation and propagation. Table 13 outlines the properties considered in the proposed method.

Table 14 illustrates the mapping relation between the existing fault taxonomies and the ontological concepts proposed by this research. We use (Avižienis et al., 2004b) as a representative research for comparison, where faults can be classified according to eight perspectives.

Restrictions on fault ontologies

By reusing the notations in Table 10, Table 15 shows the restrictions in the proposed fault ontologies. These restrictions are consistent with the existing research and fault taxonomies.

Adding known faults to fault ontologies

When a fault is observed in accidents or event reports, the observed fault can be recorded by the proposed fault ontology. The process of adding known faults to the fault ontologies consists of the following steps, which are displayed in Figure 10. The process is usually completed manually.

- (1) Define the name of the fault object based on the event report or repository describing the fault. Figure 10 demonstrates this process by using an example fault, the bit flip fault of a register in a computer processor. In the example, a compact name “RegisterBitFlipFault” is defined to describe the characteristics of the fault.
- (2) Define the properties of the added fault based on the properties provided by the fault ontology. As shown in the example, the host entity of the fault should be a processor, the phase of introduction should be defined as operation, etc.
- (3) Add the new fault to the fault ontologies, linking the new fault to the ones in the fault ontologies. According to expert knowledge, a “RegisterBitFlipFault” is a hardware fault and an operational fault, as shown in Figure 10.

Fault generation principles

Besides adding known faults to the fault ontologies, this paper develops a set of principles to generate new types of faults that may not have been observed historically. Since a fault is an object that may occur inside or outside a component, fault generation, in this paper, is the process of applying the fault generation principles to the properties of component ontologies and generating

Table 14. Mapping relation between the proposed fault taxonomy and the existing fault taxonomies

Existing Perspectives of Fault Classification	Contents	Fault Ontology Concepts	Contents
Phase of Creation or Occurrence	<ul style="list-style-type: none"> Development Faults Operational Faults 	Phase of Introduction	<ul style="list-style-type: none"> Development Faults Operational Faults
Objective	<ul style="list-style-type: none"> Malicious Non-Malicious 	Fault Origin	<ul style="list-style-type: none"> Physical Faults Human-Made Faults <ul style="list-style-type: none"> Non-Deliberate Faults Accidental Faults Incompetence Faults <ul style="list-style-type: none"> Deliberate Faults Non-Malicious Faults Malicious Faults
Intent	<ul style="list-style-type: none"> Deliberate Faults Non-Deliberate Faults 		
Phenomenological Cause	<ul style="list-style-type: none"> Natural Faults Human-Made Faults 		
Capability	<ul style="list-style-type: none"> Accidental Faults Incompetence Faults 		
System boundaries	<ul style="list-style-type: none"> Internal Faults External Faults 	Host Entity	The faulty component (inside or outside a system)
Dimension	<ul style="list-style-type: none"> Hardware Software 	Domain	<ul style="list-style-type: none"> Hardware Software
Persistence	<ul style="list-style-type: none"> Permanent Transient 	Occurrence	<ul style="list-style-type: none"> Permanent Periodic Transient
		Effects	<ul style="list-style-type: none"> Faulty States of Host Entities
		States	<ul style="list-style-type: none"> Dormant Activated Terminated

new faults that may affect the behaviors of HW and SW components. The fault generation principles can establish faulty states for components based on the properties defined by their ontologies. These faulty states with behavioral rules (BRs) will participate in the fault propagation inference to generate the fault propagation paths throughout the SUAs. Since the BRs are expressions containing properties and BVs, these principles can modify these elements in the BRs to deviate the behaviors of the target object from their normal states. The fault generation method expands the fault analysis scenarios beyond existing observed faults. This enables a system designer to discover unknown/unforeseen situations, or designing a system to be more robust and reliable. The fault generation principles are detailed as below by using the notations in Tables 10 and 16.

Category 1. Missing Property Principles define the rules to generate faults where a statement of a component’s behavior defined by the ontological concepts is missing. For example, a routine

pertaining to a software program is forgotten by the system designer. To generate this type of fault, the effect of such a fault complies with the following rules: (1) if the target behavior is an expression of behavior, the expression will be removed; (2) if the target behavior is a BV, all the expressions related to that BV will be removed. Table 17 displays the fault generation principle for missing property faults.

Category 2. Additional Property Principles define the rules to generate faults where an extra behavior of a component is injected into a state of that component. Several rules can be applied when adding new behaviors to a component. For example, a new disturbing BV can be added to a component and can be inserted into all the BRTs with an operator (e.g., addition). Table 18 reflects the general triggering conditions and effects related to different types of faults. The selection of the new entities added to the system depends on the configuration of the system.

Table 15. Restrictions in fault ontologies

No.	Constraints	Fault Generation Principles
FR01.	$Host_Entity(FA) \in \{Component\};$	The host entity of a fault can only be a component.
FR02.	$Fault_Origin(FA) \in \{Nature, Human\};$	Fault origin can be nature, human, or any subtype of these two mentioned in the fault origin taxonomy.
FR03.	$Phase_of_Introduction(FA) \in \{Development, Operation\};$	A fault can be introduced into a system during the development phase (e.g., a software defect) or during the operational phase (e.g., a pipe leakage).
FR04.	$Phase_of_Introduction(FA) = Development \rightarrow Fault_Origin(FA) = Human;$	All faults occurring during the development phase are caused by human errors.
FR05.	$Occurrence(FA) \in \{Transient, Periodic, Permanent\}$	A fault’s occurrence category can be transient, periodic, or permanent.
FR06.	$Impact_Direction(FA) \subseteq \{Upstream, Downstream, Self\}$	The impact direction of a fault can be upstream, downstream, or both.
FR07.	$States(FA) \subseteq \{Dormant, Activated, Terminated\}$	A fault’s state can be dormant, activated, or terminated.

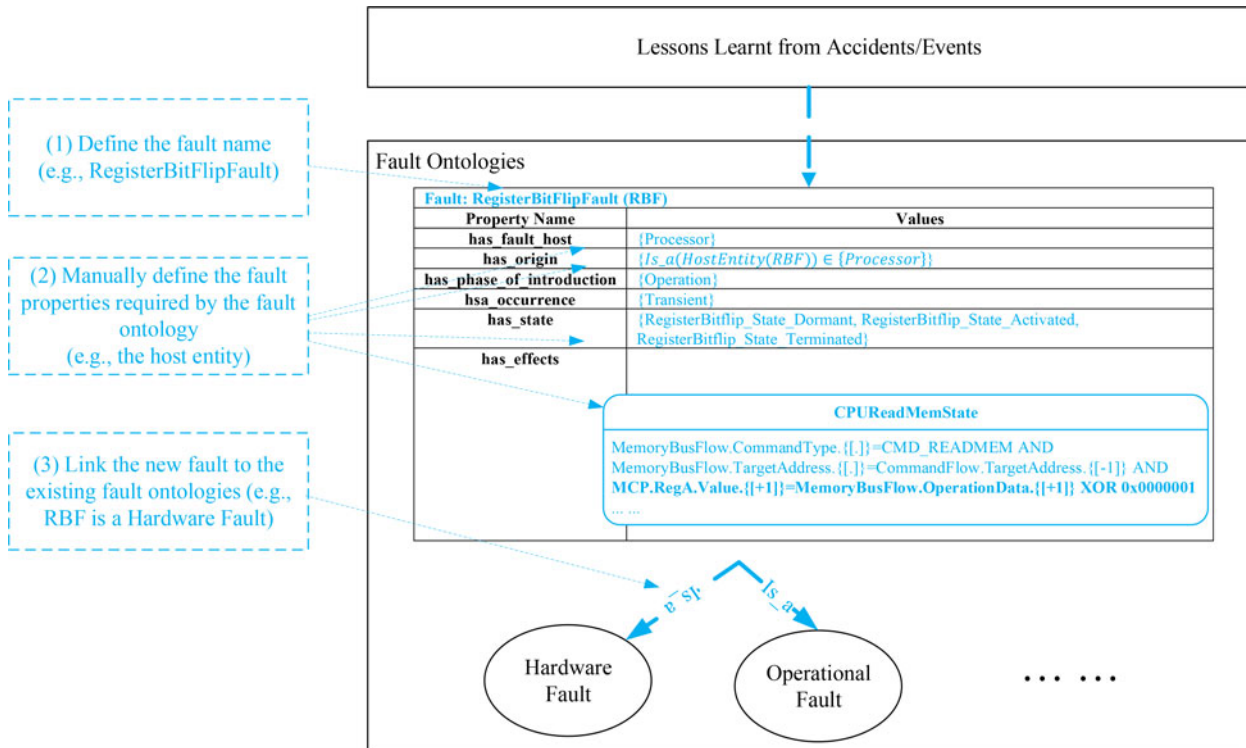


Fig. 10. The process of adding a known fault to the fault ontologies.

Table 16. Notations for representing fault generation principles

Notations	Description
X_o, Y_o, Z_o	Free variables that could be an original class of function, component, or flow selected for fault injection
X_f, Y_f, Z_f	Free variables that could be the corresponding class of X_o, Y_o, Z_o with generated faults
Q_o	A data property of X_o, Y_o, Z_o For example, $Q_o \in Qualities(CP)$, Q_o represents a quality property of a component
Q_f	The corresponding property of Q_o under faulty states
S_o	A state class related to X_o, Y_o, Z_o For example, $S_o \in States(CP)$, S_o represents a state of a component
S_f	The corresponding faulty state of S_o
KB	The knowledge base with all the classes of components, flows, and functions
$brs(\cdot)$	A set of BRs related to an object or a property For example, $brs(Q_o)$ represents the BRs containing the data property Q_o
$Value_t(\cdot)$	The value of an object or a property at time t For example, $Value_t(Q_o)$ represents the value of the property Q_o at time t

Category 3. *Incorrect Property Principles* define the rules to generate faults where an existing BRT in the nominal state of a component is modified. The modification can be a change of an operator (e.g., change a “+” to a “-”). This process is like applying a mutation operator to BRs which are analogous to software source code. Table 19 displays the fault generation principles for incorrect property faults. Selecting which entities to replace depends on the configuration of the system, and, currently, human interaction is required to make this selection.

Table 20 summarizes the faults obtained when applying the fault generation principles to a software routine. In the table, generic descriptions are given to summarize the generated faults.

Fault generation process

The process of fault generation is interpreted by Figure 11 which uses the multi-core processor component as an example. It is worth noting that the fault analysis framework can automatically implement the fault generation process. The process encompasses the following steps.

- (1) Iterate and select components in the component ontologies. As shown in Figure 11, the component “Multi_Core_Processor” is selected, which is a “Processor.”

Table 17. Fault generation principle for missing property faults

No.	Applicable Properties	Fault Generation Principles	Description
MP01.	ComposedOf	$\forall X \in ComposedOf(CP), S_o \in States(CP) \rightarrow \exists FA, CP$ $= HostEntity(FA) \cap S_f \in Effect(FA) \cap brs(X) \notin Behaviors(S_f);$	A composition of the faulty component will be removed. Also, all the behaviors of the removed composition will be removed.
MP02.	Location	$\forall X \in Location(CP), S_o \in States(CP) \rightarrow \exists FA, CP$ $= HostEntity(FA) \cap S_f \in Effect(FA) \cap brs(Inputs(X))$ $\notin Behaviors(S_f) \cap brs(Outputs(X)) \notin Behaviors(S_f);$	A location relation of the faulty component will be removed. Also, all the related inputs and outputs will be removed.
MP03.	Inputs	$\forall X \in Inputs(CP), S_o \in States(CP) \rightarrow \exists FA, CP$ $= HostEntity(FA) \cap S_f \in Effect(FA) \cap brs(Inputs(X)) \notin Behaviors(S_f);$	One or more input ports of the faulty component will be removed.
MP04.	Outputs	$\forall X \in Outputs(CP), S_o \in States(CP) \rightarrow \exists FA, CP$ $= HostEntity(FA) \cap S_f \in Effect(FA) \cap brs(Outputs(X)) \notin Behaviors(S_f);$	One or more output ports of the faulty component will be removed.
MP05.	Purpose	$\forall X \in Purpose(CP), S_o \in States(CP) \rightarrow \exists FA, CP$ $= HostEntity(FA) \cap S_f \in Effect(FA) \cap brs(X) \notin Behaviors(S_f);$	The purpose of the faulty component will be removed.
MP06.	Qualities	$\forall Q_o \in Qualities(CP), S_o \in States(CP) \rightarrow \exists FA, CP$ $= HostEntity(FA) \cap S_f \in Effect(FA) \cap brs(Q_o) \notin Behaviors(S_f);$	The behaviors related to the missing qualities will be removed.
MP07.	States	$\forall S_o \in States(CP) \rightarrow \exists FA, CP$ $= HostEntity(FA) \cap S_f \in Effect(FA) \cap Behaviors(S_o) \notin Behaviors(S_f);$	One or more behaviors of the faulty component will be removed.

- (2) Iterate and select properties of the component under consideration. In the figure, one of the “inputs” properties, “Command Flow,” is selected.
- (3) Apply the fault generation principles to the selected properties. In this example, the missing inputs rules defined in Table 17 is applied to the “Command Flow” and correspondingly a new fault object “Processor Missing Input Command Flow Fault” is added to the fault ontology.
- (4) Define the properties of the new fault object. The fault analysis framework can automatically define a portion of the new

fault’s properties, such as the host entity and the effects of the fault. For example, the effects of the generated “Processor Missing Input Command Flow Fault” contain the impacted state “Read Memory State” which includes an abnormal behavior “MemoryBusFlow.TargetAddress.[.] = NULL.” This behavior rule derives from the original rule “MemoryBusFlow.TargetAddress.[.] = CommandFlow.TargetAddress.[-1].” Since the input “Command Flow” is selected as the missing property in this example, all the variables related to this input are assigned an invalid value “NULL.”

Table 18. Fault generation principle for additional property faults

No.	Applicable Properties	Fault Generation Principles	Description
AP01.	ComposedOf	$\forall CP \in KB \rightarrow \exists FA, CP = HostEntity(FA) \cap S_f$ $\in Effect(FA) \cap (\exists X, X \notin ComposedOf(CP) \cap brs(X) \in Behaviors(S_f));$	The behaviors of a new composite will be added into the state of the faulty component.
AP02.	Location	$\forall CP \in KB \rightarrow \exists FA, CP = HostEntity(FA) \cap S_f$ $\in Effect(FA) \cap (\exists X \notin Location(CP) \cap brs(Inputs(X))$ $\in Behaviors(S_f) \cap brs(Outputs(X)) \in Behaviors(S_f));$	The behaviors of a new location relation will be added into the state of the faulty component.
AP03.	Inputs	$\forall CP \in KB \rightarrow \exists FA, CP = HostEntity(FA) \cap S_f$ $\in Effect(FA) \cap (\exists X \notin Inputs(CP) \cap brs(X) \in Behaviors(S_f));$	The behaviors related to a new input will be added to the faulty component.
AP04.	Outputs	$\forall CP \in KB \rightarrow \exists FA, CP = HostEntity(FA) \cap S_f$ $\in Effect(FA) \cap (\exists X \notin Outputs(CP) \cap brs(X) \in Behaviors(S_f));$	The behaviors related to a new output will be added to the faulty component.
AP05.	Purpose	$\forall CP \in KB \rightarrow \exists FA, CP = HostEntity(FA) \cap S_f$ $\in Effect(FA) \cap (\exists X \notin Purpose(CP) \cap brs(X) \in Behaviors(S_f));$	The behaviors related to a new function will be added to the faulty component.
AP06.	Qualities	$\forall CP \in KB \rightarrow \exists FA, CP = HostEntity(FA) \cap S_f$ $\in Effect(FA) \cap (\exists Q_f, Q_f \notin Qualities(CP_o) \cap brs(Q_f) \in Behaviors(S_f));$	The behaviors related to an additional quality will be added to the faulty component.
AP07.	States	$\forall CP \in KB \rightarrow \exists FA, CP = HostEntity(FA) \cap S_f$ $\in Effect(FA) \cap \exists S_f, S_f \notin States(CP) \cap Behaviors(S_f) \notin Behaviors(CP);$	A new state will be added to the faulty component. The behaviors in the new state is different from the ones in the original component.

Table 19. Fault generation principle for incorrect property faults

No.	Applicable Properties	Fault Generation Principles	Description
IP01.	ComposedOf	$\forall X \in ComposedOf(CP), S_o \in States(CP) \rightarrow \exists FA, CP$ $= HostEntity(FA) \cap S_f \in Effect(FA) \cap brs(X)$ $\notin Behaviors(FA) \cap (\exists Y, Isa(X) \neq Isa(Y) \cap Y$ $\notin ComposedOf(CP) \cap brs(Y) \in Behaviors(S_f));$	The behaviors of an existing component will be replaced by the ones of some other components with different types.
IP02.	Location	$\forall X \in Location(CP), S_o \in States(CP) \rightarrow \exists FA, CP$ $= HostEntity(FA) \cap S_f \in Effect(FA)$ $\cap (\exists Y, Y \notin Location(CP) \cap brs(Inputs(Y)) \in Behaviors(S_f)$ $\cap brs(Outputs(Y)) \in Behaviors(S_f)$ $\cap brs(Inputs(X)) \notin Behaviors(S_f) \cap brs(Outputs(X)) \notin Behaviors(S_f));$	The behaviors of the existing location relations of the faulty component will be replaced by the ones related to some other location relations.
IP03.	Inputs	$\forall X \in Inputs(CP), S_o \in States(CP) \rightarrow \exists FA, CP$ $= HostEntity(FA) \cap S_f \in Effect(FA)$ $\cap (\exists Y, Y \notin Inputs(CP) \cap brs(Y) \in Behaviors(S_f)$ $\cap brs(X) \notin Behaviors(S_f) \cap (Isa(X) \neq Isa(Y) \cup (Isa(X)$ $= Isa(Y) \cap \exists t, Value_t(Qualities(X)) \neq Value_t(Qualities(Y)))));$	The behaviors of an existing input will be replaced by the ones related to another input with a different type or a different value of a quality.
IP04.	Outputs	$\forall X \in Outputs(CP), S_o \in States(CP) \rightarrow \exists FA, CP$ $= HostEntity(FA) \cap S_f \in Effect(FA)$ $\cap (\exists Y, Y \notin Outputs(CP)$ $\cap brs(Y) \in Behaviors(S_f) \cap brs(X) \notin Behaviors(S_f)$ $\cap (Isa(X) \neq Isa(Y) \cup (Isa(X) = Isa(Y) \cap \exists t, Value_t(Qualities(X))$ $\neq Value_t(Qualities(Y)))));$	The behaviors of an existing output will be replaced by the ones related to another output with a different type or a different value of a quality.
IP05.	Qualities	$\forall Q_o \in Qualities(CP), S_o \in States(CP) \rightarrow \exists FA, CP$ $= HostEntity(FA) \cap S_f \in Effect(FA)$ $\cap (\exists Q_f, Q_f \notin Qualities(CP)$ $\cap brs(Q_f) \in Behaviors(S_f) \cap ((Isa(Q_f) \neq Isa(Q_o)) \cup (Isa(Q_f)$ $= Isa(Q_o) \cap \exists t, Value_t(Q_f) \neq Value_t(Q_o)))));$	The behaviors related to a quality of the faulty component deviate from the original ones.
IP06.	Purpose	$\forall X \in Purpose(CP), S_o \in States(CP) \rightarrow \exists FA, CP$ $= HostEntity(FA) \cap S_f \in Effect(FA)$ $\cap (\exists Y, Y \notin Purpose(CP) \cap brs(Y)$ $\in Behaviors(S_f) \cap brs(X) \notin Behaviors(S_f));$	The behaviors related to the purpose of the faulty component will be replaced by the ones related to another purpose.
IP07.	States	$\forall S_o \in States(CP) \rightarrow \exists FA, CP$ $= HostEntity(FA) \cap S_f \in Effect(FA) \cap Behaviors(S_o)$ $\neq Behaviors(S_f);$	One or more behaviors of a state of the faulty component will be replaced by different ones.

The generated faults will be introduced into the SUA in the fault injection process and their impacts on the SUA will be inferred during fault propagation analysis.

Fault injection

Faults are injected into the system model before fault propagation inference. Fault injection is the process that decides on the fault types and locations at which faults will occur in SUAs and injects the abnormal behaviors of these components in faulty states into the SUAs. Based on the properties of the fault ontologies introduced in this section, the fault injection process can automatically select potential faults of SUAs and inject them to the possible occurrence locations. As shown in Figure 12, the fault injection process consists of the following steps: (1) *fault selection*, select appropriate types of faults from the fault ontologies based on the information related to components in the SUA; (2) *individual creation*, create an instance of the selected type of fault and specify

the properties related to the individual; and (3) *state replacement*, replace the states of the host entity in terms of the states defined by the effect property of the fault individual. The replaced state with the abnormal behavior will be involved in the fault propagation inference which establishes a fault propagation path. The following subsections explain these steps.

Fault selection

Fault selection is the process to select appropriate types of faults from the fault ontology. In the fault ontology, the host entity is the property that assists in the identification of whether the current fault can be applied to the SUA. As an example shown in Figure 12, the first task of fault selection (task 1.1) is to iterate on the system model and select components for fault injection. The component individual “CPU_0,” which is an instance of a processor (identified in task 1.2), is selected. Then, task 1.3 searches the fault ontologies for the faults that will occur in a processor. In this case, the

Table 20. Fault generation for a software routine

Fault Type	Fault Description	Fault Injection Implementation
Missing ComposedOf	The designer forgets to define a data structure in the routine.	The behaviors related to the faulty data structure will be removed.
Missing Inputs	The designer forgets to define one or more input parameters.	The behaviors related to the input parameters will be invalid.
Missing Outputs	The designer forgets to define one or more output parameters.	The behaviors related to the output parameters will be invalid.
Missing Locations	The designer forgets to call this routine (Dynamic Location). The developer forgets to write this routine to the file (Static Location).	The call of this routine will be removed from the original program. The code of this routine will be removed from the original file.
Missing Qualities	The designer did not regulate the routine execution time.	A long delay will be added to the original routine.
Missing States	The designer forgets to consider a possible state.	The missing state with its triggering condition will be removed.
Missing Purposes	Not Applicable	Not Applicable
Additional ComposedOf	The designer defines an extra variable in the routine.	An extra variable “dummy_var” will be added to the routine.
Additional Inputs	The designer defines an extra input parameter to the routine.	An extra input “dummy_input” will be added to the routine.
Additional Outputs	The designer defines an extra output parameter to the routine.	An extra output “dummy_output” of the type “signal” will be added to the routine.
Additional Locations	The routine is abnormally called twice.	This routine will be unexpectedly called at another point in the program.
Additional Qualities	A new erroneous quality is defined.	A dummy data structure will be added to the routine for the extra memory occupation.
Additional States	The designer defines an extra state.	An extra state “dummy_state” with triggering conditions will be added to the routine.
Additional Purposes	The designer creates malicious codes in the routine.	An extra function will be added to the purpose of this routine.
Incorrect ComposedOf	The type of the data belonging to the routine is wrong.	An original data structure will be replaced.
Incorrect Inputs	The type or value of input parameters is wrong.	An original input variable will be changed.
Incorrect Outputs	The type or value of output parameters is wrong.	The original output variable will be changed.
Incorrect Locations	The routine is called at a wrong place.	The calling position of this routine will be changed.
Incorrect Qualities	The execution time exceeds the design value.	A delay will be added to this routine.
Incorrect States	The state is not correctly defined.	The behaviors and triggering conditions of the states will be changed.
Incorrect Purposes	The designer misunderstands the requirement.	The purpose of this routine links to another function.

“RegisterBitFlipFault” (RBF) is located by the fault injection algorithm since the host entity of the RBF is a processor.

Individual creation

Once the object of the fault has been identified, the fault analysis framework creates an individual of such type of fault (task 2.1) and specifies the properties related to the fault class (task 2.2). In Figure 12, a fault individual “RBF_CPU_0” is created which effects include the “Read Memory State” with abnormal behaviors.

State modification

In this step, the original states of the target component will be replaced by the states defined in the “Effects” property of the created fault individual. By doing this, a new system model is generated which contains the components with faulty states. During the fault propagation inference, these injected faulty states will be activated and the corresponding behavioral rules will be executed. In the example shown in Figure 12, the target component of the fault

individual “RBF_CPU_0” is located by referring to its host entity (CPU_0) in task 3.1 and then by replacing the original states of the original component individual “CPU_0” by the effects of the fault individual. In detail, the original state “Read Memory State” of the component “CPU_0” contains the normal behavior rule “MCP.RegA.Value.{{+1}} = MemoryReturnFlow.OperationData.{{+1}}” (shown in the system model block at the top). After the fault injection process, the original state “Read Memory State” is replaced by a faulty “Read Memory State” derived from the effects of the fault individual “RBF_CPU_0.” In the faulty state, the normal behavior mentioned above is replaced by an abnormal behavior “MCP.RegA.Value.{{+1}} = Memory ReturnFlow.OperationData.{{+1}} XOR 0 × 00000001” (shown in the system model block at the bottom).

Restrictions in fault injection

The fault analysis framework defines dependencies and restrictions for fault injection using the fault ontologies. To clearly

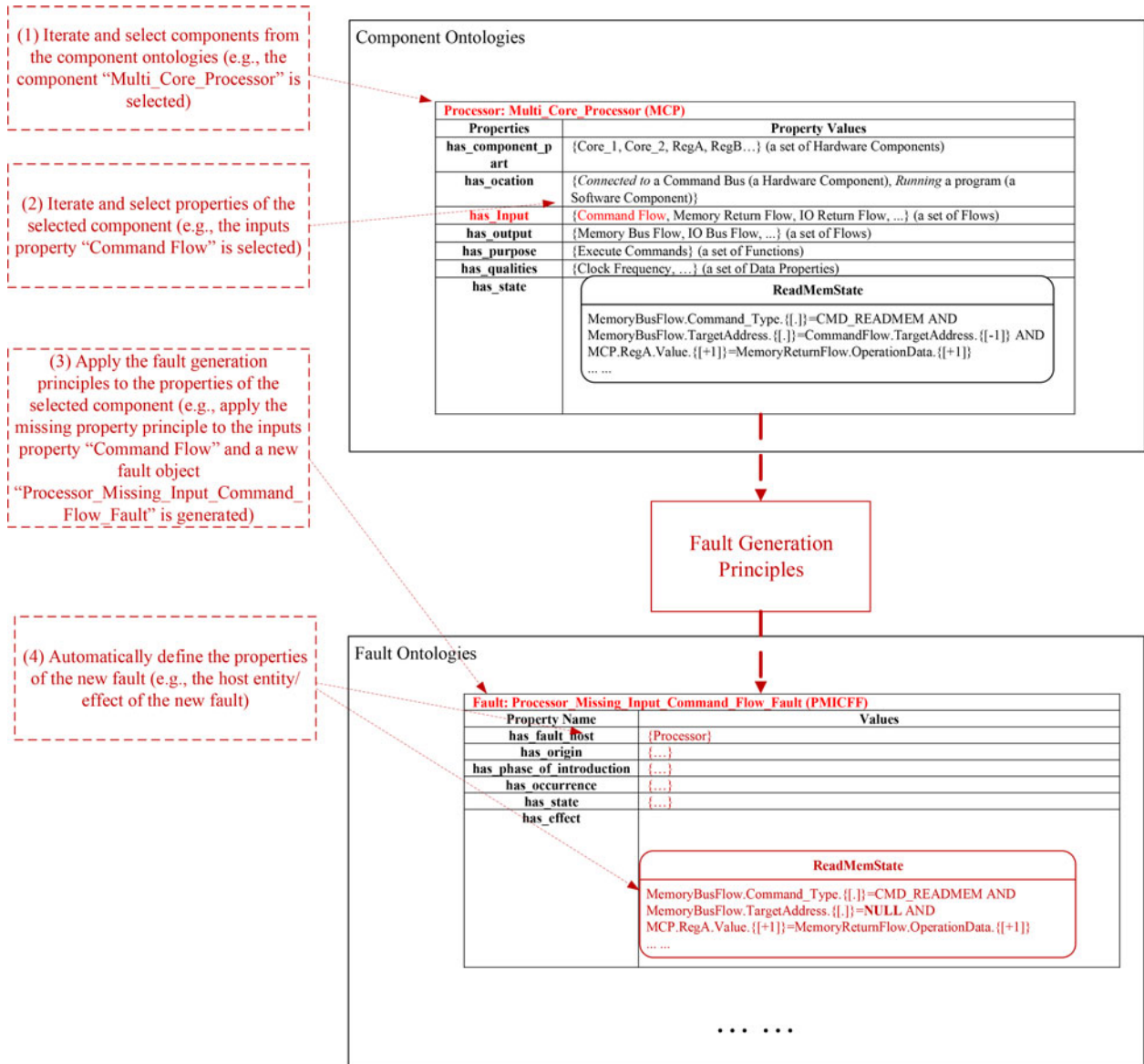


Fig. 11. The process of fault generation (using the multi-core processor component as an example).

represent the dependencies and restrictions related to the introduced ontological concepts, we extend the notations defined in Table 10 to the ones in Table 21.

In the table, we use the suffix *o* to represent the entities in the original system and use the suffix *f* to denote the entities in the system with the injected faults. Table 22 contains the general dependencies and restrictions applicable to the fault injection.

Fault propagation inference

Fault propagation inference is the process of emulating the behaviors of components, flows, and functions chronologically and deducing their states to visualize the effects of a fault.

Inference workflow

The process of fault inference is shown in Figure 13. At the beginning, the system models with the injected faults (see Section

“Fault injection”) are read and parsed by the analysis framework. Then, the states of all components, flows, and functions will be set to their default states and the time step counter is set to 0. Then, the behaviors under the default states will be executed (i.e., be used as assertions, aka evidence) to determine any changes in the states caused due to fault propagation. Details of the state inference is discussed in the section “State inference.” After that, the inference process enters a loop for each time step, starting from time step 1 ($I = 0 + 1$). For each time step of the fault inference, the BRs of components, flows, and functions will be inserted into the evidence pool, aka a set of assertions or assumptions.

State inference

To infer the states of components, flows, and functions at every time step, the BRs in the evidence pool are used as proofs. During the inference, the behavioral rules with time are used as

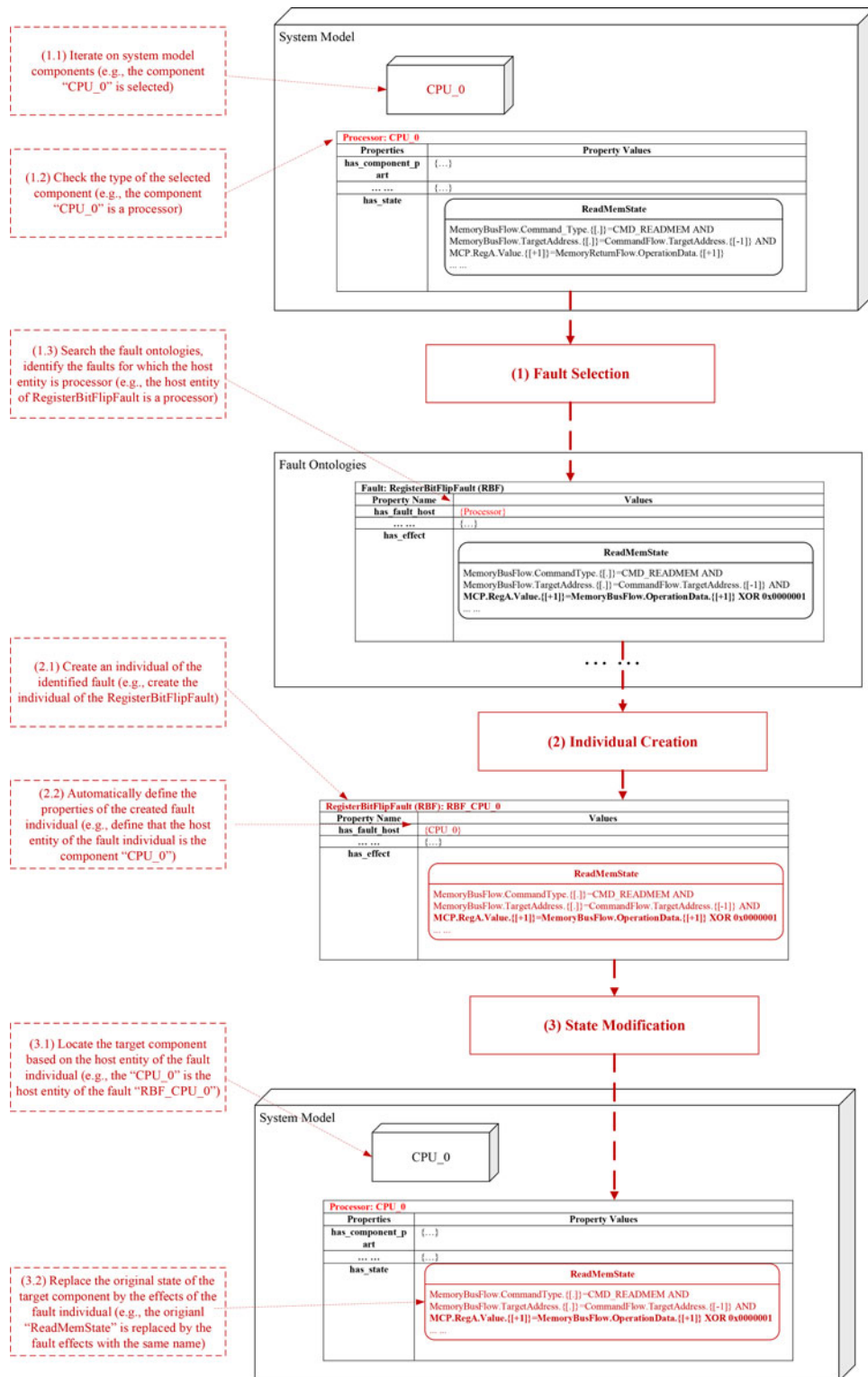


Fig. 12. Fault injection process (using the "RegisterBitFlipFault" as an example).

assertions that are inserted into an evidence pool, which is the set of all assertions that describe the "objective facts" of the SUA. For example, we assume that at the beginning of a simulation, the default state of a processor is "IdleState." According to the states of a processor defined, the behaviors "MemoryBusFlow.CommandType.t0 = NULL AND ..." will be inserted into the

evidence pool, which means that the "Command Type" of the "Memory Bus Flow" is invalid at time step 0.

Based on the assertions contained in the evidence pool, the states of components, flows, and functions can be inferred by identifying whether the triggering conditions (TCs) of states are True or False through Satisfiability Modulo Theories (SMT);

Table 21. New symbols used for representing fault injection restrictions

Symbols	Description
CPI_o	The original individual of component selected for fault injection
CPI_f	The individual of component with the injected fault
CPI	An individual of component (nominal or faulty) in the system under analysis
FLI	An individual of flow in the system under analysis
FCI	An individual of function in the system under analysis
SYS_o	The original system includes the individuals of original components, flows, and functions
SYS_f	The system includes the individuals of components, flows, and functions with the injected faults

Bjorner and De Moura, 2011). For example, if we define the states of the function “Provide Demand” as shown in Figure 14, to infer its state, we need to determine whether the TC “RDP.State.[.] == RDP.State.RDPRunningState AND ReadDemand_CommandReturnFlow.OperationData == DEMAND_VALUE” is True or the TC “RDP.State.[.] == RDP.State.RDPRunningState AND ReadDemand_CommandReturnFlow.OperationData != DEMAND_VALUE” is True.

According to SMT, the status of a statement can be (1) *Valid*, which means that the statement is definitely True; (2) *Invalid*, which means that the statement is definitely False; and (3) *Satisfiable*, which means that the statement can be True, depending on further assertions.

When a TC is identified as a true TC, this confirms that the corresponding state should be activated. On the other hand, when a TC is identified as a false TC, this verifies that the corresponding state is inactive. However, if a TC is identified as a

Table 22. Dependencies and restrictions for fault injection

No.	Dependencies and Restrictions	Descriptions
IR01.	$\forall X, X \in Composedof(CPI_o), X \notin Composedof(CPI_f) \rightarrow \forall t, Value_t(Qualities(X)) = NULL, Value_t(Outputs(X)) = NULL;$	The qualities and outputs of an object which is no longer a composition of another object will be invalid.
IR02.	$\forall X, X \in Location(CPI_o), X \notin Location(CPI_f) \rightarrow \forall t, Value_t(Inputs(X)) = NULL, Value_t(Outputs(X)) = NULL;$	When an object no longer has a location relation to another object, the related inputs and outputs will be invalid.
IR03.	$\forall X, X \in Inputs(CPI_o), X \notin Inputs(CPI_f) \rightarrow \forall t, Value_t(X) = NULL;$	When an input is removed from an object, the values related to that input will be invalid.
IR04.	$\forall X, X \in Outputs(CPI_o), X \notin Outputs(CPI_f) \rightarrow \forall t, Value_t(X) = NULL;$	When an output is removed from an object, the values related to that output will be invalid.
IR05.	$\forall X, X \in Purpose(CPI_o), X \notin Purpose(CPI_f) \rightarrow \forall t, Value_t(Inputs(X)) = NULL, Value_t(Outputs(X)) = NULL;$	When an object no longer has a purpose relation to another object, the related inputs and outputs will be invalid.
IR06.	$\forall Q, Q \in Qualities(CPI_o), Q \notin Qualities(CPI_f) \rightarrow \forall t, Value_t(Q) = NULL;$	When a quality is removed from an object, the values related to that quality will be invalid.
IR07.	$\forall CPI_o \in SYS_o, \forall FA, Isa(CPI_o) = HostEntity(FA) \rightarrow \exists FAI, CPI_f, Isa(CPI_f) = Isa(CPI_o), Isa(FAI) = FA, CPI_f = HostEntity(FAI), States(CPI_o) \neq States(CPI_f), Effects(FAI) \subseteq States(CPI_f);$	When a fault class is selected and instantiated, the effect of the fault individual will be included by the faulty individual of the target component.
IR08.	$\forall FL, Source(FL) = \emptyset \rightarrow \forall t, Value_t(Inputs(FL)) = NULL, Value_t(Outputs(FL)) = NULL;$	When a source of a flow is removed, the inputs and outputs of the flow will be invalid.

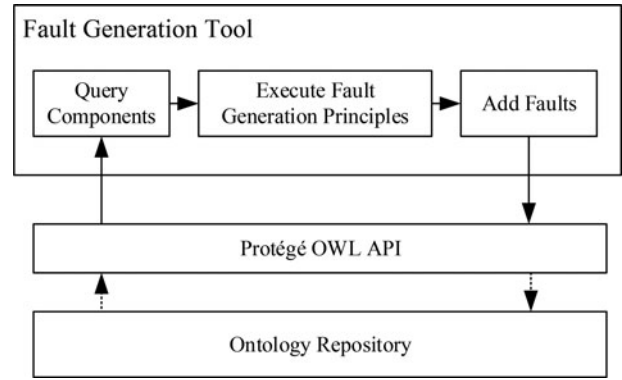


Fig. 13. Workflow of fault propagation inference.

satisfiable statement, then further inferences are required. For example, if another TC is verified as Valid, then its state will be activated. However, if all the other TCs among the current object are identified as Invalid, then the satisfiable TC will be used as a true TC and its state will be activated. The possible situations and corresponding results are listed in Table 23. When the system branches, one satisfiable state will be activated in every branch and the inference continues.

Using the system shown in Figure 8 as an example, Table 24 shows the states of selected components, flows, and functions at each time step and the assertions inserted into the evidence pool based on their BRs. Assume that at the beginning of the fault propagation inference (time step 0), the states of all the components are idle. We can infer that the state of the function “Provide Demand” (PDF) is unknown, which is the default state. Then, the software program RDP is activated and its state changes to “RDPRunningState.” In this case, the behaviors under the state “RDPRunningState” of the component “RDP” are executed (i.e., inserted into the evidence pool). As a result,

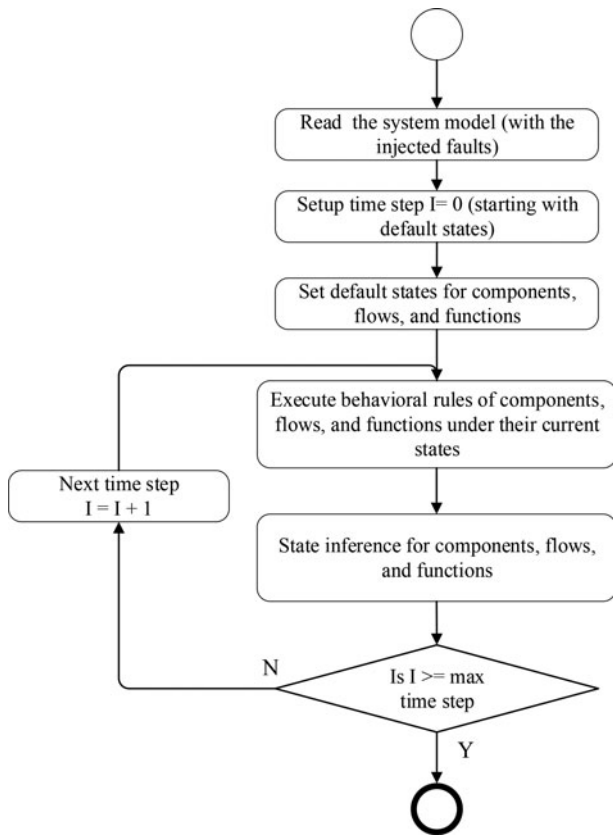


Fig. 14. States of function “provide demand.”

the “Command Type” of “ReadDemand_CommandFlow” from “RDP” changes from an invalid value (NULL) to CMD_READIO, as shown in the table. According to Table 24, the state of “CPU_0” is changed to “CPU0ReadIOState.” Then, at time step 2, the behaviors under the state “ReadIOState” of “CPU_0” are executed. These behaviors further activate the state “DUDReadState” of “DUD.” At time step 3, the behaviors of the state “DUDReadState” are executed. Based on the assertions inserted into the evidence pool, we can infer that “RDP.State.t4 == RDP.State.RDPRunningState AND ReadDemand_CommandReturnFlow.OperationData.t4 == DEMAND_VALUE” is True. As a result, the state of the function “PDF” changes to “PDFOperatingState” at time step 4.

Flow merging and branching

In a system model, multiple individuals of components or functions will possibly connect to the same individual of flow, for example, two data receivers are attached to one data bus. In this case, the final value of the flow’s qualities will be impacted by all the connected individuals. How to calculate the final value of the qualities (e.g., a flowrate) depends on the type of such flow and the type of the quality. For example, if the flow is an electricity flow, when calculating the quality “current,” the final value should be the sum of all the output “current” of all the connected individuals. Hence, the rule “Sum” will be applied to the variable “current” of the electricity flow. Table 25 summarizes some general rules of flow merging. The selection of the rules for a specific flow’s quality is usually based on physics or other related standards. In Tables 25 and 26, the notation FL represents an

Table 23. Possible situations and results

Situations	Results
One is valid, the others are invalid/satisfiable	Activate the valid state.
More than one are valid	Mistake in TCs, the inference process halts.
All are invalid	Mistake in TCs, the inference process halts.
One is satisfiable, the others are invalid	Activate the satisfiable state.
More than one are satisfiable, the others are invalid	System branches.

individual of flow, the notation CF represents an individual of component or function.

Correspondingly, multiple individuals of components or functions may accept objects from one individual of flow. In this case, the actual input of the connected component or function would be a portion of the flow, such as an electricity flow. Table 26 summarizes general rules for flow branching. The selection of the rules for a specific flow’s quality is usually based on physics or other related standards.

Case study

In this section, the correctness and effectiveness of the proposed method are verified by using a water tank control system, a simplified cyber physical system with a computer-based controller and corresponding mechanical devices. In the case study, faults that possibly occur during system design, development, and operation are generated and their impacts on the functionality of the system are analyzed. The experiments attempt to cover all the types of faults that currently exist in the fault ontology. As a result, the proposed framework simulates the propagation and impacts of all the generated faults and generates a table containing the states of the components and functions in the system under analysis. The experiment is designed to verify the correctness of the inference results. Most of the generated faults are injected into an actual implementation of the system, and the experiment will compare the data sampled from the actual system to the inference results. The ratio of errors and the accuracy of time sequences will be used as metrics to compare the results. It should be noted that the fault propagation inference is at the design level, that is, based on design level knowledge, and contrasted with an implementation which in contrast is fully fledged with all low-level implementation details defined.

System introduction

The system under analysis is a computer-controlled feedwater system which is a simplified version of the one that can be found in a nuclear power plant. The structure of the computer system is displayed in Figure 15. The components and flows in the system are grouped into three layers, including a hardware layer, an operating system layer, and a user application layer.

To implement the functionality of the case study system, several mechanical system components and their corresponding functions and flows are created. Figure 16 illustrates the mechanical components involved in the SUA.

Table 24. Example of function state inference

Time steps	Objects	States	Assertions in the evidence pool
0	RDP	RDPIdleState	ReadDemand_CommandFlow.CommandType.t0=NULL AND ReadDemand_CommandFlow.TargetAddress.t0=NULL AND ...
	CPU_0	CPU0IdleState	ReadDemand_IOBusFlow.CommandType.t0=NULL AND ReadDemand_MemoryBusFlow.CommandType.t0=NULL AND ...
	DUD	DUDIdleState	DUD.StoredData.t0=DEMAND_VALUE AND DUD.Address=DUD_DEMANDADDRESS...
	PDF	PDFUnknownState	N/A
1	RDP	RDPRunningState	ReadDemand_CommandFlow.CommandType.t1=CMD_READIO AND ReadDemand_CommandFlow.TargetAddress.t1=DUD_DEMANDADDRESS AND ...
	CPU_0	CPU0IdleState	ReadDemand_IOBus_Flow.CommandType.t1=NULL AND ReadDemand_MemoryBus_Flow.CommandType.t1=NULL AND ...
	DUD	DUDIdleState	DUD.StoredData.t1=DUD.StoredData.t0 AND ...
	PDF	PDFUnknownState	N/A
2	RDP	RDPRunningState	...
	CPU_0	CPU0ReadIOState	ReadDemand_IOBusFlow.CommandType.t2=ReadDemand_CommandFlow.CommandType.t1 AND ReadDemand_IOBusFlow.TargetAddress.t2=ReadDemand_CommandFlow.TargetAddress.t1 AND CPU_0.RegA.Value.t3=ReadDemand_IOReturnFlow.OperationData.t3 AND ...
	DUD	DUDIdleState	DUD.StoredData.t2=DUD.StoredData.t1 AND ...
	PDF	PDFUnknownState	N/A
3	RDP	RDPRunningState	...
	CPU_0	CPU0ReadIOState	...
	DUD	DUDReadState	ReadDemand_IOReturnFlow.OperationData.t3=DUD.StoredData.t3 AND DUD.StoredData.t3=DUD.StoredData.t2 AND ...
	PDF	RDFUnknowState	N/A
4	RDP	RDPRunningState	...
	CPU_0	CPU0ReadIOState	ReadDemand_CommandReturnFlow.OperationData.t4=CPU_0.RegA.Value.t4 AND CPU_0.RegA.Value.t4=CPU_0.RegA.Value.t3 AND ...
	DUD	DUDReadState	...
	PDF	PDFOperatingState	N/A

The primary functions of the SUA are summarized in Table 27. They encompass storing water and supplying water. The detailed conditions for identifying the states of each function are also shown.

The two major functions are implemented by several software programs. Figure 17 illustrates the relations between these

programs. In detail, the program “ReadDemand_Program” first reads the set point of the water level and flowrate from an existing data file. Then, the program “InletCtrl_Program” and “OutletCtrl_Program” will sample the measures provided by the pressure and flow sensors deployed in the physical system and send the samples to the corresponding memory units. The routine

Table 25. General rules for flow merging

Name	Rules	Description	Examples
SUM	$Value_t(Quality(FI)) = \sum_{i=1}^n Value_t(Output(CPI_i FCI_i));$	The final value of a quality is the sum of all the output values of the connected components or functions.	Current of an electricity flow
AVG	$Value_t(Quality(FI)) = \frac{1}{n} \sum_{i=1}^n Value_t(Output(CPI_i FCI_i));$	The final value of a quality is the average of all the outputs of the connected components or functions.	Voltage of an electricity flow combined from two electricity flow with the same current.
CAT	$Value_t(Quality(FI)) = Concatenate \left(\begin{matrix} Value_t(Output(CPI_1 FCI_1)), \\ Value_t(Output(CPI_2 FCI_2)), \\ \dots \end{matrix} \right);$	The final value of a quality is the concatenation of all the outputs of the connected components or functions.	A buffer receiving data from multiple providers.
SLT	$Value_t(Quality(FI)) = Select \left(\begin{matrix} Value_t(Output(CPI_1 FCI_1)), \\ Value_t(Output(CPI_2 FCI_2)), \\ \dots \end{matrix} \right);$	The final value of a quality is the value of the component or functions that is activated (the value is not NULL).	A Control Area Network (CAN) bus with multiple microcontrollers attached.

Table 26. General rules for flow branching

Name	Rules	Description	Example Flows
EQU	$Value_i(Input(CPI_i FCI_i)) = Value_i(Quality(FLI)), 1 < i < N;$	All connected components or functions will receive the same value from the flow.	Network broadcasting
PMA	$Value_i(Input(CPI_i FCI_i)) = a_i \times Value_i(Quality(FLI)), \sum_{i=1}^N a_i = 1, 1 < i < N;$	Every connected component or function will receive a parameter-controlled value from the flow.	Power of an energy flow, Software defined networks

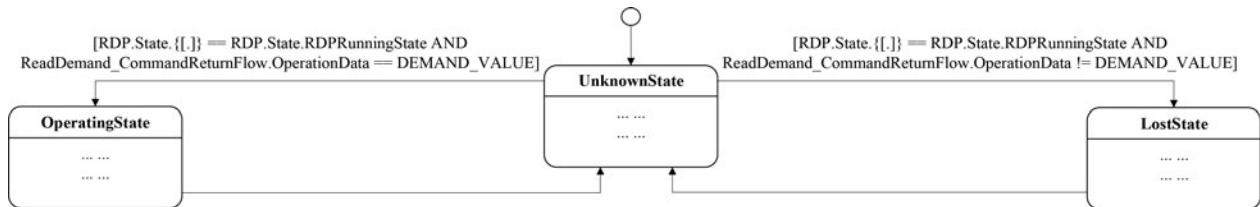


Fig. 15. Architecture of the case study system.

```

list_unexecuted_components = all components in the system
For component_i in list_unexecuted_components
    Execute_Rules(component_i)

Func Execute_Rules (component_i)
    State_x = get_current_state (component_i)
    Rules = get_behavioral_rules(component_i, State_x)
    Result = assert(Rules)
    Remove component_i from list_unexecuted_components
    If Not Result == Satisfiable Then
        Throw Execution_Error # the current situation is unfeasible
        Check_Related_Components(component_i)

Func Check_Related_Components(component_i)
    For output_m in get_outputs(component_i)
        For component_c in get_sinks(input_m)
            Check_State (component_c)

Func Check_State (component_c)
    For State_x in get_states(component_c)
        Rules = get_triggering_condition(State_x)
        Result = test_assertions(Rules)
        If Result == Valid Then:
            set_current_state(component_c, State_x)
    
```

Fig. 16. Mechanical subsystem involved in the case study system.

Table 27. Functions associated to the case study system

Function	States	Conditions
Storing Water	Operating	$(Demand * 95\%) < Water_Level < (Demand * 105\%)$
	Degraded	$(Demand * 50\%) < Water_Level \leq (Demand * 95\%)$
	Lost	$Water_Level \leq (Demand * 50\%)$
Supplying Water	Operating	$(Demand * 95\%) < Output_Flowrate < (Demand * 105\%)$
	Degraded	$(Demand * 50\%) < Output_Flowrate \leq (Demand * 95\%)$
	Lost	$Output_Flowrate \leq (Demand * 50\%)$

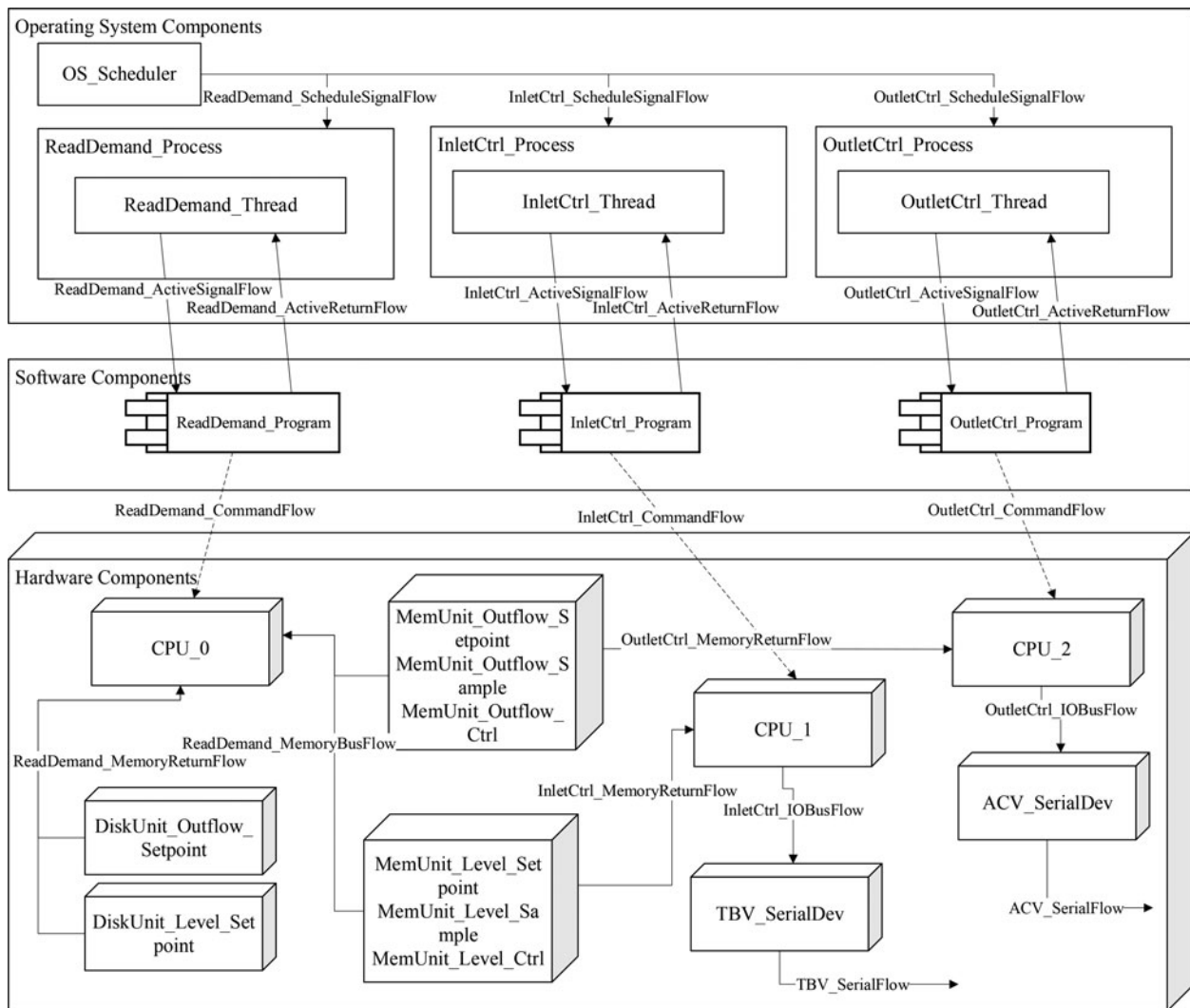


Fig. 17. Activity diagram of the case system.

“Calculate_Level_Control” implements the control algorithm and calculates the control signal for the level control valve (aka TBV). Concurrently, the routine “Calculate_Outflow_Control” is in charge of calculating the control signal for the outflow control valve (aka ACV). Finally, the outcomes of “Calculate_Level_Control” and “Calculate_Outflow_Control” are used by the routines “Send_Level_Control” and “Send_Outflow_Control,” which send the actual control signals to the corresponding mechanical components through the serial ports. The system will periodically execute the aforementioned control process to maintain the water level and the output flowrate close to their set points.

Model construction and fault injection

Based on the proposed ontologies, the system model with 90 individuals (i.e., instances of the ontological concepts) is built for the case study system. Table 28 provides the detailed numbers of individuals in the case study system.

Various types of faults were injected into the system model, including faults collected from existing research (as shown in Table 29) and the faults generated by the proposed ontological

methodology (as shown in Table 30). The faults in Table 30 are grouped by the fault generation principles applied to the system components. In summary, 1467 faults were generated.

Table 31 calculates the overlap between the individuals of the existing faults and the ones of the generated faults. Since one fault class may have multiple individuals in the case study system, the number of fault individuals is usually greater than the number of fault classes. The table shows that a large number of generated faults are not covered by existing faults described in the literature.

Analysis results and comparisons

As one example of the results of analysis, Tables 32 and 33 describe the results obtained for the test scenario associated with the fault “Incorrect_Outputs” applied to the disk unit “MemUnit_Outflow_Setpoint” (i.e., the output of the disk unit storing the set point of the flowrate is NOT_A_NUMBER). Components are grouped by domain: “Application,” “OS,” “PC Hardware,” and “Mechanical System.” In this case, an illegal set point value was read from the control file. However, since there is a defect in the “ReadDemand_Program” application such that the validity of the data is not fully verified, the illegal value was

Table 28. Number of individuals in the system model

Ontologies	Domains/Types		Number
Component	HW	Mechanical	5
		Computer Hardware	11
	SW	Operating System	3
		User Application	5
Flow	Material	12	
	Energy	2	
	Data	32	
Function			20
Total			90

consequently sent to the software “InletCtrl_Program” and caused the control algorithm to halt and send out an invalid control signal (NULL). The NULL signal fully closed the valve “ACV” (the default state of the valve) and finally caused a system failure. The failures (the lost state) of components’ and systems’ functions are highlighted in the table.

We used the actual, that is, physical/real world implementation, of the water control system to verify our framework. As an example, we manually modified the set-point file and added

Table 31. Statistics related to individuals of the existing faults and the generated faults

	PC hardware	OS	Application	Total
Number of existing faults covered by fault generation principles	14	3	16	33
Number of existing faults that cannot be covered by fault generation principles	31	20	12	63
Number of generated faults not covered by existing faults	706	259	249	1214

illegal data to the disk to mimic the faults in reading the disk during system operation. In the experiment, it is observed that the inflow setpoint is “corrupted” in the control processor at “Calculate_Level_Control” to a zero value at 300 s, shown in Figure 18. Due to the illegal value of the set point of the output flow, the “ACV” was fully closed at 300 s. This is a permanent fault. Then, the closed “ACV” led to a dramatic increase of the water level and hence led to the failure of the system function “Store_Water.” This result is consistent with the prediction of our framework.

Table 34 provides statistics that allow comparison between fault inference and real system behavior under fault. Because of

Table 29. Fault classes in the fault ontology (existing faults documented in the literature)

Host Components	Fault Classes
Processor	Address Error, Data Error, Control Error, Bit flips in registers (Duba and Iyer, 1988; Mehdizadeh et al., 2008), stuck-at logic error (Carter et al., 2005)
Memory	Stuck-at Fault, Address Decode Fault, Stuck-open Fault (van de Goor and Al-Ars, 2000)
Bus	Delay, Crosstalk, Transient (Metra et al., 2000), Instruction Fault, Data bit is Stuck at 1 or 0 (Naraway and Venkatesan, 1986)
Disk	Stuck-at Error, Read Timeout (Talagala and Patterson, 1999)
OS	Deadlock, Floating Point Error, Stack Overflow, Memory Leakage (Chou et al., 2001)
User Program	Invalid Program Flow, Incorrect Opcode Address, Unused Memory, Invalid Read Address, Invalid Opcode, Invalid Write Address, Non-Existent Memory (Mahmood and McCluskey, 1988)

Note: The behaviors of components in bold faults can be covered by the fault generation principle introduced in this paper.

Table 30. Statistics related to fault generation for the case study system (aka new faults)

Fault Categories	Physical	PC Hardware	Operating System	Application	Total
Missing Composed of	5	10	5	8	28
Additional Composed of	5	95	5	16	121
Missing Inputs	7	53	5	7	72
Additional Inputs	57	163	60	80	360
Missing Outputs	8	20	8	21	57
Additional Outputs	79	267	166	64	576
Incorrect Qualities	14	0	0	10	24
Incorrect Locations	40	67	6	47	160
Missing States	5	45	7	12	69
Total	220	720	262	265	1467

Table 32. Component states for the example scenario

Time Steps	Application				OS					PC Hardware								Mechanical	
	RP	SO	CC	SD	SC	OP	OT	PP	TP	C0	C1	C2	DL	ML	MS	SL	SA	AC	TK
0	I	I	I	I	I	I	I	I	I	I	I	I	I	I	I	I	I	N	N
1	R	I	I	I	R	R	R	I	I	RI	I	I	R	I	I	I	I	N	N
2	R	I	I	I	R	R	R	I	I	WM	I	I	I	W	I	I	I	N	N
...
5	I	R	I	I	R	I	I	R	R	I	RI	RI	I	I	I	R	I	N	N
6	I	R	I	I	R	I	I	R	R	I	WM	WM	I	I	W	I	I	N	N
7	I	I	R	I	R	I	I	R	R	I	RM	RM	I	I	R	I	I	N	N
8	I	I	R	I	R	I	I	R	R	I	MP	MP	I	I	I	I	I	N	N
9	I	I	R	I	R	I	I	R	R	I	WM	WM	I	W	I	I	I	N	N
10	I	I	I	R	R	I	I	R	R	I	RM	RM	I	R	I	I	I	N	N
11	I	I	I	R	R	I	I	R	R	I	WO	WO	I	I	I	I	W	C	N
12	R	I	I	I	R	R	R	I	I	RI	I	I	R	I	I	I	I	N	N
...
37	R	I	I	I	R	R	R	I	I	RI	I	I	R	I	I	I	I	N	F
38	R	I	I	I	R	R	R	I	I	WM	I	I	I	W	I	I	I	N	F

Note: I: Idle State, N: Nominal State, R: Reading/Running State, W: Writing State, RI: ReadIOState, WM: WriteMemState, WO: WritelOState, RM: ReadMemState, MP: MultiplyMemState, C: Changing State, F: Full State.

Components: C0: CPU_0, C1: CPU_1, C2: CPU_2, DL: DiskUnit_Level_Setpoint, ML: MemUnit_Level_Setpoint, MS: MemUnit_Level_Sample, SL: Outflow_SerialDev, SA: ACV_SerialDev, SC: OS_Scheduler, OP: ReadDemand_Process, OT: ReadDemand_Thread, AC: ACV, TK: Tank, RP: ReadDemand_Program, SO: Sample_Outflow, CC: Calculate_Outflow_Control, SD: Send_Outflow_Control.

Table 33. Functional states for the example scenario

Time Steps	Application				OS					PC Hardware					Mechanical		SYS	
	RS	SL	PC	WC	FS	RR	ER	RC	EC	RL	CL	CS	GL	GA	CO	WA	SW	PW
0	O	O	O	O	O	O	O	O	O	O	O	O	O	O	O	O	O	O
1	L	U	U	U	O	O	O	U	U	L	U	U	U	U	O	O	O	O
2	L	U	U	U	O	O	O	U	U	U	L	U	U	U	O	O	O	O
...
5	U	O	U	U	O	U	U	O	O	U	U	U	O	U	O	O	O	O
6	U	O	U	U	O	U	U	O	O	U	U	O	U	U	O	O	O	O
7	U	U	O	U	O	U	U	O	O	U	U	O	U	U	O	O	O	O
8	U	U	L	U	O	U	U	O	O	U	U	U	U	U	O	O	O	O
9	U	U	L	U	O	U	U	O	O	U	L	U	U	U	O	O	O	O
10	U	U	U	L	O	U	U	O	O	U	L	U	U	U	O	O	O	O
11	U	U	U	L	O	U	U	O	O	U	U	U	U	L	L	O	O	O
12	L	U	U	U	O	R	R	U	U	L	U	U	U	U	O	O	O	O
...
20	U	U	L	U	O	U	U	O	O	U	U	U	U	U	O	O	D	O
21	U	U	L	U	O	U	U	O	O	U	L	U	U	U	O	O	D	O
...
37	L	U	U	U	O	O	O	U	U	L	U	U	U	U	O	O	L	L
38	L	U	U	U	O	O	O	U	U	U	L	U	U	U	O	O	L	L

Note: O: Operating State, D: Degraded State, L: Lost State, U: Unknown State.

Functions: RS: read set point, SL: sample level, PC: PID control, WC: write control value, FS: schedule processes, RR: run read set point software, ER: execute read set point software, RC: run control software, EC: execute control software, RL: record level set point, CL: cache level set point, CS: cache level sample, GL: sample level, GA: control ACV, CO: control outlet water flow, WA: store water in tank, PW: provide water, SW: store water.

technical limitations, only a portion of fault types can be applied to the real hardware and software. For example, an extreme high voltage signal may damage the physical equipment (e.g., the pumps and valves). As a result, 450 test scenarios can be faithfully implemented in the real system. The results derived from the proposed framework successfully predict all of 450 real system test scenarios.

After inspection of the results from the fault inference and the real system, we found that all results from the real system agree with the predictions of the fault inference. Since the inference is a qualitative simulation with inference but the results from the real system yield a large data set, inspecting the results consists of the following activities: (1) check the intermediate and final states of functions and components (e.g., failed or not) and (2)

check the time order of the important events that occurred during the system operation (e.g., functional failures, state transitions).

Discussion

As shown by the above analysis, faults that can occur in computer systems were simulated and their effects on functional failures were analyzed. The analysis emulates the behavior of every component involved in the fault propagation. The results of this analysis visualize the fault propagation paths and explicitly show the causality between faults and functional failures. These causal relationships are useful for researching fault prediction and can assist in the design of fault tolerance and fault recovery mechanisms.

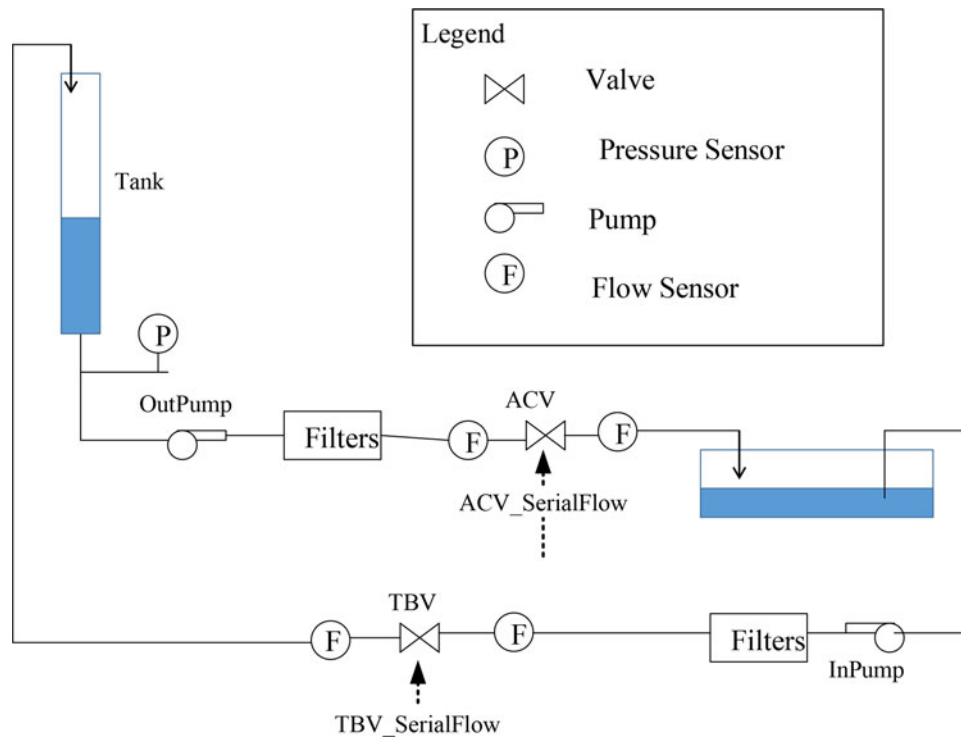


Fig. 18. Signals sampled from the real system.

Table 34. Comparison of results between fault inference and real system

Result from	Fault Host Entity	O	D	L	U	Total
Inference	Mechanical	52	17	158	8	230
	Computer Hardware	271	2	446	11	730
	Operating System	92	0	146	4	242
	User Application	78	0	168	6	274
	Total	493	19	926	29	1467
Real System	Mechanical	47	7	122	-	176
	Computer Hardware	18	0	60	-	78
	Operating System	2	0	10	-	12
	User Application	54	0	130	-	184
	Total	121	7	322	-	450

Note: O: Number of Operating Scenarios, D: Number of Degraded Scenarios, L: Number of Lost Scenarios, U: Number of Uncertain Scenarios, Total: Total Number of Scenarios.

A case study using the proposed method used a model with 24 components and 46 flows to verify 20 functions and subfunctions of the system at the early design stage. The framework generated 1467 faults based on the ontological concepts. All of the faults were analyzed and 98% of faults' impacts were clearly predicted (missing were the scenarios with "uncertain" outcomes). The result proves that the proposed method can effectively generate faults and their propagation paths at the design level, which is useful for improving the robustness of the system.

Since the specification of the system was not well defined at the early design stage, uncertainties existed in the system design. The uncertainty could be an unclear type of component, a free flow quality without constraints, or an undefined subfunction parameter. These aspects will probably lead to uncertainties in

the final results. For example, without any specification of a component in the feedwater system, the function (supplying water) of the system cannot be inferred because the fault inference engine cannot confirm if the output flowrate of water is within the design range. However, this uncertainty can be reduced when we specify the maximum flowrate of the pipes and valves composing the system. Along with the development process, the concreteness of the system will finally remove the uncertainties in the results once it is built and deployed.

The fault propagation inference takes reasonable time to produce the outcomes, about 5 min to analyze one fault scenario. Theoretically, 1467 scenarios require 122 h, about 5 days. However, we can analyze the scenarios simultaneously since they are independent. By running the inference on a

High-Performance Computer (HPC) with 50 cores, calculating the results only requires 2.4 h. The performance of the calculation is significantly improved.

Conclusion

This research provides a novel method (IS-FAON) for analyzing fault propagation and its effects. Starting from the ontologies of components, functions, flows, and faults, this paper constructed a scientific foundation for describing and tracking faults in a computer system across multiple domains throughout design and development. In order to construct the system and fault models, a series of fundamental concepts were introduced in the form of ontologies and their dependencies. An investigation was then performed into the faults, including their type, cause, life cycle aspects, and effect. Principles and rules were created to generate various faults based on system configurations. After the modeling process, a fault inference engine was proposed to execute actions and simulate the process of fault generation and propagation. As a result, fault paths that impact components and functions were obtained.

Gathering fault propagation paths at an early design phase significantly help to predict and improve the reliability and safety of a system. First, the paths provide intuitive evidence for fault detection and diagnosis. Second, fault prevention mechanisms and redundancy policies can be applied to the most frequently traversed nodes in order to efficiently implement fault masking and isolation. Also, the fault propagation paths are helpful for generating test cases for system verification since they provide useful information on triggering faults that are possibly hiding in the system under analysis.

Future work will be focused on how to improve the proposed method. First, the ontologies of components, flows, and functions for computer systems will be enriched. Domain-specific hardware and software components for various engineering domains (e.g., aerospace, nuclear, medical, etc.) and more specific sources of faults (e.g., electromagnetic, vibration) will be considered and added to the repositories. Also, tools for automating model construction will be studied and developed. Due to the sophistication of models specially built for complex systems, these tools should be capable of automatically reading components and flows existing in the target system. In addition, further optimization (e.g., concurrent computation) will be applied to the inference process to accelerate the fault propagation analysis.

Acknowledgments. We would like to thank Yunfei Zhao for reviewing this paper. We would also like to thank Paul Johnson and Alex Scarmuzzi, the undergraduate students who assisted in the investigation of components and behaviors of the computer architecture and operating system.

Financial support. This research was supported by the Air Force Office of Scientific Research (AFOSR) and the Advanced Research Projects Agency-Energy (ARPA-E) from the Department of Energy (DOE).

Conflict of interest. Dr. Fuqun Huang is on the Board of Institute of Interdisciplinary Scientists. The other authors declare none.

References

- Allen JD and Unicode Consortium (2007) OWL 2 Web Ontology Language, p. 1417.
- Avizienis A, Laprie J-C and Randell B (2001) Fundamental concepts of computer system dependability. *IARP/IEEE_RAS Workshop on Robot*
- Dependability: Technological Challenge of Dependable Robots in Human Environments*, pp.1–16.
- Avizienis A, Laprie J-C and Randell B (2004a) Dependability and its threats: a taxonomy. *Building the Information Society: Proc. IFIP 18th World Computer Congress*, 22–27 August 2004, Toulouse, France (July 1834), pp. 91–120.
- Avizienis A, Laprie JC, Randell B and Landwehr C (2004b) Basic concepts and taxonomy of dependable and secure computing. *IEEE Transactions on Dependable and Secure Computing* 1, 11–33.
- Benazzouz Y, Aktouf OEK and Parisis I (2014) A fault fuzzy-ontology for large scale fault-tolerant wireless sensor networks. *Procedia Computer Science* 35, 203–212.
- Bjorner N and De Moura L (2011) Satisfiability modulo theories: introduction and applications. *Communications of the ACM* 54, 69–77.
- Brüning S, Weißleder S and Malek M (2007) A fault taxonomy for service-oriented architecture. *Proceedings of IEEE International Symposium on High Assurance Systems Engineering*, pp. 367–368.
- Carter JR, Ozev S and Sorin DJ (2005) Circuit-level modeling for concurrent testing of operational defects due to gate oxide breakdown. *Proceedings of the Design, Automation and Test in Europe 2005*, Vol. I, pp. 300–305.
- Chen R, Zhou Z, Liu Q, Pham DT, Zhao Y, Yan J and Wei Q (2015) Knowledge modeling of fault diagnosis for rotating machinery based on ontology. *Proceedings of the 2015 IEEE International Conference on Industrial Informatics, INDIN 2015*. IEEE, pp. 1050–1055.
- Chou A, Yang J, Chelf B, Hallem S and Engler D (2001) An empirical study of operating systems errors. In *Proceedings of the eighteenth ACM symposium on Operating systems principles*, pp. 73–88.
- Diao X, Zhao Y, Pietrykowski M, Wang Z, Bragg-Sitton S and Smidts C (2018) Fault propagation and effects analysis for designing an online monitoring system for the secondary loop of the nuclear power plant portion of a hybrid energy system. *Nuclear Technology* 202, 106–123.
- Dibowski H, Holub O and Rojiček J (2017) Ontology-based fault propagation in building automation systems. *International Journal of Simulation: Systems, Science and Technology* 18, 1.1–1.14.
- Duba P and Iyer RK (1988) Transient fault behavior in a microprocessor - a case study, pp. 272–276.
- Durães JA and Madeira HS (2006) Emulation of software faults: a field data study and a practical approach. *IEEE Transactions on Software Engineering* 32, 849–867.
- Etzioni Z, Keeney J, Brennan R and Lewis D (2010) Supporting composite smart home services with semantic fault management. *Proceedings of the 5th International Conference on Future Information Technology, FutureTech 2010*. IEEE. doi:10.1109/FUTURETECH.2010.5482708.
- Foster JC, Osipov V, Bhalla N, Heinen N and Aitel D (2005) *Buffer Overflow Attacks - Detect, Exploit, Prevent*. doi:10.1016/B978-1-932266-67-2.X5031-2.
- Gao J, Li G and Gao Z (2008) Fault propagation analysis for complex system based on small-world network model. *Proceedings of the 2008 Annual Reliability and Maintainability Symposium*, pp. 1–5.
- Gruber T, Acquisition K, Ontology F ... Level TK (2012) *What is an ontology?* pp. 1–2.
- Hecht M and Baum D (2019) Failure propagation modeling in FMEAs for reliability, safety, and cybersecurity using SysML. *Procedia Computer Science*, 153, 370–377.
- Hirtz J, Stone R, McAdams D, Szykman S and Wood K (2002) A functional basis for engineering design: reconciling and evolving previous efforts. *Research in Engineering Design* 13, 65–82.
- Horrocks I, Patel-schneider PF, Boley H, Tabet S, Grosz B and Dean M (2004) SWRL: A Semantic Web Rule Language Combining OWL and RuleML. W3C Member Submission 21, pp. 1–20.
- Hummer W (2012) Deriving a unified fault taxonomy for event-based systems categories and subject descriptors. *Proceedings of the 6th ACM International Conference on Distributed Event-Based Systems*, pp. 167–178.
- Isaksson AJ, Harjunkoski I and Sand G (2018) The impact of digitalization on the future of control and operations. *Computers and Chemical Engineering* 114, 122–129.
- Jiang Y, Yin S and Kaynak O (2018) Data-driven monitoring and safety control of industrial cyber-physical systems: basics and beyond. *IEEE Access* 6, 47374–47384.

- Lackovic M, Talia D, Tolosana-Calaszan R, Bañares JA and Rana OF** (2010) A taxonomy for the analysis of scientific workflow faults. *Proceedings of the 13th IEEE International Conference on Computational Science and Engineering, CSE 2010*, pp. 398–403.
- Lauer C, German R and Pollmer J** (2011) Fault tree synthesis from UML models for reliability analysis at early design stages. *ACM SIGSOFT Software Engineering Notes* **36**, 1.
- Liu B, Ding Z, Wu J and Yao L** (2019) Ontology-based fault diagnosis: a decade in review. *ACM International Conference Proceeding Series*, pp. 112–116.
- Mahmood A and McCluskey EJ** (1988) Concurrent error detection using watchdog processors—a survey. *IEEE Transactions on Computers* **37**, 160–174.
- Mehdizadeh N, Shokrolah-Shirazi M and Miremadi SG** (2008) Analyzing fault effects in the 32-bit OpenRISC 1200 microprocessor. *Proceedings of the 3rd International Conference on Availability, Security, and Reliability, ARES 2008*, pp. 648–652.
- Metra C, Favalli M and Riccò B** (2000) Self-checking detection and diagnosis of transient, delay, and crosstalk faults affecting bus lines. *IEEE Transactions on Computers* **49**, 560–574.
- Musen MA** (2015) The Protégé project: a look back and a look forward. *AI Matters* **1**, 4–12.
- Mutha C, Jensen D, Tumer I and Smidts C** (2013) An integrated multidomain functional failure and propagation analysis approach for safe system design. *Artificial Intelligence for Engineering Design, Analysis and Manufacturing* **27**, 317–347.
- Narraway JJ and Venkatesan R** (1986) Bus and instruction fault diagnosis. *IEE Proceedings: Computers and Digital Techniques* **133**, 333–340.
- Natarajan S and Srinivasan R** (2010) A distributed intelligence system for improving fault diagnostic performance in large scale chemical processes. *Escape-20*, December.
- O'Callahan R and Choi JD** (2003) Hybrid dynamic data race detection. *ACM SIGPLAN Notices* **38**, 166–177.
- Papakonstantinou N and Sierla S** (2012) Early phase fault propagation analysis of safety critical factory automation systems. *IEEE International Conference on Industrial Informatics (INDIN)*, pp. 364–369.
- Park J, Kim HJ, Shin JH and Baik J** (2012) An embedded software reliability model with consideration of hardware related software failures. *Proceedings of the 2012 IEEE 6th International Conference on Software Security and Reliability, SERE 2012*, pp. 207–214.
- Shu S, Wang Y and Wang Y** (2016) A research of architecture-based reliability with fault propagation for software-intensive systems. *Proceedings of the 2016 Annual Reliability and Maintainability Symposium*, April. doi:10.1109/RAMS.2016.7447984.
- Talagala N and Patterson D** (1999) An analysis of error behavior in a large storage system, pp. 1–17.
- van de Goor AJ and Al-Ars Z** (2000) Functional memory faults: a formal notation and a taxonomy. *Proceedings of the IEEE VLSI Test Symposium*, pp. 281–289.
- Wallace M** (2005) Modular architectural representation and analysis of fault propagation and transformation. *Electronic Notes in Theoretical Computer Science* **141**, 53–71.
- Weichhart G, Molina A, Chen D, Whitman LE and Vernadat F** (2016) Challenges and current developments for sensing, smart and sustainable enterprise systems. *Computers in Industry* **79**, 34–46.
- Yang F, Xiao D and Shah SL** (2013) Signed directed graph-based hierarchical modelling and fault propagation analysis for large-scale systems. *IET Control Theory and Applications* **7**, 537–550.
- Yang Z, Zhang C, Liu MZ, Yu Y, Qiao L and Peng X** (2015) Fault propagation characteristics analysis for large-scale electronic system by hierarchical signed directed graph. *Proceedings of the AUTOTESTCON*, pp. 219–226.
- Zhao L, Thulasiraman K, Ge X and Niu R** (2016) Failure propagation modeling and analysis via system interfaces. *Mathematical Problems in Engineering* **2016**. doi:10.1155/2016/8593612.
- Zhou Y, Li Q and Zuo Y** (2009) Fault knowledge management in aircraft maintenance. In *2009 8th International Conference on Reliability, Maintainability and Safety*, pp. 645–649.

Xiaoxu Diao is a post-doctoral researcher working in the Reliability and Risk Laboratory in the Department of Mechanical and Aerospace Engineering, The Ohio State University. He received the MTech and PhD degrees in Software Reliability Engineering from the School of Reliability and System Engineering at Beihang University, Beijing, China, in 2006 and 2015, respectively. He has participated in several research and projects regarding software reliability and testing. His research interests are real-time embedded systems, software testing methods, and safety-critical systems.

Michael Pietrykowski is a PhD candidate in the Nuclear Engineering Program in the Department of Mechanical and Aerospace Engineering at The Ohio State University. He received his BS in Electrical and Computer Engineering from OSU specializing in computers and solid state devices and is an NRC Graduate Fellowship and DOE NEUP Fellowship recipient. His research is focused on investigating digital instrumentation and controls systems in nuclear power plants using hardware-in-the-loop experimental testing, and network effects on distributed control systems.

Fuqun Huang is currently a Principal Scientist and the director of the “Software Engineering & Psychology” Interdisciplinary Research Program at the Institute of Interdisciplinary Scientists. She received her PhD on Systems Engineering from Beihang University in 2013 and was a visiting scholar at Centre for Software Reliability, City University of London in 2011. Dr. Huang was a Postdoctoral Researcher (2014–2016) with The Ohio State University. Dr. Huang research interests include human errors in software engineering, reliability and safety of software systems, and software quality assurance. She is the member of IEEE Standards, Program Committee for the IEEE International Workshop on Software Certification.

Chetan Mutha has a PhD in Mechanical Engineering from The Ohio State University. His research interests include systems and software reliability assessment, integrated system design and analysis, and fault diagnosis early in the design phase, automotive systems, and artificial intelligence. He has several published journal and conference papers. He collaborates with Dr. Smidts and her Risk and Reliability Laboratory located at The Ohio State University. Currently, he is working as a technical consultant in the Patent Law and is employed by Pillsbury Law firm.

Carol Smidts is a Professor in the Department of Mechanical and Aerospace Engineering at The Ohio State University. She graduated with a BS/MS and PhD from the Université Libre de Bruxelles, Belgium, in 1986 and 1991, respectively. She was a Professor at the University of Maryland at College Park in the Reliability Engineering Program from 1994 to 2008. Her research interests are in software reliability, SW safety, SW testing, PRA, and human reliability. She is a senior member of the Institute of Electrical and Electronic Engineers and a member of the editorial board of Software Testing, Verification, and Reliability