Automatic Synthesis of Static Fault Trees from System Models

Jianwen Xiang, Kazuo Yanoo, Yoshiharu Maeno, and Kumiko Tadano
Service Platforms Research Labs.
NEC Corporation
Kawasaki, 211-8666 Japan
Email: {j-xiang@ah, k-yanoo@ab, y-maeno@aj, k-tadano@bq}.jp.nec.com

***Abstract*—Fault tree analysis (FTA) is a traditional reliability analysis technique. In practice, the manual development of fault trees could be costly and error-prone, especially in the case of fault tolerant systems due to the inherent complexities such as various dependencies and interactions among components. Some dynamic fault tree gates, such as Functional Dependency (FDEP) and Priority AND (PAND), are proposed to model the functional and sequential dependencies, respectively. Unfortunately, the potential semantic troubles and limitations of these gates have not been well studied before. In this paper, we describe a framework to automatically generate static fault trees from system models specified with SysML. A reliability configuration model (RCM) and a static fault tree model (SFTM) are proposed to embed system configuration information needed for reliability analysis and error mechanism for fault tree generation, respectively. In the SFTM, the static representations of functional and sequential dependencies with standard Boolean AND and OR gates are proposed, which can avoid the problems of the dynamic FDEP and PAND gates and can reduce the cost of analysis based on a combinatorial model. A fault-tolerant parallel processor (FTTP) example is used to demonstrate our approach.**

***Index Terms*—Fault tree analysis, reliability analysis, system model, functional dependency, sequential dependency**

I. Introduction

To achieve a dependable computer-based system, system reliability analysis techniques must be applied during both system design and maintenance stages. At the architecture design stage, the reliability requirements must be estimated to achieve a wise tradeoff between cost and reliability. At the maintenance stage, especially for fault-tolerant systems, it is important to understand the current system state, such as the current single point of failures (SPOFs) under the failures and/or recoveries of some components.

A number of reliability analysis tools have been developed to solve different reliability models, but the construction of the reliability models mostly relies on a manual procedure. The manual development could be difficult and error-prone, especially in the case of fault-tolerant systems due to their inherent complexities such as redundancy, dependency and various interactions between components. One solution to this problem is to automatically generate reliability models from system architecture models, which is one of the main motivations behind our work.

Fault tree analysis (FTA) [1] is a traditional reliability analysis technique. It is basically a deductive procedure for determining the various combinations of basic component failures that could result in the occurrence of a specific undesired top event at the system level. Standard static Boolean logic constructs, such as AND, OR, and Voting (k-out-of-n, k/n) gates, are used to decompose the fault events and construct the static fault trees. In addition, to model how sequences of fault events cause system failures, several dynamic constructs, such as Priority AND (PAND) [1], [2] and Functional Dependency (FDEP) gates [2] are also proposed. One benefit of the dynamic fault trees (DFTs) is that they can represent the dynamic behaviors of system failure mechanisms in a more compact way compared with traditional Markov models. The DFTs are not only used in research-oriented projects, but also recently introduced in some commercial FTA tools such as Relex Fault Tree [3].

Unfortunately, the formal semantics of DFTs has not been well defined, and there are some fundamental semantic paradoxes in the informal descriptions of some dynamic gates such as the FDEP gates. These semantic troubles have not been addressed before and may result in incorrect system unreliabilities. Moreover, one constraint of the DFTs is that the history of event occurrences must be maintained during system maintenance stage, since the specifications of dynamic gates such as PAND depend on the past order of event occurrences. Consequently, the maintenance of the DFTs could be a critical issue since it now relies on a permutational rather than a combinatorial model, and the history of component failures (and recoveries) must be used to update the fault tree states. How to avoid the semantic troubles and correctly represent the dynamic relations with standard static logic gates in a combinatorial model is another important motivation behind our work.

The idea of using system models and their transformation into reliability models is not new. An early transformation from UML system models to DFTs could be found in [4], and a recent generation of DFTs from Architectural Analysis and Design Language (AADL) models is proposed in [5]. Our approach differs from the DFT transformations by using a static fault tree model to represent dynamic relations such as functional and sequential dependencies, which is one of our main technical contributions. A process to automatically derive traditional static fault trees from AADL models could be found in [6], in which the dynamic relations such as sequential

dependencies among components are not considered.

A synthesis method of fault trees from MATLAB-Simulink system models is presented in [7], in which dynamic fault tree gates are still used, and the method is exclusive to the transformation of MATLAB-Simulink models (http://www.mathworks.com/). A tool to manually associate a user-built, product-line software fault tree analysis with the related product-line AADL models is developed in [8]. The work presented in this article differs from that of [8] in that we advocate for the automatic derivation of fault trees directly from the SysML [9] system models to bridge the divide between reliability and system engineering.

There are also some works on the transformation from systems models into reliability models other than fault trees. Examples of them are RASCad [10], a tool using reliability block diagrams and Markov chains for computer systems with reliability, availability, serviceability (RAS) characteristics (specific to Sun products), reliability analysis of composite services using Markov reward models [11], and derivation of timed petri nets (TPN) from UML structural diagrams (e.g., use case, class, object and deployment diagrams) [12] and SysML state machine diagrams [13]. In addition, a model-based methodology and a tool for service and business process availability assessment are proposed in [14].

In our approach, we first develop a reliability configuration model (RCM) to embed the system configuration information needed for reliability analysis. The RCM is defined as a general logical system independent from specific system modeling languages, such that it can be applied to different system models if similar logic structures can be extracted from their modeling languages. A static fault tree model (SFTM) is then developed to generate static fault trees from the RCM specifications. The separation of SFTM and RCM is to separate the success and failure criteria of the systems, such that the system designers and reliability analysts can work independently in a more effective way. Both RCM and SFTM are specified with Maude, an executable algebraic formal specification language [15], [16]. The Maude specifications of RCM and SFTM serve as the FTA engine for fault tree analysis thanks to the executability of Maude. Finally, to verify our approach, we develop a software tool to transform SysML [9] system models into RCM specifications. This approach is one of the essential functions of NEC's case-based system assessment environment (CASSI) [17].

The rest of this article is organized as follows. First, we present some background knowledge of FTA in Section II. The RCM embedding system configuration information needed for FTA is presented in Section III, in which issues such as classification of components, modeling of functional dependencies between components and fault-tolerant components (e.g., spares) are discussed. The SFTM is proposed afterwards in Section IV to generate static fault trees based on the RCM specifications. Issues and problems of traditional dynamic gates, such as FDEP and PAND gates, as well as their static representations are addressed in this section. The transformation from SysML system models to our RCM specifications is discussed in Section V. A fault-tolerant parallel processor (FTTP) example is used to demonstrate our approach in Section VI. Finally, the concluding remarks are summarized in Section VII.

II. Background of FTA

FTA was first developed in the 1961 to facilitate analysis of the Minuteman missile system [18] and has been widely used in the aerospace, electronics, and nuclear industries for safety and reliability analysis for years. It is a top-down approach; input consists of knowledge of the system's functions as well as its failure modes and their effects. The approach is graphical, constructing fault trees with standardized symbols representing events, static and dynamic logic gates, as shown in Figure 1. There are several variations and extensions, but in this article we limit ourselves to these symbols.

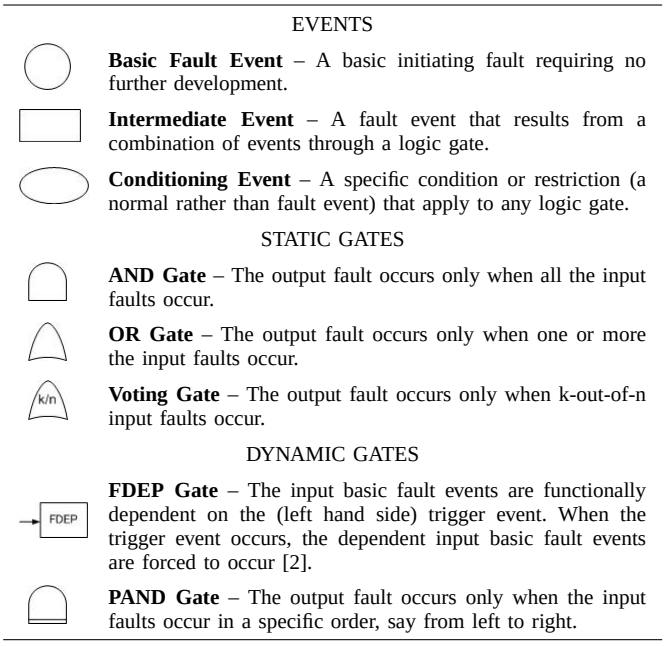


Fig. 1: Fault tree symbols

One of the main purposes of FTA is to find all the smallest combinations of basic events (e.g., basic fault and conditioning events) which will result in the top event, namely minimal cut sets (MCSs). In addition the qualitative analysis with MCSs, FTA is also often used in probabilistic analysis, such as calculation of system failure rate (i.e., the probability of the top event) and allocation of software reliability [19].

III. Reliability Configuration Model

A. Classification of Components

By a `component`, we shall understand a `hardware` or `software` that is a part of system and performs some specific functions. A `composite` component may consists of a set of sub components, and a `system` could be regarded as a composite component. A `cluster` is also a composite component which typically consists of a non-empty set of

sub components with the same type and provides certain redundancy. The `redundancy` of a cluster is understood as the maximal number of component failures which can be tolerated by the cluster. The maximal redundancy of a cluster is equal to the total number of sub components of the cluster minus 1. In contrast, an `atomic component` is a basic component which does not include any other component.

A special kind of hardware is the `buses` which provide accesses between hardware. A function, `connectedBy`, is introduced to define the connection between a hardware and a bus. For the further classification of hardware and software of computer systems, we refer to [20].

```
sorts   Hardware Software Component .
subsort Hardware Software < Component .
sorts   Cluster System Composite AtomicComponent .
subsort System Cluster < Composite < Component .
subsort AtomicComponent < Component .
sorts   Bus .
subsort Bus < Hardware .

op connectedBy : Hardware  -> BusSet .
op redundancy  : Cluster   -> Nat .

var C : Cluster .

eq redundancy(C) <= (| consists(C) | - 1) = true .
```

Fig. 2: Classification of Components

B. Modeling of Functional Dependencies

A key issue of reliability modeling is the functional dependencies between components. Generally speaking, the functional dependencies could be classified at two dimensions, namely *horizontal* and *vertical* dependencies between some "brother" components and between a composite component and its sub components, respectively.

A component may `requires` a set of brother components for its normal functioning, in which we assume that if either of the required components fails, the requiring component will fail. In other words, it is a kind of "requires *ALL*" relation. To model the "requires *ANY*" relation in terms of set rather than logical constructs, a concept of `component collection` is introduced to denote a particular set of components which are not necessary with the same type but can provide some common functions. The difference between a component collection and a cluster is that the (sub) components of the component collection could be different components with different failure rates. For instance, suppose that in a water machine, the input valve will close to avoid overflow if either the sensor detects a maximal water level in the tank, or the timer reaches a maximal time for inputting water. In this case, both the sensor and timer could be included in a component collection required by the input valve. Note that the clusters could also be used to denote the requires ANY relation when its redundancy is equal to the total number of its sub components minus 1, i.e., the component who requires the cluster will fail if all the sub components of the cluster fail.

Similarly, a composite component may functionally depends on a non-empty set of its sub components. Note that in our model, it is not necessary to include all the sub components as the vertically depended components of the composite component. In contrast, we prefer to specify only some of the *key* sub components, such as processors, with the vertical dependency relations, while leave the functional dependencies between the key sub components and other sub components, such as buses, memories, and devices, to the horizontal dependency relations. Such a modeling strategy can give more flexibilities to the specifiers. An alternative way to represent vertical dependencies is to collapse the layers of composite and sub components into one flattened layer and represent the dependencies at the flattened layer. This strategy actually requires only one dependency relation, i.e., the horizontal one.

In addition, the binding of software and hardware could be regarded as a sub relation of the horizontal dependency. For instance, suppose a software can run on a specific hardware, then we say the software requires the hardware. Note that with concept of component collection, the binding to set of candidate hardware and reconfiguration of the software to other hardware can also be expressed in our model.

An excerption of the Maude specifications of functional dependencies is presented in Figure 3. Note that the operator '`_|_`' is defined as a constructor of component collections, and the underscores '`_`' are used as placeholders of parameters in Maude.

```
sort    ComponentCollection .
subsort ComponentCollection < Component .
op _|_ : Component Component -> ComponentCollection
         [ctor comm assoc] .

op requires : Component -> ComponentSet .
op depOn    : Composite -> ComponentNeSet .
op runsOn   : Software  -> HardwareSet .

var S : Software .

eq runsOn(S) <: requires(S) = true .
```

Fig. 3: Spec. of Functional Dependencies

C. Modeling Primary and Spare Components

By a `primary` component we shall understand a component which can be replaced by some `spare` when it fails. The fault tolerant mechanism of a system consisting of a primary component and some spares is similar to the case of a cluster whose redundancy is set to the maximal number. However, two differences should be addressed here. One is that a spare is not used until one of its primary components fails; while in a cluster, all the sub components are assumed to be working from the very beginning in general. The other is that competition for shared spares (i.e., spares which support more than one primary components) may happen between the primary components. For instance, if a spare has already been activated by a failure of a primary component, then it cannot

```
sorts   Primary Spare .
sorts   HotSpare ColdSpare WarmSpare .
subsort Primary Spare < Component .
subsort HotSpare ColdSpare WarmSpare < Spare .

op hasSpares    : Primary -> SpareNeSet .
op supPrimaries : Spare   -> PrimaryNeSet .
```

Fig. 4: Spec. of Primary and Spare Components

replace other primary components which are also supported by the spare anymore. The competition of shared spares may bring about the issue of sequential dependency between the failures of primary components, and we will further discuss this issue in detail later in Section IV-B.

The relationships between the primary and spare components can be summarized as follows. A primary component `has` a non-empty set of spares, likewise a spare can `support` a non-empty set of primary components. Note that the latter support relation could be derived from the former. Although a spare can support several primary component, the spare can replace at most one primary component, i.e., the one first failed. This constraint will be defined later in Section IV-B since the replace event is actually a conditioning event of FTA and need not be included in the RCM. In addition, depending on how and when the spares are used, the spares could be classified as `hot`, `cold`, or `warm` spares.

An excerption of the Maude specifications about the primary and spare components and their relationships are presented in Figure 4.

IV. Static Fault Tree Model

In the static fault tree model, we address three important issues, namely definitions of different events and gates for fault tree construction, competition of shared spares, and representation of functional dependencies of different components.

A. Events and Gates

By an `event` we shall understand a Boolean predicate representing a particular component state. An event could be either an `fault event` or a `conditioning event`, depending on whether it represents an error or normal (working) state of the component. A `basic event` is an atomic event requiring no further analysis (decomposition), and it could be either a `basic fault event` or a conditioning event. In FTA, the basic fault events usually refer to the primary failures of the atomic (i.e., non-composite) components which occur in an environment for which the components are qualified [1]. The basic fault events are usually assumed to be *statistically independent*, i.e., a component may be broken at any time regardless of the states (failures) of other components. The probabilities of the basic fault events constitute the input for the calculation of the system unreliability.

For clarity, four kinds of layered component failures are considered in our model below, and an extraction of the Maude specification of the event classification is presented in Figure 5.

- A primary component is `down` if itself is functionless and none of its spares has been activated, and the down of a non-primary component is equal to its functionless event.
- A component is `functionless` if itself is failed or it has been disabled by other components.
- A component is `disabled` if some other components which are functionally required by the component are down. The disabled event is typically an *external* failure to the component, and it could be regarded as a kind of secondary failure [1] of the component if we treat the functional dependency as a kind of qualification of the component.
- A component is `failed` if some *internal* failure of the component occurs, and it could be regarded as a kind of primary failure [1] of the component. The failed event of an atomic component is typically a basic fault event.

Note that the above classification of component failures involves concepts such as activation of spares and functional dependency between components. These concepts will be further explained in Section IV-B and IV-C.

```
sorts   Event FaultEvent ConditioningEvent .
sorts   BasicEvent BasicFaultEvent .
subsort Event < Bool .
subsort FaultEvent < Event .
subsort BasicEvent < Event .
subsort BasicFaultEvent < BasicEvent .
subsort ConditioningEvent < BasicEvent .

op down         : Component -> FaultEvent .
op functionless : Component -> FaultEvent .
op disabled     : Component -> FaultEvent .
op failed       : Component -> FaultEvent .

var C : Component .
var A : AtomicComponent .

ceq down(C) = functionless(C) if not (C :: Primary) .
mb  failed(A) : BasicFaultEvent .
```

Fig. 5: Spec. of Events

With respect to the logic gates, since we present solutions to transform dynamic FDEP and PAND gates into static gates in our model, only standard logic disjunction and conjunction are defined for the construction of static fault trees. The voting gate is defined as a function from a non-empty set of events and a non-zero natural number (representing the voting number) to an event, and a recursive equation is given to expand the voting function into a disjunctive normal form (DNF). An extraction of the Maude specification of the logic gates is presented in Figure 6.

B. Competition of Shared Spares

As we mentioned earlier in Section III-C, competition for shared spares may happen between primary components. For instance, suppose a spare, `s`, supports (is shared by) two primaries, `p1` and `p2`, then one fault event of `p1` (in addition to that both `p1` and `s` are failed), called `T` for short, is that `p1` becomes failed (short in `P1`) after `p2` is failed (short in `P2`).

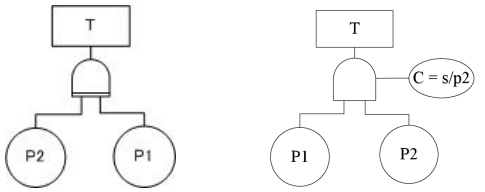
```
op _&_ : Event Event -> Event [assoc comm prec 55] .
op _+_ : Event Event -> Event [assoc comm prec 59] .
op vote : EventNeSet NzNat -> Event .

var E : Event .
var N : NzNat .
var P : EventPreSet .

ceq vote({P}, N) = false if N > | {P} | .
ceq vote({E, P}, N) = E & vote({P}, N - 1)
    if | {E, P} | = N .
eq  vote({E, P}, 1) = E + vote({P}, 1) .
eq  vote({E}, 1) = E .
ceq vote({E, P}, N) = E & vote({P}, N - 1) + vote({P}, N)
    if N < | {E, P} | /\ N > 1 .
```

Fig. 6: Spec. of Logic Gates

The standard PAND gate solution (e.g., [2]) to this example is presented in Figure 7a.



(a) PAND (b) AND with condition

Fig. 7: PAND vs. AND gates

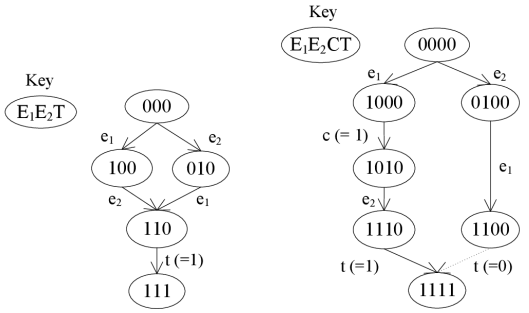
In our static fault tree model, the sequential dependency of PAND gate is modelled by a standard static AND gate with a dependent conditioning event (call CAND for short). The corresponding CAND gate of the PAND gate of Figure 7a is illustrated in Figure 7b, in which the conditioning event `C` is defined as that the spare `s` replaces (with symbol "/" for clarity) the primary `p2`. `C` is a statistically dependent event because its occurrence *only* depends on the occurrence of `P2` under the non-occurrence of `P1`. With the transformation, the occurrence of the output event T does not rely on the specific sequence of the input events anymore, and the maintenance of the fault tree becomes history-independent. Note that an event, called "substitutes for" with the same meaning as of our conditioning event `replaces` is used to model spares in the synthesis of DFTs from UML system models [4]. However, the usage of such an event for the representation of sequential dependencies has not been discovered in that approach.

A lemma is introduced below to support the transformation from PAND to CAND.

Lemma 1: Given a PAND with an output event T and two input events E_1 and E_2, assuming that the component failures are independent and unrepairable, and the system satisfies the Markov property, there must exist some dependent conditioning event, C, which *must be and can only be* activated in the acceptable path (where E_1 occurs before E_2) if it is initially false, or negated in the unacceptable path (where E_2 occurs before E_1) if it is initially true [21], [22].

The proof of the existence of the conditioning event C can be done by contradiction with the semantics of the PAND gate based on the Markov chain of the PAND gate. Fig. 8a presents the Markov chain of the PAND gate, in which Boolean values 0 and 1 are used to denote the non-occurrence and occurrence of events, respectively. Without the occurrence of any other event, both of the acceptable and unacceptable paths of the two input events can reach the same state (110 representing $E_1 \cdot E_2 \cdot \neg T$) accounting for the occurrence of the top event. Here, a contradiction arises in terms of the semantics of the PAND gate, i.e., only the acceptable path can result in the output event.

The additional event is a conditioning (normal) event rather than a fault event, because it is a statistically dependent event with respect to the two input fault events, and we assume that all the component failures are independent. More specifically, if the conditioning event is initially false, then it must only be activated (to become true) in the state where E_1 just occurred under the non-occurrence of E_2; If it is initially true, then it must only be negated in the state where E_2 just occurred under the non-occurrence of E_1. Note that it is impossible for the conditioning event, C, to occur or to be negated in the state where E_1 and $E2$ both occurred, since such a state can be reached by both acceptable and unacceptable sequences of the two inputs. The revised Markov chain with the dependent conditioning event C for the case that it is initially false and is activated in the state 1000 is presented in Fig. 8b.



(a) Without C (b) With C (initially false)

Fig. 8: Markov models of PAND and CAND

Actually, Lemma 1 breaks a traditional incomplete understanding of PAND gates, that is, while Markov models are typically used to analyze the dynamic behaviors of fault trees with PAND gates (e.g., [23]), the fundamental Markov property is somehow overlooked in such analysis, i.e., the occurrence of the output of a PAND gate should *not* depend on the specific sequence of states that led the system to the source state of the occurrence (in which both of the two input events have occurred).

Lemma 1 only deals with the PAND gate with two inputs, as for the PAND gates with more than two inputs, another lemma can be used to transform the PAND gates with more than two inputs into a set of PAND gates with only two inputs.

Lemma 2: Any PAND gate with more than two input events can be transformed into a cascade of PAND gates each with two input events, or a conjunction of a set of consecutive PAND gates each with two input events.

The equivalent cascaded and consecutive PAND trees of a PAND gate with three inputs is presented in Fig. 9. The proof of Lemma 2 can be carried out in terms of the temporal semantics of the PAND gate, such as the one presented in [24]. By combining Lemma 1 and 2, our static transformation can be applied to any PAND gate with any number of input events.

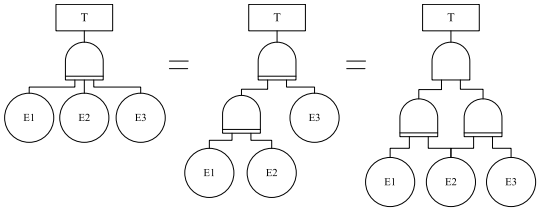


Fig. 9: Transformation to cascaded and consecutive PAND

Based on the above demonstration, the error mechanism of primary and spare components could be summarized as follows, and an extraction of the corresponding formal specification is presented in Figure 10.

```
op replaces      : Spare Primary -> ConditioningEvent .
op noneActivated : Primary SpareNeSet -> FaultEvent .
op activatedBy   : Spare PrimarySet -> Event .

vars P P' : Primary .
var  S    : Spare .

eq down(P) =
   functionless(P) & noneActivated(P, hasSpares(P)) .
eq noneActivated(P, {S}) =
   functionless(S) + activatedBy(S, supPrimaries(S) \ {P}) .
eq activatedBy(S, {P'}) =
   functionless(P') & replaces(S, P') .
eq functionless(P) & functionless(P') & replaces(S, P) +
   functionless(P) & functionless(P') & replaces(S, P') =
   functionless(P) & functionless(P') .
```

Fig. 10: Error Mechanism of Primaries and Spares

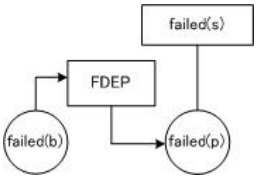
- A primary is `down` if itself is `functionless` and none of its spares has been activated.
- A spare is not activated for a primary (when the primary is functionless) could be caused by either the spare is `functionless` or the spare has already been activated by some other primary component which is also supported by the spare.
- The event that a spare is activated by a primary component equals to that the primary component is functionless and the spare `replaces` the primary.

Note that in Figure 10, the two operators, `noneActivated` and `activatedBy`, are introduced as auxiliary functions (intermediate events) for clarity, and the operator, $\setminus$, stands for standard set difference operator. Also Note that the last equation is used to absorb the "symmetric" conditioning events of `replaces` between two primary components sharing the same one spare. The equation could be understood in another way with the temporal semantics of the corresponding PAND gates, such as $A \ \& \bigcirc B + B \ \& \bigcirc A = A \ \& B$, where $\bigcirc$ stands for the standard next operator of linear temporal logic [25]. Note that here for clarity, we do not consider the simultaneity of events.

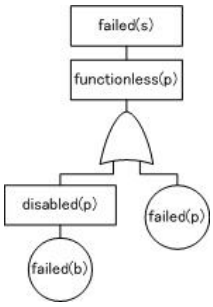
C. Representation of Functional Dependency

Before presenting our static representations of functional dependencies, we first clarify the key semantic paradoxes of traditional dynamic FDEP gate.

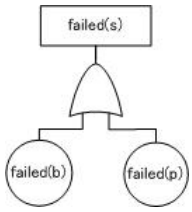
1) Paradoxes of FDEP gate: The FDEP gate was originally proposed in 1990s (e.g., [2], [26]). Currently, it is included in some commercial FTA tools, such as Relex Fault Tree [3]. The main intention of FDEP gate is to model the functional dependency between different components. An example is that if the communication is achieved through some bus unit, then the failure of the bus will cause other connected components, such as a processor, inaccessible or unusable. In this case, the failures of the bus and the connected processor are typically called as the *trigger* and *dependent basic events*, respectively. An example fault tree of the system, `s`, consisting of the bus, `b`, and processor, `p`, by using standard FDEP gate is presented in Figure 11a.



(a) FDEP



(b) OR



(c) Simplified OR

Fig. 11: FDEP vs. OR gates

Based on the above example, one of the key paradoxes of the FDEP gate is listed below.

> ***P1****: The dependent basic events are functionally dependent on the trigger event; When the trigger event occurs, the dependent basic events are forced to occur [2], [26].*

Based on the classification of internal and external component failures as discussed in Section IV-A, it becomes apparent that there is no dependency or casuality between the so-called trigger event `failed(b)` and the dependent basic event `failed(p)`. Actually, they are statistically *independent* basic fault events. The correct logical relation between the failures of

the processor and the bus could be represented in a form such as shown in Figure 11b. More specifically, the trigger event can only force the external failure of the processer, but never the internal failure of the processor, i.e, the so-called dependent basic event. The mistake of paradox P1 is made by confusing the relationship between the trigger and dependent basic events with the functional dependency between the two components without a clear understanding of different component failures and their logical relations.

Other paradoxes, such as the FDEP must be used in an imperfect coverage model to generate the correct Markov chain (in which when the trigger event occurs, all the associated dependent events must be marked as having occurred) [26], actually are based on some very strong assumptions, such as the dependent components will be shut down and safely isolated as soon as they are disabled (by the trigger events), and each dependent component can only provide one service (function) at any time [22]. Unfortunately, such important assumptions have not been clearly stated in the original literature [2], [26]. Consequently, the omission of the assumptions may result in incorrect Markov chains and calculation of system unreliabilities. For instance, the unexpected explosion of a dependent component may still have chance to fail the whole system after the disabling event, if it has not been shut down and isolated safely and immediately after disabling.

Based on the above discussion, we argue that the semantics of FDEP gate is rather confusing and problematic. To avoid the problems, we suggest the functional dependency of components should better be denoted by standard static OR gates with a clear classification of different fault events of components, such as the ones as shown in Figure 11b and 11c.

2) Decomposition Rules for Functional Dependencies: As we discussed in Section III-B, there are generally two kinds of functional dependencies, namely horizontal and vertical dependencies between different components in terms of their hierarchies. Some general rules for the decomposition of different component failures based on the functional dependencies are summarized as follows.

- A *component collection* is `functionless` if and only if (iff) all the components of the collection are functionless (recall the definition of component collection in Section III-B); otherwise a *non-collection component* is `functionless` iff itself is `failed` or `disabled`.
- A *cluster* is `failed` iff more than k of its sub components are `down`, where k is the redundancy of the cluster. This can be represented by a voting gate with voting number $k+1$.
- A *non-cluster composite component* is `failed` iff any of its vertically depended sub components is `down`.
- A *component* is `disabled` iff any of its functionally required components is `inaccessible`.
- A *component* is `inaccessible` from another component iff either the former is `down` or there is no physical connection path between the two components.

Note that the above decomposition rules are *recursive* (combined with the equations of `down` presented earlier in Figure 5 and 10) and *transitive*. The event `inaccessible` is introduced as an auxiliary function to handle the functional dependency between two components which are not necessary to be directly connected. In other words, it can be used to cover the case of two distant components which are connected by a number of buses in a simple way; otherwise we have to define all the functional dependencies from the depended component through each connection bus to the dependent component step by step.

In our model, we assume that the physical connections are established by the bus components which support bidirectional communication. A concept of minimal bus cut set (MBCS, borrowed from MCS) is introduced to decompose the no non-connection event between two components. A MBCS is defined as a smallest set of buses which if they are all `failed`, then there is no available connection path between the components.

An extraction of the Maude specification of the above decomposition rules and concepts is presented in Figure 12. Note the last (third) parameter of `anyInaccessbile` is defined to handle *cyclic* dependencies, and it is used to store the components which have already been proceeded, likewise the last parameters of `inaccessible` and `$down` are defined for the same purpose. The semantics of the auxiliary function `$down` is the same to the fault event `down` except for that the former has an additional parameter to handle cyclic dependencies. Also note that the function `allMBCSs` is used to calculate all the MBCSs between two components, which is a power set of buses.

V. Transformation from SysML System Models

We have implemented our approach into an in-house model-based design tool called CASSI (Computer Aided System model based SI environment). In this section, we first summarize the reasons for choosing SysML as our system modeling language and some of their constructs, and then discuss the translation (extraction) from the SysML models to our RCM specifications.

A. Modeling language

We employ SysML as a system modeling language because of the following reasons.

1) Being an extension of UML, SysML is more familiar to ordinary IT engineers than other modeling languages such as AADL.
2) Some extensions of SysML to UML, such as *structured compartment*, are essential for specifying system architecture intuitively.

System models are basically represented by two kind of diagrams, namely internal block diagram (IBD) and sequence diagram (SD). The former describes the structure of the system, and the latter describes the behavior of the system.

Three important constructs of IBD are *Part Property*, *Connector*, and *Dependency*. A part is notated by a rectangle,

```
op  downUnion : NeComponentSet -> EventNeSet .
op  anyDown : ComponentSet -> FaultEvent .
op  anyInaccessible : ComponentSet Component ComponentSet
    -> FaultEvent .
op  inaccessible : Component Component ComponentSet
    -> FaultEvent .
op  allMBCSs : Component Component -> BusPowerSet .

vars C C' : Component .
var  CL   : Cluster .
var  CP   : Composite .
var  CS   : ComponentSet .
var  PCS  : ComponentPreSet .

eq  functionless((C | C')) =
    functionless(C) & functionless(C') .
ceq functionless(C) = failed(C) + disabled(C)
    if not (C :: ComponentCollection) .
eq  failed(CL) =
    vote(downUnion(depOn(CL)), redundancy(CL) + 1) .
eq  downUnion({C, PCS}) = {down(C)} \/ downUnion({PCS}) .
ceq failed(CP) = anyDown(depOn(CP))
    if not (CP :: Cluster) .
eq  anyDown({C, PCS}) = down(C) + anyDown({PCS}) .
eq  disabled(C) = anyInaccessible(requires(C), C, {C}) .
eq  anyInaccessible({C', PCS}, C, CS) =
    inaccessible(C', C, CS) + anyInaccessible({PCS}, C, CS) .
eq  inaccessible(C', C, CS) =
    if (C' in CS)
      then true
      else $anyFailed(allMBCSs(C, C')) + $down(C', CS)
    fi .
```

Fig. 12: Error Spec. of Functional Dependencies

and represents a component of the system. The Parts can be nested. A part property is usually typed by a SysML Block that has composite aggregation. A connector is notated by a solid line, and represents connection between parts. A dependency is notated by a dashed arrow, and represents dependencies between parts.

The SD describes the flow of control between actors and systems (blocks) or between parts of a system. This diagram represents the sending and receiving of *messages* between the interacting entities called *lifelines*, where time is represented along the vertical axis.

We have defined several stereotypes that specializes SysML in order to make it suitable for IT system modeling. The following are ones that are essential for reliability analysis.

1) *Component*: a specialization of Part (or Block). A component represents a hardware or a software of the system, and may have a *redundancy* attribute when the multiplicity of the component is larger than 1 (i.e., the component has several identical instances). The redundancy attribute can have one of the following values.
   a) *k/n*: Unless $k+1$ instances fail, the group of instances will not fail.
   b) *none*: No redundancy among the instances, a special case of k/n for $k=0$.
   c) *parallel*: Each instance fully backups other instances. Unless all instances fail, the group of instances will not fail. It is also a special case of k/n for $k=n-1$.
2) *Spare*: a specialization of *Dependency*, and represents the relation between primary and spare components.

We also define some extra stereotypes so that the user can describe dedicated redundancy architectures such as RAID easily. Other constructs of SysML, such as Usecase Diagrams, Requirement Diagrams, and Allocations, are also defined, but we omit the details of them here due to space limitation.

B. Translation from SysML to RCM

For reliability analysis, we have implemented a tool that extracts logical information needed for reliability analysis from the SysML system models, and converts it into the RCM specifications.

Because the translation is straightforward, we describe only a summary of the translation process.

1) *Generation of Components*: For each part with multiplicity n in the model, generate n `component`s. If the part has a redundancy attribute other than *none*, then generate a `cluster` and the `depOn` relation between the cluster and the components. If the parts are nested, generate corresponding `depOn` relations between them.
2) *Generation of Spare Relations*: For each spare relation between part A and B, generate `hasSpare` relations between the components generated by A and B.
3) *Generation of Buses*: For each connector between part A and part B in the model, generate fully-connected `bus`es between the components generated by part A and B.
4) *Generation of Horizontal Functional Dependency*: For each message in the model, generate a `requires` relation from the sender to the receiver.

Note that we extract horizontal functional dependencies from SDs, whereas others are extracted from IBDs. Because sequence diagrams represent system's behavior and are coupled with use cases by usecase diagrams, this enables us to analyze dependability of the system based on use cases.

For example, suppose a Web Server system has a backup storage, and the storage is used for a certain use case, such as daily backup of HTML contents. When analyzing another primary use case, such as Web browsing, reliabilities of the backup storage and the network connections between the Web server and the backup storage are ignored, since there is no functional dependency generated for these components in this case. This feature is beneficial because the users can analyze the risk of system failure more precisely and flexibly.

VI. Example

A relatively complex Fault-tolerant Parallel Processor (FTTP) [27], [28] configuration is used to demonstrate our static fault tree model. The same configuration has been used to demonstrate the usages of DFTs with FDEP and PAND gates in [2] (with label $\#2$).

The configuration consists of 16 processing elements (PEs), with 4 connected to each of 4 network elements (NEs). The NEs are fully connected. The 16 PEs are logically connected to form 4 triads (`T1, T2, T3`, and `T4`), each with one spare. The four spares are distributed across the NEs, and the spare on each NE can support and replace for any failed PE connected to the same NE. For instance, `TS1` can replace `T11, T22`, or

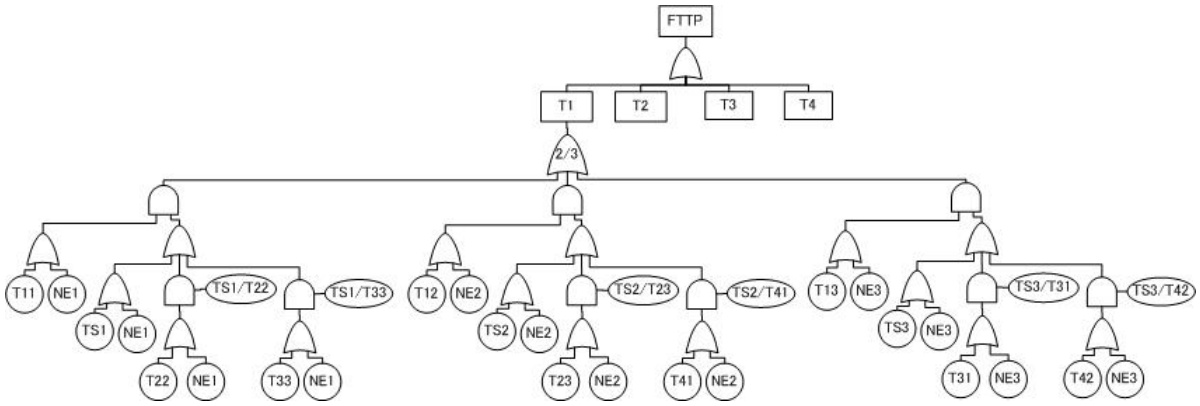


Fig. 13: Simplified Fault Tree of FTTP

`T33`. The redundancy of each triad is set to 1, i.e., if more than 1 PE are down, then the triad is failed. All the PEs functionally require the NE to which they are connected, i.e., the PEs will be disabled if the NE is failed. For simplification, no further dependency is assumed between the NEs although they are fully connected to each other. The system is down if any of the triads is down.

Based on the above descriptions, the internal block diagrams are described as Figure14 and Figure15. Figure14 represents connections amongs processors, and Figure15 represents spare relations among them. Although we can write connectors and spares in one diagram, we separate it to make it more readable. The sequence diagram of the system is rather trivial one; the client send a message, which represents arbitrary task, to the FTTP (or a computer that comprises the FTTP).

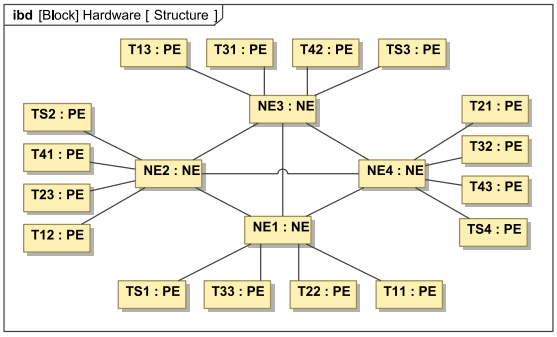


Fig. 14: Example FTTP Configuration (One spare per NE)

Based on the input SysML system models, the fault tree of the FTTP configuration can be automatically generated. Limited by the space of this paper, we only present a simplified sub fault tree of the first triad, `T1`, in Figure 13 by removing the intermediate events. Note that for simplification, we use the component names to denote the basic fault events (i.e., the internal failures) of the components, and use symbol / to simplify the conditioning event `replaces`, i.e., `S/P1` should be understood as spare `S` replaces primary component `P1`.

The static fault tree of Figure 13 is structurally different from the original DFT as shown in [2]. However, their logical

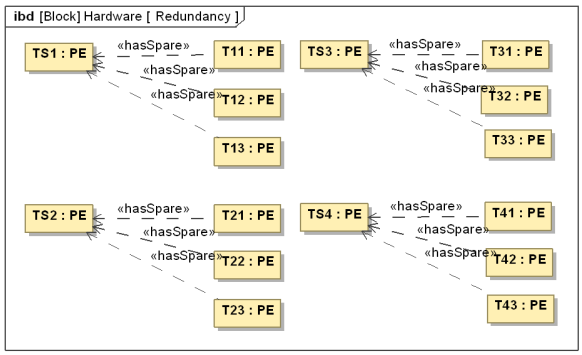


Fig. 15: Additional spare relations for Figure14

equivalence could be demonstrated in terms of the MCSs. For comparison, the MCSs of the first triad, i.e., all the combinations of basic events for the down of `T1` (and the FTTP system), are presented in Figure 16. Note that the MCSs of the DFT of [2] have not been given, but they could be manually derived from the fault tree figure. Some important observations about the MCSs are as follows.

- The orders of the MCSs are from 2 to 4, in which we define the order of a MCS as the number of basic fault events (conditioning events are not included) in it.
- Any two internal failures of the three NEs, `NE1, NE2`, and `NE3`, are enough to cause the first triad and system to down.
- The sequential dependencies caused by the competition of shared spares are captured by the conditioning event `replaces`. For instance, the MCS, `T11 & T22 & NE2 & TS1/T22`, explains that if `T11` fails after `T22` because of `TS1/T22`, and `NE2` is failed (such that `T12` is disabled), then the system is down (due to that the triad $T1$ is down).

VII. Concluding Remarks

In this article, we have presented how to generate static fault trees from fault tolerant computer-based system models. A reliability configuration model is proposed to capture

```
NE1 & NE2 + NE1 & NE3 + NE2 & NE3 +

T11 & TS1 & NE2 + T11 & TS1 & NE3 + T12 & TS2 & NE1 +
T12 & TS2 & NE3 + T13 & TS3 & NE1 + T13 & TS3 & NE2 +
T11 & T22 & NE2 & TS1/T22 + T11 & T22 & NE3 & TS1/T22 +
T11 & T33 & NE2 & TS1/T33 + T11 & T33 & NE3 & TS1/T33 +
T12 & T23 & NE1 & TS2/T23 + T12 & T23 & NE3 & TS2/T23 +
T12 & T41 & NE1 & TS2/T41 + T12 & T41 & NE3 & TS2/T41 +
T13 & T31 & NE1 & TS3/T31 + T13 & T31 & NE2 & TS3/T31 +
T13 & T42 & NE1 & TS3/T42 + T13 & T42 & NE2 & TS3/T42 +

T11 & T12 & TS1 & TS2 + T11 & T13 & TS1 & TS3 +
T12 & T13 & TS2 & TS3 +
T11 & T12 & T22 & TS2 & TS1/T22 +
T11 & T12 & T23 & TS1 & TS2/T23 +
T11 & T12 & T33 & TS2 & TS1/T33 +
T11 & T12 & T41 & TS1 & TS2/T41 +
T11 & T13 & T22 & TS3 & TS1/T22 +
T11 & T13 & T31 & TS1 & TS3/T31 +
T11 & T13 & T33 & TS3 & TS1/T33 +
T11 & T13 & T42 & TS1 & TS3/T42 +
T12 & T13 & T23 & TS3 & TS2/T23 +
T12 & T13 & T31 & TS2 & TS3/T31 +
T12 & T13 & T41 & TS3 & TS2/T41 +
T12 & T13 & T42 & TS2 & TS3/T42 +
T11 & T12 & T22 & T23 & TS1/T22 & TS2/T23 +
T11 & T12 & T22 & T41 & TS1/T22 & TS2/T41 +
T11 & T12 & T23 & T33 & TS1/T33 & TS2/T23 +
T11 & T12 & T33 & T41 & TS1/T33 & TS2/T41 +
T11 & T13 & T22 & T31 & TS1/T22 & TS3/T31 +
T11 & T13 & T22 & T42 & TS1/T22 & TS3/T42 +
T11 & T13 & T31 & T33 & TS1/T33 & TS3/T31 +
T11 & T13 & T33 & T42 & TS1/T33 & TS3/T42 +
T12 & T13 & T23 & T31 & TS2/T23 & TS3/T31 +
T12 & T13 & T23 & T42 & TS2/T23 & TS3/T42 +
T12 & T13 & T31 & T41 & TS2/T41 & TS3/T31 +
T12 & T13 & T41 & T42 & TS2/T41 & TS3/T42
```

Fig. 16: MCSs of The First Triad

the necessary system configuration information for reliability analysis, such as system structure, vertical and horizontal functional dependencies (requirements) between components, and logical relations between primary and spare components. A static fault tree model is proposed to generate fault trees based on the reliability configuration information, in which the potential troubles and problems of some traditional dynamic gates, such as FDEP and PAND gates, as well as their static representations within SFTA are addressed. In addition, the transformation from SysML system models to our reliability configuration models is discussed.

REFERENCES

[1] W. E. Vesely, F. F. Goldberg, N. H. Roberts, and D. F. Haasl, "Fault tree handbook," U.S. Nuclear Regulatory Commission, Washington, D.C, Tech. Rep. NUREG-0492, Jan 1981.

[2] J. B. Dugan, S. Bavuso, and M. Boyd, "Dynamic fault tree models for fault tolerant computer systems," *IEEE Transactions on Reliability*, vol. 41, no. 3, pp. 363–377, 1992.

[3] Relex Fault Tree, http://www.relex.com/products/faulttree.asp.

[4] G. J. Pai and J. B. Dugan, "Automatic synthesis of dynamic fault trees from UML system models," in *Prof. of The 13th International Symposium on Software Reliability Engineering (ISSRE'02)*. Los Alamitos, CA, USA: IEEE, 2002, pp. 243–256.

[5] J. Dehlinger and J. B. Dugan, "Analyzing dynamic fault trees derived from model-based system architectures," *Nuclear Engineering and Technology*, vol. 40, no. 5, pp. 365–374, Aug 2008.

[6] A. Joshi, S. Vestal, and P. Binns, "Automatic generation of static fault trees from AADL models," in *DSN 2007 Workshop on Architecting Dependable Systems*, Edinburgh, Scotland - UK, 2007.

[7] F. Tajarrod and G. Latif-Shabghi, "A novel methodology for synthesis of fault trees from MATLAB-Simulink model," *World Academy of Science, Engineering and Technology*, vol. 41, pp. 630–636, 2008.

[8] H. Sun, M. Hauptman, and R. Lutz, "Integrating product-line fault tree analaysis not AADL models," in *Proc. of the 10th IEEE International Symposium on High Assurance System Engineering*, Dallas, TX, 2007, pp. 15–22.

[9] OMG, "OMG systems modeling language, version 1.1," http://www.omg.org/spec/SysML/1.1, Nov 2008.

[10] D. Tang, J. Zhu, and R. Andrada, "Automatic generation of availbility models in rascad," in *Proc. of International Conference on Dependable Systems and Networks (DSN'02)*, 2002, pp. 488–492.

[11] N. Sato and K. Trivedi, "Accurate and efficient stochastic reliability analysis of composite services using their compact markov reward model representations," in *Proc. of IEEE International Conference on Services Computing (SCC07)*, 2007, pp. 114–121.

[12] A. Bondavalli, I. Majzik, and I. Mura, "Automated dependability analysis of uml designs," in *Proc. of 2nd IEEE Intl. Symp. on Object-oriented Real-time Distributed Computing*, 1999.

[13] E. Carneiro, P. Maciel, G. Callou, E. Tavares, and B. Nogueira, "Mapping sysml state machine diagram to time petri net for analysis and verification of embedded real-time systems with energy constraints," in *ENICS '08: Proceedings of the 2008 International Conference on Advances in Electronics and Micro-electronics*. Washington, DC, USA: IEEE Computer Society, 2008, pp. 1–6.

[14] N. Milanovic and B. Milic, "Automatic generation of service availability models," *IEEE Transactions on Services Computing*, vol. 99, Mar 2010.

[15] M. Clavel, S. Eker, P. Lincoln, and J. Meseguer, "Principles of maude," in *Proceedings of The First International Workshop on Rewriting Logic and its Applications*, J. Meseguer, Ed., vol. 4 of Electronic Notes in Theoretical Computer Science. Asilomar, California: Elsevier Science, Sep 1996.

[16] M. Clavel, F. Durán, S. Eker, P. Lincoln, N. Martí-Oliet, J. Meseguer, and C. Talcott, *Maude Manual (Version 2.4)*, 2008.

[17] K. Yanoo, S. Izukura, J. Xiang, D. Kimura, H. Sakaki, M. Tani, and S. Tajima, "Evaluation of non-functional requirements based on model-based system integration environment," in *Proc. of the 5th International Conference on Project Management (ProMAC)*. The Society of Project Management, 2010.

[18] H. A. Watson and B. T. Laboratories, "Launch control safety study," Bell Telephone Laboratories, Murray Hill, NJ, Tech. Rep., 1961.

[19] J. Xiang, K. Futatsugi, and Y. He, "Fault tree analysis of software reliability allocation," in *Proc. of The 7th World Multiconference on Systemics, Cybernetics and Informatics*, vol. Volume II - Computer Science and Engineering. Orlando, USA: International Institute of Informatics and Systemics, Jul 2003, pp. 460–465.

[20] P. H. Feiler, D. P. Gluch, and J. J. Hudak, "The architecuture analysis & design language (AADL): An introduction," Carnegie Mellon University, Tech. Rep. CMU/SEI-2006-TN-011, February 2006.

[21] J. Xiang and K. Yanoo, "Automatic static fault tree analysis from system models," in *Proc. of The 16th IEEE Pacific Rim International Symposium on Dependable Computing (PRDC'10)*. Tokyo, Japan: IEEE, Dec 2010, pp. 241–242, fast abstract.

[22] ——, "Formal static fault tree analysis," in *Proc. of the 6th International Conference on Computer Engineering and Systems*. Cairo, Egypt: IEEE, Dec 2010, pp. 280–286.

[23] J. B. Dugan, S. J. Bavuso, and M. A. Boyd, "Fault trees and Markov models for reliability analysis of fault-tolerant digital systems," *Reliability Engineering & System Safety*, vol. 39, pp. 291–307, 1993.

[24] G. Merle, J.-M. Roussel, J.-J. Lesage, and A. Bobbio, "Probabilistic algebraic analysis of fault trees with priority dynamic gates and repeated events," *IEEE Transactions on Reliability*, vol. 59, no. 1, pp. 250–261, 2010.

[25] Z. Manna and A. Pnueli, *The Temporal Logic of Reactive and Concurrent Systems*. New York: Springer-Verlag, 1992.

[26] J. B. Dugan, S. J. Bavuso, and M. A. Boyd, "Fault trees and sequence dependencies," in *Proc. of Annual Reliability and Maintainability Symposium*. IEEE, 1990, pp. 286–293.

[27] R. E. Harper, J. H. Lala, and J. J. Deyst, "Fault tolerant parallel processor architecuture overview," in *Proc. 18th Symp. Fault Tolerant Computing*, 1988, pp. 252–257.

[28] R. E. Harper, "Reliability analysis of parellel processing systems," in *Proc. 8th Digital Avionics Systems Conf.*, 1988, pp. 213–219.