Automatic Synthesis of Dynamic Fault Trees from UML System Models

Ganesh J Pai, Joanne Bechta Dugan
Department of ECE, University of Virginia, Charlottesville, USA
{gpai,jbd}@virginia.edu

Abstract

The reliability of a computer-based system may be as important as its performance and its correctness of computation. It is worthwhile to estimate system reliability at the conceptual design stage, since reliability can influence the subsequent design decisions and may often be pivotal for making trade-offs or in establishing system cost. In this paper we describe a framework for modeling computer-based systems, based on the Unified Modeling Language (UML), that facilitates automated dependability analysis during design. An algorithm to automatically synthesize dynamic fault trees (DFTs) from the UML system model is developed. We succeed both in embedding information needed for reliability analysis within the system model and in generating the DFT. Thereafter, we evaluate our approach using examples of real systems. We analytically compute system unreliability from the algorithmically developed DFT and we compare our results with the analytical solution of manually developed DFTs. Our solutions produce the same results as manually generated DFTs.

Keywords: Dynamic fault trees, Fault tree synthesis, Reliability analysis, UML

1. Introduction

To make the design process of a dependable computer-based system effective, we must estimate the system reliability requirements during the conceptual design stage itself. If system designs can be analyzed for their reliability characteristics as soon as they are available, both design time and associated resources can be reduced allowing designers to decide if redesign is required. Tradeoffs are most effective when key attributes of the system, such as performance and reliability, can be calculated much earlier during the critical design stages.

Many fault-tolerant systems are complex because of redundancy, reconfigurability and various interactions between their components. One practice in realizing these complex systems is to use block diagrams annotated with behavioral descriptions in an architectural description language. A promising approach in system modeling is to use advanced formalisms such as constraint automata [1]. Although these design-stage modeling approaches are sufficient to simulate a system and estimate its performance, they do not convey enough information to support automatic reliability analysis. Instead, information regarding spares, error propagation and redundancy is generally supplied in the accompanying specifications and requirements documents. While there are tools that automatically solve reliability models, constructing the models themselves requires perusing these documents and largely remains a manual procedure. As a result, the reliability analysis process often takes a long time. Hence the design and the reliability analysis stages are usually separated and the results of reliability analysis are available only much later in the engineering cycle. By performing reliability analysis in parallel with system design, we can alleviate this problem and aid early validation of the architectural design. This is the primary motivation behind our work.

Our approach is to embed statistical information needed for reliability analysis and information related to component redundancy, reconfiguration or dependencies within the architectural design itself. Thereafter, we automatically generate a reliability model, which can be analyzed using existing solvers.

The primary reason why the UML was chosen was our sponsor's constraints. The UML is "a standard graphical language used to visualize, specify, construct and document the artifacts of a software-intensive system" [2]. It provides formal constructs to deal with varying levels of modeling abstraction to visualize and specify both the static and dynamic aspects of the system. Although these constructs have no formal semantics, there are generic frameworks for formalizing the UML [3]. While the UML was designed with the intent to model software systems, the logical models produced using UML constructs can be used to model hardware systems as well. Extensibility is a powerful feature of the UML: it has mechanisms like stereotypes, tagged values and constraints with which the semantics of model elements can be customized and extended. It also provides conceptual tools to manage the complexity of system design. The models generated in the UML can be connected to a variety of object oriented programming languages such as C++ and Java, or to architectural description languages such as VHDL [4].

The UML provides designers with a variety of diagrams to graphically model a system. The rationale is that the problem of designing a complex system is best approached through a set of concise and independent views of the system, instead of a single viewpoint.

Proceedings of the 13 th International Symposium on Software Reliability Engineering (ISSRE'02)
1071-9458/02 $17.00 © 2002 IEEE

Fault tree analysis is a popular analytical technique for reliability estimation [5]. Fault trees are graphical models that represent logical relationships between events that lead to system failure. They also provide a systematic mathematical framework for analyzing potential causes of failure. From a design perspective, they allow the designer to understand the ways in which a system may fail. Fault trees comprise basic events connected by gates, in a logical path, to a top node that represents system or subsystem failure. Dynamic fault trees (DFTs) [6] are introduced as extensions to static fault trees to model sequence dependencies, functional dependencies and spares in a fault-tolerant system. Owing to the complexity of these systems, corresponding lower level reliability models such as petri nets and markov chains are themselves complex. Other combinatorial models like reliability block diagrams are insufficient to model dynamic failure behavior. DFTs provide an elegant abstraction to model both non-combinatorial and combinatorial failures in these systems.

We first develop a methodology, based on the UML for modeling fault tolerant computer-based systems. The focus is on specifying the criteria needed to allow reliability analysis to proceed in parallel. Next, we develop an algorithm to automatically synthesize DFTs from the UML system model. It works on the logical system structure inherent in the UML model, and can also be applied to non-UML system models if similar logical structures can be extracted from them. This is our main contribution. The goal is to permit designers, who have first hand knowledge of the system, to also specify the failure or success criteria. We want to have the modeling methodology be able to convey, as far as possible, not only what the designer has in mind but also the information necessary for reliability analysis.

The rest of our paper is organized as follows: Section 2 briefly introduces dynamic fault trees and presents current approaches to automatic synthesis of dependability models from system designs. In section 3, we describe our UML based modeling methodology with an example of a co-designed and reconfigurable system. We present an algorithm to automatically generate DFTs from the UML system model in section 4. In section 5 we present the results of reliability analysis using our methodology over a broad class of real systems. Reconfigurable systems, co-designed systems, and hierarchical systems are chosen for the illustrative examples. Section 6 concludes the paper and identifies future work.

2. Background

There are two approaches to constructing reliability models from the engineering model of a system. One technique is "simulation-based" and synthesizes reliability models from behavioral models [7]. Behavioral models are rather complex and reliability models can be obtained from them only once the system has been completely described. The second approach is to use functional models and obtain analytical solutions to estimate reliability. Analytical techniques for reliability assessment are desired because they are versatile and allow the designer to perform "what-if" analyses. Our belief is that functional models are sufficient for providing accurate results in estimating system reliability [15]. Analytical models, *e.g.* DFTs, are relatively easy to solve and are easier to use for qualitative analysis than when using simulation. The main advantage of using analytical models is that they can be generated during the design stage from the engineering model itself.

2.1. Dynamic fault trees

Fault trees are used to graphically and mathematically model events that can cause system failure. Fault tree analysis is both qualitative and quantitative, and may be used in analyzing failure modes of critical systems. Besides this, they are hierarchical enabling them to model large systems. In qualitative analysis we use minimal cut-sets to determine the presence of single points of failure in the system or the combination of events that will lead to unsafe operation. In quantitative analysis, we determine the probability of occurrence of the top event of the tree, given the probability of occurrence of the basic (failure) events. Classical fault tree analysis considered only static fault trees with *AND*, *OR*, and *M-of-N* type logic gates. DFTs, extensions to static fault trees, allow modeling of spares, sequence, and functional dependencies, which static fault trees cannot model. Conversion of dynamic fault trees to markov models [5] allows expression of certain kinds of sequence dependencies. We use *Galileo* [8, 9], a tool that uses a combination of static and dynamic fault trees, to solve the DFTs that we generate from UML system descriptions. The tool incorporates the BDD solution of static fault trees [10] and the DIFTree methodology [11].

Besides this, fault trees can be used to model software [12]. The idea is to use a fault tree to represent the potential failures associated with each function that a software system is intended to perform, *i.e.* representing software modules that may produce failure as basic events and connecting these to the top node through a set of gates that defines the logical path to failure. These basic events in the fault tree model of software can be decomposed to the point where they can be tested or detected by reliable failure detection mechanisms. Thus, the fault tree for a software system and its associated analysis can help the designer to discover particular classes of software failure that leads to system failure. The ability to model hardware and software systems and their static and dynamic aspects using fault trees is a forté, because most fault-tolerant systems today are software intensive and dynamic. Thus, a combination of static and dynamic fault trees can elegantly model a large variety of dependable systems.

2.2. Related work

Our idea of using UML system models [21] and their translation into reliability models is not new. The *HIDE*

framework [13] incorporates transformation of UML designs into timed petri nets (TPNs) for model based dependability evaluation. In *HIDE*, the UML is primarily used for designing the system after which subsequent dependability analysis is performed. UML structural views are used to convey relevant dependability information and are then converted into an intermediate model (IM). The IM is actually the dependability model and it is defined as a hypergraph, where every node corresponds to a component described in the UML model and every arc corresponds to the associated UML relationship. The TPN model represents the broad class of deterministic and stochastic petri nets. Each module of the IM is converted into a semantically equivalent TPN subnet. The system, as a whole, comprises an interconnection of these subnets. This work also creates a set of extensions for the UML to identify dependability structures and to define dependability parameters. Specifically, a redundancy manager, a variant and an adjudicator are prescribed as three stereotype-based extensions to the basic UML constructs, to indicate the objects that are used for redundancy management, voters and comparators respectively. Besides this, tagged value based extensions are used to indicate dependency parameters. One of the goals of the *HIDE* framework is to analyze the dependability attributes of a system while it is still being designed. The validation of designs described in the UML is the main goal of the *HIDE* framework [20].

Our approach differs from the HIDE approach in the following manner: Primarily, our target reliability models are dynamic fault trees instead of TPNs. We extract the logical system structure and obtain structural redundancy, at translation time, from the UML model, while failure parameters are conveyed as a property of the structural model of the component. We develop a scheme to capture and classify dependencies within the system. Further, we address the software to hardware mapping issue as well as dynamic reconfiguration in the presence of failures.

OpenSESAME [14], *RIDL* [15] and the *RBD translator* [16], represent other related work in translating design-stage system descriptions into reliability models. Briefly, the *OpenSESAME* approach attributes a set of parameters for every component, and uses a separate graphical expression of dependencies. The tool accepts a high level graphical model of a system described using redundancy structure diagrams. Failure propagation diagrams are used to indicate dependencies that arise as a result of error propagation. Quantitative availability analysis is then performed on the system by automatically translating the system description into generalized stochastic petri nets (GSPNs). *RIDL* is a graphical design language for modeling dependable systems and has a functional modeling framework that facilitates automatic translation of a system model described in the *RIDL*, to fault trees. *RIDL* graphically projects a functional view of the system by considering each component in the system as a black box with associated statistical failure information. Basic building blocks, interconnects and success criteria, along with their associated semantics are defined to capture the system schematic. It explicitly defines constructs to represent components and classifies them as being essential, redundant, supporting, irrelevant, repeated, support, *N*-of-*M* type, or as spares. Directed arcs are used to indicate error propagation. The *RBD translator* uses elementary UML modeling principles developed in [16] and [17] to convert the description into reliability block diagrams.

Our approach differs from these three approaches in the following ways: We do not define components as being essential, redundant, or otherwise as in *RIDL*. Instead, this information is inherent and is interpreted during translation. A key observation is that both *RIDL* and *OpenSESAME* offer only a singular projection into the system. However, our approach allows the designer to model the system from at least two views: a purely functional perspective, and a behavioral perspective. Besides these, we address the issues of reconfiguration and hardware to software mapping, which has not been addressed in *RIDL*. Unlike *OpenSESAME*, we indicate dependencies in a single diagram of the logical system structure. Moreover, error propagation is explicitly shown using stereotypes that extend the dependency construct of the UML. We indicate probabilities of error or failure propagation as attributes. Finally, we generate DFTs from our system description instead of GSPNs or RBDs.

3. Modeling fault tolerant systems in the UML

Current reliable systems usually have redundant components (both hardware and software) and may be reconfigured on the fly to allow system operation in the presence of faults. Therefore, capturing redundancy information and reconfigurability issues unambiguously within the UML model is important. Using UML constructs we would like to have a system model that primarily conveys the following:

- The number of copies of each component that exists in a redundant system,
- The minimum number of components required for successful system operation.
- In the presence of spares, the active components that are replaced by the spares when failures occur. If a pool of spares exists, the components that are drawn from this pool to replace the faulty components.
- Dependencies that exist within redundant components, and the effect of failure of one component on others.
- Software to hardware mapping as well as any mapping changes or reconfiguration in response to failures.

We would also like the model to be as complete and correct as possible, and as close as possible to the designer's perspective of the system. The UML has semi-formal syntax and semantics that are specified using a meta-model. One approach to modeling in UML is to augment the UML with an existing and well-accepted architectural description language (ADL) [18]. However, this results in

making the UML non-standard, although it would provide the modeler with greater freedom and flexibility due to the combined constructs of the UML and the ADL. Instead, we introduce stereotypes to constrain the UML to modeling hardware and dependable systems. Stereotypes, tagged values and constraints are extension mechanisms with which the semantics of model elements can be customized and extended. The extension mechanisms are used to restrict the way the UML meta-model is employed in constructing and interpreting UML models. Although stereotypes create new modeling constructs they are valid UML interpretations. Thus a model described in a syntax that does not deviate from the UML constructs permits it to be universally understood. Moreover, it also allows existing commercial UML tools to easily manipulate the model.

Explanation of our modeling methodology is necessarily anecdotal. We use an example of a reconfigurable and co-designed system, namely the mission avionics system (MAS), to describe our approach.

3.1. The mission avionics system

The mission avionics system (MAS) is a highly redundant mission-critical and safety-critical system, with both hardware and software components. It comprises five critical subsystems whose success is crucial for system success: the crew-station control system, the scene and obstacle control system, the local path generation sub-system, the system management subsystem and the vehicle management subsystem. Figure 1 shows the block diagram model for the MAS illustrating these subsystems and their interconnections. Initially, we assume that every subsystem is a software module that has been mapped to a dedicated processor. Each processor (and the mapped software) has a hot spare backup unit that takes over control from the primary processing unit if a failure or error is detected. This is the static model for the mission avionics system without any reconfiguration. *CrewStnA* represents the crew-station control system primary processor, while *CrewStnB* is its hot spare backup processor. Similarly *S&OA, S&OB, PathGenA, PathGenB, SysMgtA* and *SysMgtB* represent the primary and hot spare backups for the scene and obstacle, local path generation and the system management subsystems respectively.

The vehicle management system has additional functionality and requires the use of two processing units *VM1A* and *VM1B.* Each of these units has its respective hot spare backups, *VM2A* and *VM2B.* There is a pair of cold spare processors shared by the first four subsystems (*Spare1* and *Spare2*) and another pair of cold spares for the vehicle management system (*VMSp1* and *VMSp2*). The first pool of spares (*Spare1* and *Spare2*) is used as backups in the event of two failures in any of the four subsystems (crew station, scene and obstacle, local path generation and system management). These spares do not cover any faults that may occur in the vehicle management subsystem. Instead, they have their own dedicated pool of cold spares (*VMSp1* and *VMSp2*) that cover two failures. A triplicated mission management bus (*MMBus*) interconnects these five subsystems. Besides this, memory units (*Memory*1 and *Memory2*) are connected to four sub-systems by a triplicated background data bus (*B/G Data Bus*). The vehicle management system has its own duplicated vehicle management bus (*VMBus*).

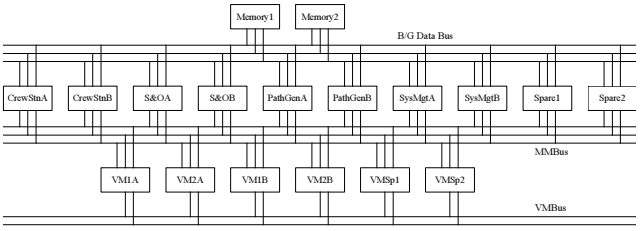

Figure 1. MAS block diagram

System failure occurs if both memories fail, if all of the duplicated or triplicated buses fail, or if any of the critical subsystems fail. Clearly, neither these failure criteria nor the spare allocation strategies are apparent from the block diagram.

3.2. Modeling hardware and redundancy

Class and object diagrams are used to logically model the structural associations between the components of the system. Our work is an initial attempt to present that reliability analysis can be performed in conjunction with design. We anticipate that UML features such as the object constraint language (OCL) or other diagrams *e.g.* component diagrams, may also be used. In this paper, however, we use class, object and deployment diagrams mainly because these are intuitive. Every component in MAS can be considered as an instance of a class (an object) with properties, attributes and operations. The statistical reliability information of any component (such as failure rate and distribution, coverage, restoration, etc.) is an attribute of that component. Hence, this information is specified among the attributes in the class construct. The class construct also allows the designer to specify cardinality of the classes. Essentially, the number of instances of the component that exist, when the system is operational, is present within the UML class construct itself. Figure 2 shows the class diagram for the static MAS. The class diagram conveys information about the static structure of the system. Each of the components in the system has been modeled as a class with attributes, and each of the classes has the appropriate name of the component. The classes are labeled with the stereotype *<<hardware>>*, to indicate that the components being modeled are hardware components.

The solid lines without arrowheads, that connect the classes, represent bi-directional associations between the components. Associations are structural relationships and are primarily used to connect structural constructs like classes. Each association relationship has a number indica-

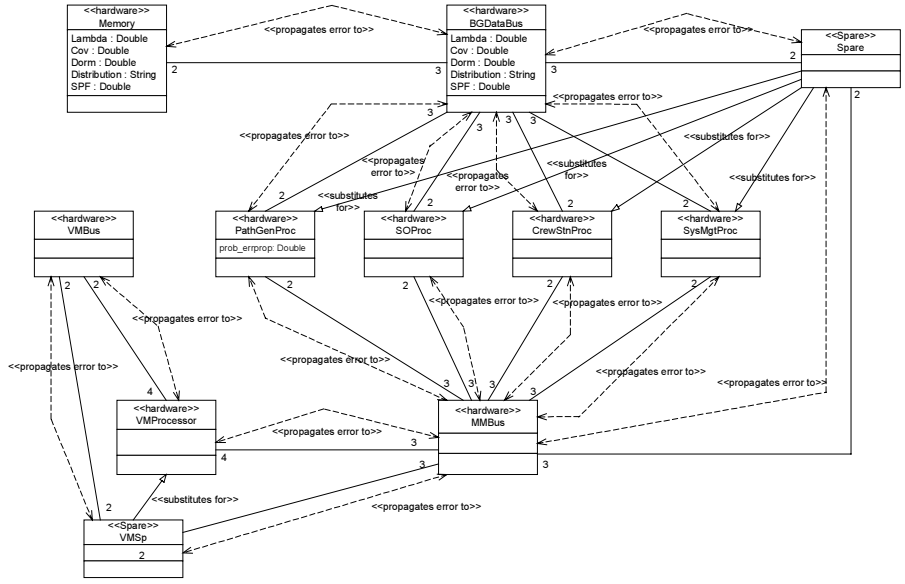

Figure 2. UML class diagram for the MAS (static)

Although class diagrams provide a high level model of the system structure, the abstraction causes some information loss. Specifically information about how the individual class instances are connected to each other has been lost. For fully interconnected systems such as the MAS, this information loss is inconsequential. In systems that are not fully interconnected, we use the object diagram to show the structural view of the system in more detail. The object diagram simply instantiates the classes defined in the class diagram to show the details of all the object interconnections that exist in the system. Thus, class diagrams allow us to define how many components exist in the system.

ting the multiplicity of the association. The multiplicity shows the cardinality of the classes on either end of the association. The dependencies between components are represented in the UML as dotted lines with arrowheads, while the solid lines with solid arrowheads represent the generalization relationships.

Dependencies are semantic constructs that connect two or more structural or behavioral things, and they are used to show that a change in the independent thing may lead to a change in the semantics of the dependent thing. In Figure 2, dependencies are annotated with the *<<propagates error to>>* stereotype, to show that the dependency between components is established because of error propagation. The generalization relationship relates objects of a child element to a parent element. A generalization indicates that a structural thing "*is a type of*" the thing it is related to, and that it may be substituted for the parent.

This is a powerful UML construct because it is useful while modeling dependable systems to show that some components can substitute for or replace others. In Figure 2, the generalization is stereotyped as *<<substitutes for>>* to indicate that the class *Spare* can substitute for the classes *PathGenProc*, *SOProc*, *CrewStnProc* and *SysMgtProc*. The dependency, association and generalization relationships are the some of the basic relational building blocks of the UML.

Virtually all information that was conveyed from the block diagram is also apparent from the class diagram. Besides this, the class diagram also conveyed additional information about dependencies and interactions among the components. Attributes related to these dependencies, such as the probability of error propagation is conveyed as an attribute of the classes. The attributes for the components have been shown for two of the classes to indicate how this information is conveyed from the class diagram.

From a redundancy perspective, we need to know the minimum number of components that are needed for successful system operation. Deployment diagrams are used to model these system success criteria. Figure 3 shows the deployment diagram for the MAS system. The deployment diagram shows only one memory, four processors and the software that has been mapped to them, a single *MMBus*, *VMBus* and *BGDataBus*, and two *VMProcessors*. These are precisely the success criteria, and this is the information that we are interested in obtaining from the model. The deployment diagram is a snapshot of the system at runtime and shows the minimal operational version. Therefore, so long as the deployment diagram shows the minimal success criteria, the particular configuration of the deployed structure is inconsequential for generating the reliability model.

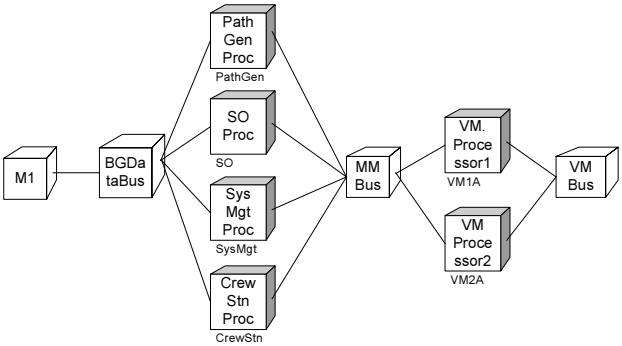

Figure 3. Deployment diagram for the MAS

Complexity management involves being able to modularize the system design to permit modeling of each module in detail, or being able to model hierarchy in the system. The UML not only provides different diagrams to model the structural and behavioral aspects of a system in

varying levels of detail, it also provides the concept of *packages* and *subsystems* that allow us to organize modules into manageable portions. Capturing hierarchy using our modeling methodology is straightforward. We simply model the details of the upper level or the lower level in another class or object diagram *e.g.* consider the *Memory* class of the MAS; we would model the internals of this component using another class diagram that describes its structural organization. We would use the object diagram to describe the internal connections in greater detail.

Similarly, if a larger system (say an aircraft) actually contains the MAS the UML model of this system itself has a class *MAS* that abstracts the complexity of Figure 2 into a single class construct. Similarly, each class diagram would have its associated deployment diagram that captures the success criteria.

3.2. Modeling spares and dependencies

Spares are components that provide redundancy by replacing faulty components. Since spares are replicas of a particular class of components, we indicate spares in the class diagram but not in the deployment diagram. We showed how the spare components were modeled (Figure 2) as separate classes with a generalization relationship between the spares and the components that they replace. Recall that a generalization relationship is used to represent an "*is-a-type-of*" or a substitution relationship. The generalization indicates that the instance of the spare class is a type of the instance of the class that it replaces and may substitute for it. We also define a <<*spare*>> stereotype for the class itself identifying it as a spare, with a simple generalization relationship between the respective classes. Depending on how and when the spares are used, the spares are classified as hot, cold or warm spares. The dormancy factor attribute of each component is used to identify if the component is a spare and determine the type of spare it is.

Modeling dependencies is straightforward, using the dependency construct. We have already shown in Figure 2 how error propagation dependencies were modeled in the MAS. Dependency between the hardware and software is modeled analogous to dependencies among hardware components. We define the <<*Runs On*>> stereotype to indicate that the dependency is one where the software runs on the hardware. The dependency is unidirectional from the software module to the hardware module. It is implicit from this that failure of the hardware module causes the software to fail, but not the reverse. Horizontal dependencies among two or more classes are represented by using the dependency construct and the corresponding stereotype to connect the classes. Vertical dependencies exist across layers of hierarchy and these are represented using a class that is common to the layers across which the dependencies exist. An alternate way to represent dependencies across layers is to collapse both layers into one diagram and represent the dependencies among the classes in these layers. We introduce a tagged value *"SEQDEP = a"* indicating the sequence enforcing order for a dependency. We also introduce the constraints *{OR}* and *{AND}* to indicate the possible relation between two or more dependencies *i.e.* if an object depends on two components then it may be dependent on either of the two or both of them.

Table 1 provides a list of stereotypes that we define to identify dependencies, generalizations and the type of component that is being modeled, and their corresponding fault tree realizations. Error propagation can occur not only between two or more hardware components, but also between a hardware component and the software that has been mapped to it. Besides this, an error caused in software may propagate to another software component. In either case, we use the <<*propagates error to*>> stereotype. The probability of error propagation is defined along with other attributes in a class, *i.e.* it is considered as a property of the component causing the error. We can also define attributes that define the rate with which faults are activated for both hardware and software components, and include them as attributes.

3.3. Modeling reconfiguration

Reconfiguration is an important property in dependable systems and it is the ability to change system structure or behavior, either for the purpose of providing higher performance or to maintain system service in the presence of failures. A structural reconfiguration might be the reallocation of resources such as a redistribution of the load on the system onto available computing units, or a change in the mapping between hardware and software resources. Behavioral reconfiguration occurs when the system has to show a predefined failure behavior such as graceful degradation. Reconfiguration either in the structure or behavior of the system affects the overall reliability of the system. In this paper, we only address the issue of modeling structural reconfiguration because the events leading to reconfiguration directly affect the construction of the target dependability model.

Incorporating reconfigurability into the MAS adds an interesting aspect to the system as compared to hardware redundancy. Suppose that all processors are identical (which in fact they are in reality) and that the software mapped to the processors can run on any of them. We further suppose that two of the software subsystems, scene & obstacle, and local path generation, have alternate minimal versions that provide reduced functionality and require comparatively fewer computing resources. The minimal versions are switched in, to replace the full versions in one of following scenarios. *MinSO* (the minimal version) replaces *FullSO* (the full version) if the full version of the software fails. Similarly, *MinPathGen* replaces *FullPathGen*. We assume that both the minimal versions together require only one processor to execute, while the full versions each require two (one primary processor and one hot spare). If both the processors fail and there are no spares

Table 1. Stereotype definitions for dependency relationships, generalizations and classes

Stereotype	*Stereotype Meaning*	*Fault Tree Realization*
<<propagates error to>>	Propagation of error caused due to hardware fault (transient or permanent) to direction indicated by dependency	Error propagation probability associated with the class as an attribute, used as a basic event parameter
<<Runs On>>	Dependency because of software mapping to hardware components. Failure of the hardware component causes the software to fail.	FDEP trigger and dependent components
<<requires>>	Dependency to indicate that the target component is required by the component from where the dependency originates.	FDEP trigger and dependent components
<<spare>>, *<<hotSpare>>, <<coldSpare>>, <<warmSpare>>*	Spare component and type of spare. (Alternately, the dormancy factor can be used to identify the type of spare)	CSP, HSP, WSP
<<substitutes for>>	Generalization that indicates that the spare substitutes for a particular component	Determines spare and primary connection to HSP/CSP/WSP
<<hardware>>, <<software>>	Hardware and software components respectively.	-

available to replace both, then the minimal versions of both modules replace the full versions and run on a single processor *i.e.* irrespective of whether one full version is still functioning, if two processors of the other essential module fail, then both modules are replaced with their minimal versions running on a single processor (if there are no spares to replace the failed processors).

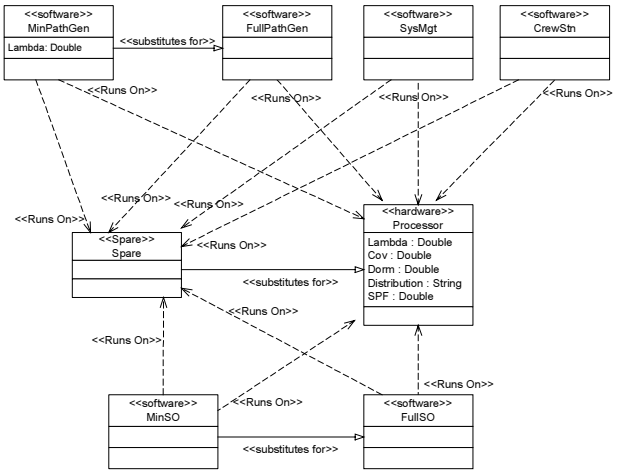

Figure 4. Class diagram for MAS with reconfiguration

We assume that a mechanism exists where the failure of a processor will cause the software mapped to it to be remapped to the spare that is switched in. Conceivably, the minimal versions and reconfigurability give the system a chance to tolerate or recover from massive failures (such as failure of the two pro-cessors and their respective hot spares running *SO* and *PathGen*). The block diagram is insufficient to model this complex nature of reconfiguration. Figure 4 shows the modified class diagram incorporating the minimal versions and the hardware to software mappings for the first four sub-systems.

The modified class diagram for the *VMProcessor* class is similar. Only the additions to the class diagram of Figure 2 are shown. The processors have been modeled as a single class because all the processor instances belong to the same class. The mapping between the software and the processors is shown using a *<<Runs on>>* stereotype. The *<<substitutes for>>* stereotype in the generalization between the full software versions and the minimal software versions indicates that the minimal versions replace the full versions. The main advantage of this representation is that it shows the true nature of the system *i.e.* the software can run on any of the processors. More importantly it shows that the software to processor mapping is transient and that the mapping is reconfigurable. Since the reconfiguration activity is itself a behavior, it has to be shown explicitly using an activity or statechart diagram (Figure 5).

Activity and statechart diagrams are the behavioral modeling constructs of the UML. Figure 5 essentially indicates the triggers that lead to reconfiguration and the actions taken during reconfiguration. Our interest in this diagram is to determine how the occurrence of certain failures (basic events in the fault tree) will affect the construction of the target model. The reconfiguration activity diagram shows that the full versions of both *SO* and *PathGen* are replaced by their respective minimal versions, if

either of the modules fails or if two processors fail. Further, if two processors fail and two spares are not available, the minimal versions are both mapped to one spare. The translation process for mapping these reconfiguration activities into the correct dependability model is based on keywords that we have used, namely *Fails, Replace,* and *With*.

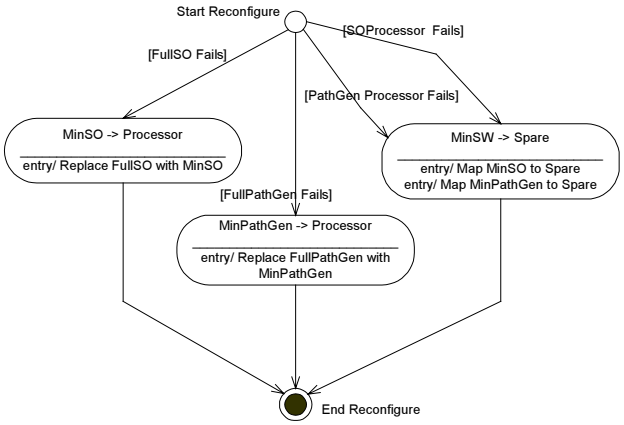

Figure 5. Reconfiguration activity diagram for MAS

The key points to be identified in this activity diagram are the modules that failed and the modules that replace failed components. These are analogous to basic event failures and spares in the fault tree. We now describe our algorithm to convert the UML model to a dynamic fault tree.

4. Automatic DFT synthesis

For analyzing our approach, we used Rational Rose, a commercially available UML modeling tool. Our approach to compiling the UML model to generate the DFT is simple: we use a customized parser to extract the logical model of the system, with all information about system structure and the embedded reliability information. Conceptually, this can be considered as creating an object that logically encodes the system structure and the reliability information. The actual synthesis algorithm works on this logical structure to generate the dynamic fault tree.

4.1. The logical system object

We express the fault tree in code that is understandable by Galileo. However the expression of the fault tree can be customized to be input to other fault tree analysis tools. The parser is tuned to recognize the keywords used by either Rose in its internal representation of the UML. The internal representation is a monolithic text file i.e. information from the class, object and deployment diagrams are all stored in a single file. For hierarchical modeling in Rose, the hierarchies have to be created separately and cannot be created from the tool itself. Therefore a model for each hierarchical level is stored as a separate file.

We generate the object encoding the system structure (Figure 6) as follows. The association relationships from the class and the object diagrams are used to obtain the system structure. The cardinality of the classes is used to obtain how many components are present in the system; the deployment diagram and the object diagram are used to determine how many components are needed, and the specifics of class associations. Together, these determine redundancy information of the system. In figure 6, *itsM*, *itsN* and *itsK* represent the number of components present, the number needed and the number that will cause failure, respectively. Stereotypes defined for the UML constructs are used to build the corresponding fault tree analogues, however not all stereotypes are used during the generation of the DFT.

```
Sys_Object {
   itsName; itsStereotype;
   itsAssociations; itsGeneralizations;
   itsDependencies {
            itsStereotype; itsTaggedValue;
            itsConstraint; };
   itsInstances {
    itsInstanceNames; itsInstanceGeneralizations;
    itsInstanceAssociations; };
   itsAttributes { ItsAttributeNames;
               itsAttributeValues;};
   itsMultiplicity {itsM; itsN; itsK; };
   }
```

Figure 6. Structure of the logical object for the system model

The attributes are used obtain the probabilistic information needed for reliability analysis *i.e.* failure rates, dormancy factor, coverage, restoration, distribution and SPF. The reconfiguration activity diagram is used to account for the reconfiguration in the system. Multiple model files are used to determine the hierarchies in the system and the model of each tier in the hierarchy is parsed to create its own system object, which is then translated into its corresponding DFT. Tagged values are used to determine the input order or sequence for connecting the basic events to the spare gates in case a primary component shares more than one spare. The tagged values are specified on the dependency, labeled with the appropriate stereotype.

The DFT generation algorithm then uses this object to create the fault tree for the system. Figure 6 only shows the structure of this object. Depending on the number of classes within a model and how these classes are associated or depend on each other, there will be as many entries in this object as there are classes with the respective values for each of the fields.

4.2. Algorithm for DFT generation

Our algorithm for generating the DFT from the object that encodes the system structure is a three-pass algorithm. In the first pass we compute the count of the number of classes, the number of instances per class and the number

of dependencies per instance of a class. A basic event (BE) is created from each class instance and all BEs are connected to a top level OR gate. Creation of the BEs includes assigning each BE parameter with the respective parameter from the object. Thereafter, for every class that is identified in the class diagram we perform the following steps:

- From the object encoding the structure, we determine how many components exist and how many cause failure. Appropriately, an OR, AND or a K-of-M gate is defined. If the system that is modeled is completely static, then we need go no further because a static system can be modeled using these three gates.
- For a dynamic system, the dependencies for each BE is determined and connected to the appropriate functional dependency (FDEP) gate normal inputs.
- Next, the BEs that are independent in the dependency relationship are connected to the correct FDEP gates as primary inputs *i.e.* as the trigger to the FDEP. If the dependencies have an *OR* / *AND* constraint, then either an *OR* / *AND* gate is created and the trigger BEs connected to this gate.

In the second pass, the spare gates (HSP, CSP or WSP for hot, cold and warm spares respectively) are identified from the logical structure. In this case, for each class the stereotype is used to determine if the spare should be translated into an HSP, CSP or WSP. Once the spare gates are identified for each instance of each class, the spare gates are created and the respective basic events are connected to either the primary or the normal connections of the spare gates depending on the generalization relationships *i.e.* if an object is a spare for another object then the corresponding generalization entry is filled with the name of the BE that it is a spare for. Therefore, the generalization entry is connected to the primary input of the spare gate and the object itself is connected to the secondary. If the basic events now connected to the spare gates were originally connected to other gates (such as OR, AND or K-of-M gate) then the spare gates are then connected to these gates also. The links from the BE to the original gates are then deleted. We can use multiple linked list type data structures for this purpose, and the idea is to simply update the pointers, as new gates and the correct connections are determined in each pass. At this stage, the DFT is complete if there is no reconfiguration in the system that has been indicated with an activity diagram.

In the third pass, the DFT created from the first two passes is modified to include the reconfiguration in the system (if it exists). Recall that reserved words are used in the activity diagrams that show reconfiguration. These reserved words are then used to determine the events occur that will cause or '*trigger'* reconfiguration or system failure. Accordingly, the basic events represented in the DFT by these classes are connected to the FDEP gates and the spare gates. Figure 7 shows our algorithm for generating dynamic fault trees.

In summary, our DFT generation algorithm can generate the AND, OR, K-of-M, CSP, WSP, HSP and FDEP gates. Figure 8 shows the DFT that is generated using our algorithm for the reconfigurable MAS system (Figures 2 and 3). We have shown a reduced version of the fault tree and we have not expanded all the basic events. However, we have indicated how many basic events are present in each sub-tree. The DFT identifies all the failure criteria as well as the reconfiguration in the system from the UML model.

5. Example Systems

We evaluated our modeling methodology and the DFT synthesis algorithm using three types of systems; namely co-designed and reconfigurable systems, hierarchical systems and phased mission systems. In this paper, we present the results of modeling and reliability analysis of only the first two types.

The MAS (which we have already explained) is chosen as an example of a co-designed and reconfigurable system. A relatively complex configuration of the fault tolerant parallel processor (FTPP) is chosen as an example of the hierarchical system. First we analyzed the MAS with and without reconfiguration; then we analyzed the FTPP both as a hierarchical and a non-hierarchical system.

As a validation effort (*i.e.* informally), we used our algorithm to generate DFTs for the MAS configurations from [15] and [19] respectively and we compared our solution with the DFTs that were manually generated. For the first example, our algorithm produced a DFT that is identical to the one presented in [15]. Figure 9 shows the complete fault tree that was obtained for the second configuration. The DFT that we generated differs from the one in [19], as shown by the dotted lines. The DFTs that were obtained are structurally different but logically equivalent. We analytically solved them using the Galileo tool, using the same basic event parameters. In all our analyses, an exponential failure distribution was assumed. Table 2 lists the parameters that we used for analysis and also summarizes the results of the analysis *i.e.* the computed top event unreliability *(Q)* for the MAS. Although the DFT we produced differed, the analysis results clearly indicate that our solution is correct as well. In fact, the result is not surprising, simply because our solution can be reduced to the solution of [19]. Since our DFT is algorithmically generated, it lists *all* failure criteria. The sub-tree shown by the dotted lines are actually inconsequential because they have already been covered by the *S&O*, *PathGen*, *Crew* and *SysMgmt* sub-trees.

As a second set of analyses, we modeled a relatively complex FTPP configuration (#2 from [5]) both as a hierarchical system and as a non-hierarchical system. We present the results of these analyses and the parameters that we used, in table 3. In both cases, the DFTs that were generated were structurally different from the manually generated ones. We do not present the fault trees in this paper due to lack of space. However, the complete fault trees that we generated are documented in [17].

```
// Pass 1 - BE, AND, OR, K-of-M and FDEP

Define TopLevel Gate as OR;
j = classCount = Number of Classes (Sys_Object);
n = dependencyCount = Number of Dep (Sys_Object.itsName);
p = instanceCount = Number of Instances (Sys_Object.itsName)

For (i = 0, i <= classCount, i ++) {
      Create BE from Sys_Object.itsInstances;  }
EndFor;

While (j != 0) {
  Assign Sys_Object.itsAttributes to BE.itsParameters;
  Connect BE to TopLevel Gate;
  If ((Sys_Object.itsK < Sys_Object.itsM) &&
                        (Sys_Object.itsK != 0,1))
            { Define KofM Gate; }
  ElseIf  (Sys_Object.itsK = Sys_Object.itsM = 1)
            { Define OR Gate; }
  ElseIf  ((Sys_Object.itsK = Sys_Object.itsM) &&
          (Sys_Object.itsM > 1)) { Define AND Gate; }
  EndIf;
  Connect BE to defined Gate In;
  Connect defined Gate to TopLevel;
  Delete BE link to TopLevel;
  For (p > 0; p --)
  {
    If ((Sys_Object.itsDependency exists) && (FDEP !exist))
    {  Define FDEP Gate;
       While (n != 0) {
         If (Sys_Object.itsDependency.itsConstraint = OR||AND)
         {   If (OR || AND Gate Exists) {
                 Connect OR||AND Gate Out to FDEP Trigger In; }
             Else {
               Define Gate = Sys_Object.itsDependency.itsConstraint;
              Connect Gate Output to FDEP Trigger In;
              Connect Sys_Object.itsDependency to Gate In; }
            EndIf;  }
          Else {
            Connect Sys_Object.itsDependency to FDEP Trigger In;}
          EndIf;
          n-- ;
        } EndWhile;
    }
    ElseIf  ((Sys_Object.itsDependency exists) && (FDEP exists))
       { Connect FDEP Normal In to BE; }
    Else {
       Connect FDEP Normal In to BE;}
    EndIf;
   } EndFor;
   j-- ;
} EndWhile;                  // End Pass 1
```

```
// Pass 2 - HSP, WSP, CSP

Reset j;
Reset p

While (j != 0) {
 If (Sys_Object.itsStereotype = "hotSpare" || "warmSpare" ||" coldSpare")
 {
    For (p > 0; p--) {
      If (HSP || WSP || CSP Exists) {
         Connect BE to Gate Normal In;  }
      Else {
         Define Spare Gate = Sys_Object.itsStereotype;
         Connect BE to Spare Gate Normal In;  }
      EndIf;
      Connect Sys_Object.itsGeneralization to Pri (Spare Gate);
      Connect Spare Gate to Gate that BE is connected to;
      Delete Connection from BE to Upper Gate;
      InputOrder = Sys_Object.itsTaggedValue;
    } EndFor;
  } EndIf;
  j-- ;
} EndWhile;                  // End Pass 2
```

```
// Pass 3 - Reconfiguration

Reset j;
Reset p;

While (j != 0) {
  Locate Keyword "Fails";
  Define OR Gate;
  Define FDEP Gate;
  If (Keyword = Found) {
    For (p > 0; p--) {
      If (BE connected to Gate){
            Connect Gate Out to OR Gate In; }
      Else { Connect BE to OR Gate In;  }
      EndIf;
    } EndFor;
  } EndIf;
  Connect OR Gate Out to FDEP Trigger In;
  Locate Keyword "Replace" and "with";
  If (Keyword = Found) {
      For (p > 0; p--) {
        Connect FDEP Normal In to BE before "with"; }
       EndFor;
  } EndIf;
  j-- ;
} EndWhile;                  // End Pass 3
```

Figure 7. Algorithm for automatic DFT synthesis

In this case as well, our results were identical to the manually generated DFTs. The reason for this, again, is that our algorithm considers all failure criteria and therefore, our DFTs can always be reduced to simpler ones.

6. Summary and Conclusions

We have successfully conveyed, using our proposed UML modeling methodology, the number of components that exist in a fault tolerant computer-based system. We also captured the success (and failure) criteria, the components replaced from the spares when failures occur, the dependencies between hardware components and between the software and the hardware. We presented the modeling of reconfiguration in a system, and described how system complexity is managed. The models developed using our approach also embed statistical reliability information such as failure rates, coverage and failure distribution. The methodology is flexible enough to capture system hierarchies. It could be viewed that our approach is equivalent to specifying a dependability model in non-standard UML. However, our approach simply specifies the criteria needed for automating reliability analysis at design time itself. *e.g.* number of components present, number of spares, etc.

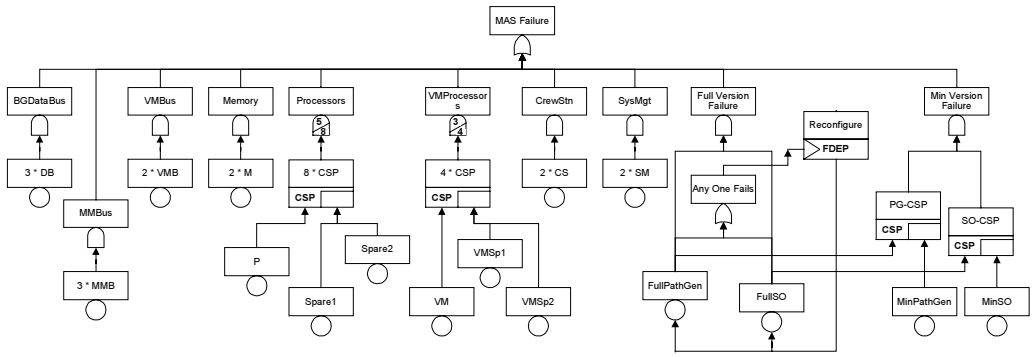

Figure 8. DFT for the reconfigurable MAS

Component	*λ*	*Coverage*	*Restoration*	*SPF*	*Q from [19]*	*Q (From Our Algorithm)*
All Processors	5E-04	0.999	0.0	1E-03		
Memory	1E-06	0.4999	0.5	1E-04	7.76449E-04	7.76449E-04
All Buses	1E-05	1.0	0.0	0.0		
Software	5E-06	0.9999	0.0	1E-04		

Table 2. List of Parameters and Summary of Results (MAS)

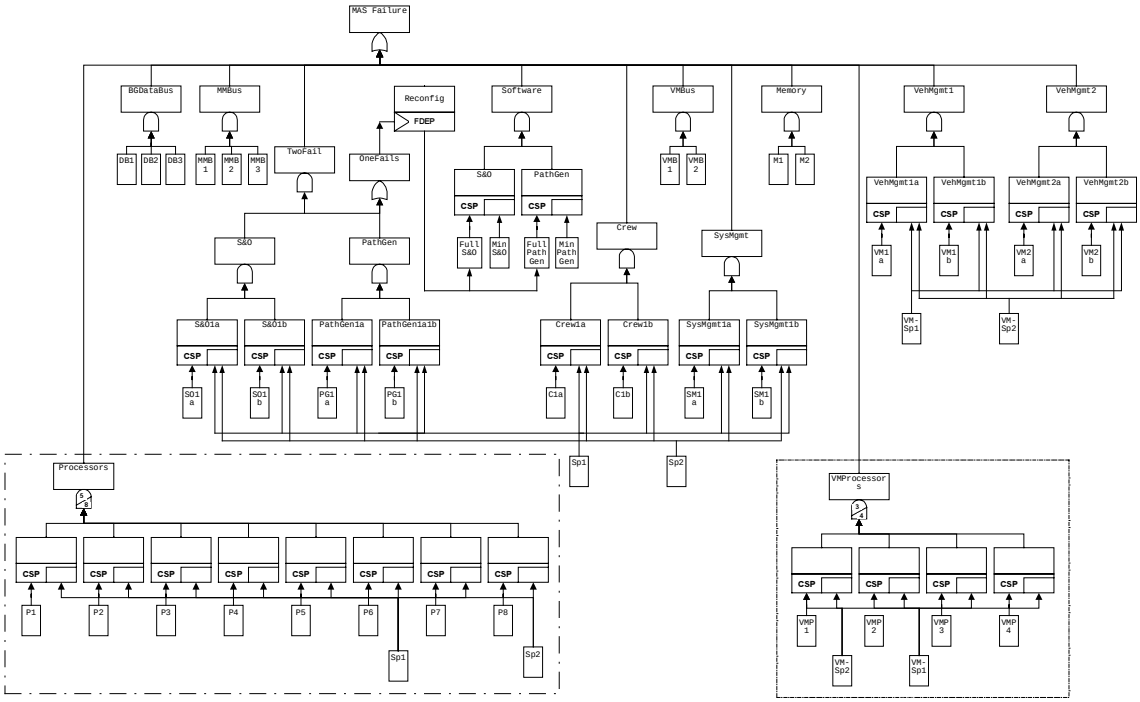

Figure 9. DFT generated using our algorithm for MAS configuration of [19]

Component	*λ*	*Coverage*	*Restoration*	*Q from [5]*	*Q (Our Algorithm)*	*Q (Our Algorithm)*
Network Element	3.0E-07	1.0	0.0			
A-Triad	1.5E-04	1.0	0.0		Non Hierarchical	Hierarchical
B-Triad	2.5E-04	1.0	0.0			
C-Triad	3.5E-04	1.0	0.0			
D- Triad	4.5E-04	1.0	0.0	1.20372E-05	1.20372E-05	1.20372E-05
Spare	5E-06	1.0	0.0			

Table 3. List of Parameters and Summary of Results (FTPP)

Since the designer knows this information, extending a traditional UML model permits automatic reliability analysis along with design. This extensibility of the UML is one of its major advantages. We use this property and extend the constructs' semantics to define stereotypes that indicate the types of dependencies, and generalizations that exist in the system. Class or object diagrams are used to capture the static system structure while deployment diagrams captures a runtime snapshot. Another advantage is that the UML itself is fairly standard, in that most designers understand its constructs.

We then presented an algorithm that converted the UML model into a dynamic fault tree. We have shown, using examples of some complex systems, that the models we develop are sufficient to automatically generate dynamic fault trees. We compared the results of the analytical solutions of the DFTs that we obtained with the results from manually generated DFTs. Our results indicate that the solution of our algorithm is usually identical to, or can be reduced to, the DFTs that are manually generated. The algorithm we presented generates the A*ND, OR, K/M, FDEP, CSP, WSP* and *HSP* gates. We believe that a significantly large variety of real world dependable systems can be analyzed using the same.

One disadvantage of the UML is that it is a semi-formal modeling language. Another limitation is that the UML design tool that the designer chooses could limit our modeling methodology. Although the constructs are precise, the semantics are not. A formally defined model, on the other hand, has several desirable advantages. Application of formal methods to precisely define the model removes any ambiguities about the model from the designer's mind. Therefore, one of the avenues of future work is to define a translation from UML to a formal model that can be checked by model checkers such as SPIN, and to formally define the DFT synthesis algorithm. Our algorithm for generating dynamic fault trees from system descriptions is independent of the internal UML representations; it is also independent of the tools. In fact, were the UML to change in its constructs (but not in their semantics) we will still be able to generate DFTs automatically using our approach. This is mainly because the initial stage of the automatic translation converts the system description to an internal logical representation of the system. It is this representation that is used to generate the DFT. Conceivably, if we construct the same logical system object from alternate modeling methodologies, specifically those that do not use the UML, we will still be able to automatically construct dynamic fault trees from the engineering model.

Acknowledgements

We thank the anonymous reviewers for the their helpful comments and suggestions on improving the paper. This work was partially supported by Lockheed Martin under grant EE-LM/NESS-0429-01 & NASA-LaRC under grant NAS1-99098.

References

[1] A. Rauzy, G. Point, "AltaRica: Langage de modélisation par automates à contraintes", *Modélisation des Sytèmes Réactifs,* Mar. 1999, pp. 81 – 90.

[2] G. Booch, J. Rumbaugh, I. Jacobson, *The Unified Modeling Language User Guide*, Addison Wesley, 1999.

[3] W.E. McUmber, B.C. Cheng, "A generic framework for formalizing UML", *Proc. of the IEEE International Conference on Software Engineering*, May 2001.

[4] W.E. McUmber, B.C. Cheng, "UML based analysis of embedded systems using a mapping to VHDL", *Proc. of 4th IEEE International Symposium on High Assurance Systems*, Nov. 1999

[5] J.B. Dugan, S.J. Bavuso, M.A. Boyd, "Fault trees and markov models for reliability analysis of fault tolerant systems", *Journal of Reliability Engineering and System Safety,* Vol. 39, 1993, pp. 291 – 307.

[6] J.B. Dugan, S.J. Bavuso, M.A. Boyd, "Dynamic fault-tree models for fault tolerant computer systems", *IEEE Trans. on Reliability,* Vol. 41, No. 3, Sept. 1992, pp. 363 – 373.

[7] R. Rao *et al.*, "Synthesis of reliability models from behavioral performance models", *Proc. of Ann. Rel. and Maint. Symposium*, 1994, pp. 292 – 297.

[8] D. Coppit, J.B. Dugan, K.J. Sullivan, "The Galileo fault tree analysis tool", *Proc. of IEEE FTCS,* June. 1999, pp. 232 – 235.

[9] D. Coppit, J.B. Dugan, K.J. Sullivan, "Developing a high-quality software tool for dynamic fault tree analysis", *Proc. of ISSRE-99* ., Nov. 999.

[10] S.A. Doyle, J.B. Dugan, "Dependability assessment using binary decision diagrams", *Proc. of the IEEE FTCS,* 1995.

[11] J.B. Dugan *et al.*, "DIFTree: A software package for the analysis of dynamic fault tree models", *Proc. of Rel. and Maint. Symposium*, Jan. 1997.

[12] D. S Hermann, *Software Safety and Reliability*: *Chapter 2*, IEEE Computer Society Press, 1999.

[13] A. Bondavalli *et al.*, "Automated dependability analysis of UML designs", *Proc. of 2nd IEEE Intl. Symp. on Object-oriented Real-time Distributed Computing*, 1999.

[14] M. Walter, "OpenSESAME: A Tool's Concept", *Proc. of the Satellite Workshops of the 27th Intl. Colloquium on Automata Languages, and Programming*, Nov. 2000

[15] K.K. Vemuri, J.B. Dugan, K. J. Sullivan, "Automatic synthesis of fault trees for computer-based systems", *IEEE Trans. on Rel.*, Vol. 48, No. 4, Dec. 1999, pp. 394 – 402.

[16] T.S. Assaf, *Automated reliability analysis of computer-based systems using UML*, MSEE Thesis, Dept. of ECE, Univ. of Virginia, Jan. 2001.

[17] G.J. Pai, *A UML framework for modeling and automated dependability analysis of computer-based systems*, MSEE Thesis, Dept. of ECE, Univ. of Virginia, Jan. 2001.

[18] N. Medvidovic *et al*., "Modeling software architectures in the Unified Modeling Language", *Tech. Rep. USC-CSE-2000-512, University of Southern California*, Aug. 2000

[19] J.B. Dugan, T.S. Assaf, "Dynamic fault tree analysis of a reconfigurable software system", *Proc. of the 19th Intl. System Safety Conference,* Sept. 2001, pp. 480 – 487.

[20] A Bondavalli *et al*., "Dependability analysis in the early phases of UML based dystem design", *Jrnl. of Computer Systems Science and Eng.*, Vol. 16, pp. 265-275, 2001

[21] H. Singh, B. Cukic *et al.*, "A bayesian approach to reliability prediction and assessment of component based systems", *Proc. of ISSRE-01*, Nov. 2001