Automatic Generation of Static Fault Trees from AADL Models

Anjali Joshi
University of Minnesota
Minneapolis, U.S.A.
ajoshi@cs.umn.edu

Steve Vestal
Honeywell Laboratories
Minneapolis, U.S.A.
steve.vestal@honeywell.com

Pam Binns
Honeywell Laboratories
Minneapolis, U.S.A.
pam.binns@honeywell.com

Abstract

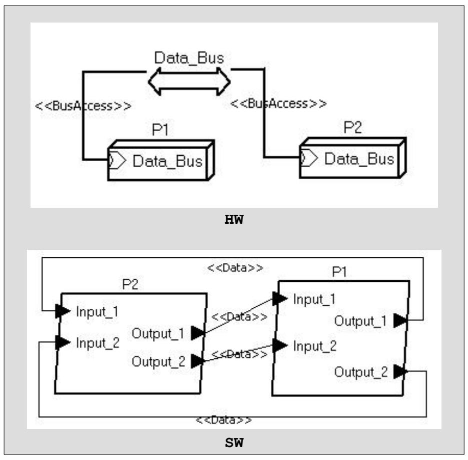
Safety-critical systems, such as avionics systems and medical devices, are developed with stringent safety requirements. System safety analysis provides assurance that the system in consideration satisfies these safety constraints. Traditionally, safety analysis is performed manually based on various informal requirements and design documents. Recent work in the area of model-based safety analysis,where safety analysis is based on a central formal model of the system, has helped demonstrate some key advantages of this methodology, including automatic generation of safety artifacts. Although most of this work is still far from being mature, we believe that this methodology holds promise in making the safety analysis process more formal, automated, consistent, and most importantly in helping tightly integrate the safety and systems engineering processes. We also believe that it is crucial to have a flexible modeling notation to capture both the system and the failure information to be able to derive "realistic" safety analysis. To corroborate our position, in this paper, we describe our prototype tool for automatically generating static fault trees based on architectural AADL models that can be input into a commercial fault tree analysis tool, CAFTA. We also put forth some challenges that we encountered that are potentially applicable to other approaches to automating generation of safety artifacts.

1. Introduction

Safety-critical systems are those systems where incorrect operation could lead to loss of life, substantial material or environmental damage, or large monetary losses; e.g., avionics systems, medical devices, and automobiles. System safety analysis provides assurance that the system in consideration satisfies certain safety constraints even in the presence of certain component failures. Safety engineers traditionally perform the safety analysis activities manually based on informal design models and various other documents such as requirements documents. This informal manual process makes these analyses subjective and dependent on the skill of the practitioner. Fault trees are one of the most common cause consequence models used by safety engineers; yet different safety engineers will often produce fault trees for the same system that differ in substantive ways. The final fault tree is typically produced only through a process of review and consensus building between the system and the safety engineers. Manually exposing the complex interactions between system components that affect safety is a non-trivial task and could result in missing information even in the final fault tree.

Model-based approaches to safety analysis have been proposed [5] [6] [2] [1] [3] [4] to address some of these issues by consolidating the information spread across various informal documents into a system model and deriving the safety artifacts automatically based on this system model. Some of the differences between the various approaches originate due to the type of the system model they consider as a basis for their analysis. The system model can be in the form of a high-level architectural model, a low-level system behavior model, or some combination of the two, based on various factors such as the current phase in the development cycle, granularity of the analysis, and so on. There exist tools for automatic fault tree generation, where the system model is encoded in a *behavioral* model; e.g., a NuSMV model for FSAP [2], a Simulink model using HiP-HOPS methodology for SAM [7], or an Altarica model [1].

Since safety analysis is performed in context of the entire system, it also needs to take into account the physical components of the system. One drawback of using the formal behavioral languages, from the safety analysis point of view, is that there is little inherent support for representing the system architecture in comparison to architecture description languages. The flexibility of the notation used for modeling the failure information is also crucial to deriving realistic safety analyses. The architecture description language, AADL (Architecture Analysis and Design Language) [9], an SAE standard, has inherent support for describing and binding various system components through the core language. The AADL standard also provides a Error Model Annex [10] sub-language that supports specification of fault and failure information. One of the advan-



```
system Duplex
end Duplex;

system implementation Duplex.Basic
  subcomponents
   P1: processor Proc.Basic;
   P2: processor Proc.Basic;
   Data_Bus: bus PCI.Basic;
  connections
   bus access Data_Bus -> P1.Data_Bus;
   bus access Data_Bus -> P2.Data_Bus;
end Duplex.Basic;
```

```
system Dual
end Dual;

system implementation Dual.Basic
  subcomponents
   P1: process SW_Proc.Basic;
   P2: process SW_Proc.Basic;
  connections
   C12: data port P1.Output_1 -> P2.Input_1;
   C21: data port P2.Output_1 -> P1.Input_1;
   C12_2: data port P1.Output_2 ->P2.Input_2;
   C21_2: data port P2.Output_2 -> P1.Input_2;
end Dual.Basic;
```

```
                DualSystem.Basic
system DualSystem
end DualSystem;

system implementation DualSystem.Basic
  subcomponents
    HW: system Duplex.Basic;
    SW: system Dual.Basic;
  properties
    Actual_Processor_Binding =>  reference HW.P1 applies to SW.P1;
    Actual_Processor_Binding =>  reference HW.P2 applies to SW.P2;
    Actual_Connection_Binding =>  reference HW.Data_Bus applies to SW.C12;
    Actual_Connection_Binding =>  reference HW.Data_Bus applies to SW.C21;
    Actual_Connection_Binding =>  reference HW.Data_Bus applies to SW.C12_2;
    Actual_Connection_Binding =>  reference HW.Data_Bus applies to SW.C21_2;
end DualSystem.Basic;
```

Figure 1. Snippet of a Simple Dually Redundant System in AADL

tages of the Error Annex is that it enables specification of error annotations on the original AADL architecture model, hence enabling the safety analysis to consider the component error models and their interactions in context of the system architecture.

In our current exercise, we use AADL as the notation for capturing the system architecture model and use the Error Model annex to capture the component faults and failure modes. We implemented a tool that automatically generates static fault trees that can be input into the commercial fault tree analysis tool, CAFTA [11]. For deriving other types of dependability artifacts from AADL and the Error Model Annex, the interested reader is referred to the work on automatically deriving GSPN (Generalized Stochastic Petri Net) [8].

The rest of the paper is organized as follows. The next Section gives a background in modeling using AADL and the Error Model Annex with the help of a small example. The following Section discusses the high-level approach to automatically generating static fault trees based on the AADL models. We conclude the paper with a discussion of some of the advantages and challenges of this approach.

2. Background

In this Section, we give a brief overview of the AADL standard [9] and then introduce a small running example to illustrate system and error modeling using AADL and the Error Model Annex.

2.1. AADL Overview

Components form the central modeling construct in AADL[1]. Components are defined through the *type* and *implementation* declarations. A component type declaration defines a component's interface elements and externally observable attributes (e.g., `features` that are interaction points with other components, `properties` that define intrinsic characteristics of a component). A component implementation declaration defines a component's internal structure in terms of `subcomponents`, `connections` among the features of those subcomponents, `properties`, etc. There are three distinct sets of component categories: 1. application software (e.g., `thread`, `process`, `data`), 2. execution platform (e.g., `processor`, `memory`, `bus`), 3. composite (`system`, which is a composite of software, execution platform, or system components). The AADL standard includes predefined binding properties to specify mappings between the relevant execution hardware components and the software components and connections. The AADL *System Instance Model* is generated from the declarative model by specifying a system implementation as the root of the system instance and recursively instantiating the subcom-

[1]Due to lack of space, we will only explain the relevant AADL and Error Model Annex constructs. Please refer to the standard for details.

```
package Standard_Errors
public annex error_model
{**
error model Basic
 features
  err_free: initial error state;
  loss_avail, loss_int: error state;
  fail_stop, fail_babble: error event;
  loss_data, corrupt_data: in out error propagation;
end Basic;

error model implementation Basic.Hardware
 transitions
  err_free -[fail_stop, in loss_data]-> loss_avail;
  err_free -[fail_babble, in corrupt_data]-> loss_int;
  loss_avail -[fail_babble]-> loss_int;
  loss_avail -[in loss_data, out loss_data]-> loss_avail;
  loss_int -[in corrupt_data, out corrupt_data]-> loss_int;
end Basic.Hardware;

error model implementation Basic.Software
 transitions
  err_free -[in loss_data]-> loss_avail;
  err_free -[in corrupt_data]-> loss_int;
  loss_avail -[fail_babble]-> loss_int;
  loss_avail -[in loss_data, out loss_data]-> loss_avail;
  loss_int -[in corrupt_data, out corrupt_data]-> loss_int;
end Basic.Software;
**};
end Standard_Errors;
```

Figure 2. Error Model Specification using the Error Model Annex

ponents. The system instance is bound by identifying all the actual binding properties defined in the components. The Software Engineering Institute (SEI) has developed an open source AADL Tool Environment, OSATE, as a set of plug-ins on top of the open source Eclipse platform. OSATE provides a toolset for front-end processing of AADL models, such as parsing and semantic checking for textual AADL. OSATE also has support to generate a bound System Instance Model.

Example: Simple Dually Redundant System in AADL Consider a trivial dually redundant system (Figure 1) composed of two software processes and two processors connected via a data bus. The highest-level `DualSystem` system component is composed of two subcomponents - `HW` and `SW`, instances of the system implementations `Duplex.Basic` and `Dual.Basic`, respectively. `HW` and `SW` subcomponents are in turn composite components, as shown in Figure 1. The `DualSystem` implementation also includes properties that bind the two software processes to the two processors and the connections between the two software processes to the hardware bus. We can then create the system instance by instantiating the highest-level component implementation, `DualSystem.Basic`.

2.2. Error Model Annex Overview

AADL is designed to be extensible (using Properties and Annex libraries) to accommodate analyses of the runtime architectures that the core language does not completely support. The Error Model Annex [10] of the AADL standard provides a mechanism to allocate error annotations to AADL components and connections. The Annex sublanguage allows the specification of individual error models on components, interaction between error models through error propagations, rules determining error propagations between various types of components, filtering and masking error propagations, and hierarchical composition of subcomponent error models.

Example: Error Model Specification We now define a generic error model type and two slightly different implementations for the software and the hardware components in our example architecture model as shown in Figure 2[2]. The error model type `Basic` defines error states (representing failure modes), an initial error state, error events (representing intrinsic faults), and input/output error propagations. Error events are not only used to model component fault events; they can also model repair events (we did not include repair events in our current prototype). The error implementation defines error model transitions that change the error state of the component based on the error events and propagations. Occurrence properties can be defined for the error events and out propagations that specify their occurrence rates (fixed, poisson, or a user-defined rate).

Example: Error Model Association Once we have the error models defined, we can associate them to the relevant components and connections in the system model via the `annex` subclause defined as an extension point in the core AADL language, as shown in Figure 3. The `Model` property associates the given component to a particular error model type or implementation. Error models associated with components can either be derived from their subcomponent error models (`Model_Hierarchy` property value `Derived` and `Derived_State_Mapping` property maps the subcomponent error states to the component error states), or they can be abstractions of the subcomponent error models, in which case the subcomponent error models will be ignored (`Model_Hierarchy` property value `Abstract`, which is the default value). In the case of `Derived` hierarchy, the use of the `Model` property is only to identify the component error states defined in the associated error model that get used in the mapping. For components with `Abstract` error models, we can also associate `Guard_In` and `Guard_Out` properties that provide guard conditions for the `in` and `out` error propagations, respectively. The `Guard_In` property enables the component defining it to mask (i.e., ignore) or translate the incoming error propagations at the receiving interface. In our example AADL model, we associate the `HW` subcomponents (`P1`, `P2`, `Data_Bus`) with the `Basic.Hardware` implementation, and the all the software components (`SW`, `P1`, `P2`) with the `Basic.Software` implementation, with `SW` defined with a `Derived` error model.

[2]loss_of_availability, loss_of_integrity, loss_of_data, corrupted_data, error_free shortened to loss_avail, loss_int, loss_data, corrupt_data, err_free respectively

```
system implementation Dual.Basic
...
annex error_model {**
 model => Standard_Errors::Basic.Software;
 model_hierarchy => Derived;
 derived_state_mapping =>
   err_free when P1 or P2,
   loss_avail when P1[loss_avail] or P2[loss_avail],
   loss_int when others;
 report => loss_avail, loss_int;
**};
end Dual.Basic;

process implementation SW_Proc.Basic
annex error_model {**
 model => Standard_Errors::Basic.Software;
 occurrence => fixed 1E-4 applies to error fail_stop;
 guard_in =>
   mask when Input_1[err_free] or Input_2[err_free],
   corrupt_data when Input_1[loss_int] and
                     Input_2[loss_int],
   loss_data when others
 applies to Input_1, Input_2;
**};
end SW_Proc.Basic;
```

Figure 3. Error Model Associations

3. Static Fault Tree Generation

In this Section, we will describe our approach of automatically extracting static fault trees from the AADL model annotated with the relevant error models. This tool is a plug-in into the existing OSATE framework. The fault tree generation tool is designed to be flexible and can be re-targeted to more than one fault tree analysis tool. The portion of the tool that extracts the System Instance Error Model can be reused to generate different types of safety artifacts, such as Markov Chains. To analyze our generated fault trees we used CAFTA, a commercial fault tree tool that was available and widely-used within Honeywell.

3.1. Our High-level Approach

We divide our fault tree generation approach into three high-level steps: extracting a system instance error model, generating an intermediate fault tree, and formatting the intermediate fault tree for a specific analysis tool.

1. System Instance Error Model Extraction
The System Instance Error Model consists of a set of error model instances for component and connection instances and client/server bindings within that system. Though OSATE provides support to generate a bound system instance Model, it does not create the system instance error model. Hence, we need to extract the system instance error model based on the system instance model and the individual error models referenced in the system instance model. There are two types of instances in the system instance model - Component instances and Connection instances (semantic connections). The component and connection declarations can be directly associated with an error model (via the `Model` property). These can be directly retrieved for the instances. If there is no directly associated error model for a connection instance, then the error model of the ultimate source applies to that connection instance. For component instances, in addition to the `Model` property, we also need to retrieve `Model_Hierarchy`, `Derived_State_Mapping`, `Occurrence`, and `Guard_In` property values. For connection instances, only `Occurrence` properties other that the `Model` property apply. We also need to identify the error propagation sources for the component and connection instances. For simplicity, we distinguish component instance error propagation sources as: *Direct propagations* - These are the error propagations that occur through port connections, either due to the error model on the connection, or the error model of the connected component instance, and *Indirect propagations* - These are the error propagations that occur from other component instances through access connections or due to binding properties.

We store all this information in the form of nodes of a Directed Graph (DG). For component instances with derived error models, the node points to all the hierarchically contained subcomponents. For component instances with abstract error models and connection instances (which only have abstract models), the node points to all the components that could be sources of their input error propagations. The underlying error propagations paths can lead to potential cycles in the DG that need to be broken while generating a fault tree (details in the Discussion Section).

In our illustrative example, the `SW` subcomponent `P1` is associated to an `Abstract` error model `Basic.Software`, with `Guard_In` properties applied to its two input ports. The error propagations can occur directly through the input ports, `Input_1` and `Input_2`, and also indirectly from the processor that it is bound to, `HW.P1`. Note here that the `Guard_In` property does not apply to indirect error propagations. As another example, the `HW` subcomponent `Data_Bus` is associated to the `Abstract` error model `Basic.Hardware`. Based on the error propagation rule in the Error Annex that a processor can propagate errors to the bus that it is connected to, both processors `HW.P1` and `HW.P2` can propagate errors to `HW.Data_Bus`.

2. Intermediate Fault Tree Generation
Once we have the system instance error model information stored in the form of a DG, we now go on to the fault tree generation phase. The fault tree generation algorithm is a recursive algorithm, with the top level event being the error state or propagation listed in a `Report` property (these are the declared system hazards to be analyzed). Based on our example (Figure 3), the `Report` property lists two error states, `loss_of_availability`, `loss_of_integrity`, which will be considered as top-level events for generating the fault trees.

We then apply a set of optimizations to the intermediate

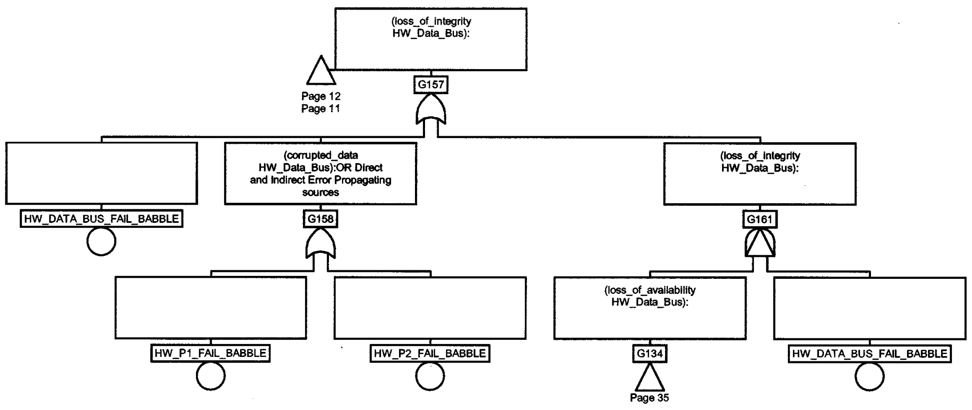

(loss_of_integrity HW_Data_Bus):
Page 12
Page 11
G157
(corrupted_data HW_Data_Bus):OR Direct and Indirect Error Propagating sources
(loss_of_integrity HW_Data_Bus):
HW_DATA_BUS_FAIL_BABBLE
G158
G161
(loss_of_availability HW_Data_Bus):
HW_P1_FAIL_BABBLE
HW_P2_FAIL_BABBLE
G134
HW_DATA_BUS_FAIL_BABBLE
Page 35

Figure 4. Snapshot of the Auto-generated CAFTA Subtree

representation that perform syntactic optimizations, such as, remove redundant operators, collapse gates, etc. More complex optimizations are also implemented such as, sharing of subtrees within the same fault tree and also between separate fault trees. This sharing may or may not be preserved in the output fault tree depending on the support in the target analysis tool. Pruning redundant subtrees is an involved optimization that we currently do not perform.

Optimization of the generated fault trees turned out to be a critical and non-trivial aspect of our work. We will discuss in detail some of the challenges associated with this optimization in the Discussion Section.

3. CAFTA Fault Tree Generation
This will parse the intermediate representation and output CAFTA fault trees and the corresponding basic events and gates database files. CAFTA allows multiple top level events, which lets us output fault trees with different top level gates, but which can share subtrees.

Example Auto-generated CAFTA Fault Tree As an example, consider a snapshot of a subtree for the error state `loss_of_integrity` for `HW.Data_Bus`, a part of the generated output CAFTA fault tree, as shown in Figures 4. Based on the error transitions defined in the `Basic.Hardware` error model implementation (Figure 2), this error state can be reached in the following three ways: (1) The component is in the `error_free` state and a error event `fail_Babble` occurs, (2) The component is in the `error_free` state and an `in` error propagation `corrupted_Data` occurs, and (3) The component is in the `loss_of_availability` state and a error event `fail_Babble` occurs. Figure 4 shows an `OR` (`G157`) gate, with three inputs corresponding to the above cases (the intrinsic error event is specific to the component and a fully instantiated name is given to the event `HW_Data_Bus_fail_babble`). The `priority-AND` (`G161`) gate graphically captures the sequence defined in the error transition. This gate is considered by CAFTA as a regular `AND` gate for quantitative analysis. As we discussed earlier, there is an error propagation path from the two processors `HW.P1` and `HW.P2` to the bus. Consider error propagation from `HW.P1` - based on the error transitions, we identify that it emits the `out` error propagation `corrupted_data` when it is in error state `loss_of_integrity`, which can be reached either with the intrinsic error event `fail_babble`, or an `in` error propagation `corrupted_data`. Note that `HW.P1` can be propagated error from the bus itself, considering which causes a cycle in the fault tree and we break it by ignoring this particular error propagation. Thus `corrupted_data` can be propagated only when the intrinsic error events, `fail_babble`, occur in either of the two processors (as shown with an `OR` (`G158`) gate in Figure 4). Note that this subtree is shared (the small triangle shows the page numbers where this subtree is referenced).

4. Discussion

Even with a really simple system architecture, we can start annotating it with fairly complex and realistic error models representing the various faults and failure modes. This is important, as one can start performing safety analysis from the time that the system is just being conceptualized and see the safety implications of the design choices made early on. This is one of the key advantages of our approach, and of model-based safety analysis in general. The resulting fault trees are hierarchical (one can trace through the system architecture hierarchy by tracing through the fault tree), consistent (different fault trees that refer to the same intrinsic error event, will always refer to the same basic event name in the fault tree database), detect common-mode failures (shared subtrees), and manually readable (the gate information will help the reviewer relate to the system and error models). Different fault trees can share subtrees that can help the reviewer or the analysis tool identify what basic events contribute to multiple top-level hazards. The rerun of the analysis is easy. By developing the prototype tool targeting CAFTA, we illustrated how one can generate common safety artifacts, such as fault trees that can be then fed into commercial analysis tools that safety engineers are comfortable with.

With the obvious benefits that this approach has to offer, there are still quite a few challenges that need to be addressed to make this approach practically feasible. One of the potential risks is in the safety artifacts completely missing out failure modes that cannot be captured in the error model. Due to the constraints imposed by AADL and the Error Model Annex languages, certain types of failure modes cannot be specified in the error model and should be considered separately.

Aggressive optimizations are crucial to automatically generating "manageable" fault trees. Here, we discuss one of the challenges that we encountered in performing optimizations for sharing and pruning of sub-trees. To illustrate the problem with an example, consider the `SW` component implemented by `Dual.Basic` (Figure 1). Note that `P1` and `P2` depend on each other for their inputs leading to a cyclic dependency in the system architecture. `SW` has an associated derived error model that refers to subcomponents `P1` and `P2` error states for computing its own error state (refer to `Derived_State_Mapping` property in Figure 3). The fault tree for the `SW` component in certain error state will have a structure as shown in Figure 5. The subtree for `P1` in the left branch is dependent on error propagations from `P2`, which in turn is dependent on error propagations from `P1`, leading to a cycle in the fault tree (this error propagation cycle exists in the intermediate representation DG). Currently, we break this cycle by simply keeping track of all the components that we have already visited (call this list *Ancestors*). In this case, when we refer to `P1` as the source to our error propagation, our *Ancestors* list contains (`SW`, `P1`, `P2`), and we can immediately detect a cycle and ignore the error propagation occurring from `P1`. Note that, we also share subtrees if we detect a component that has a subtree generated for a particular error state or propagation. In this case however, when we encounter `P2` on the right branch, we cannot share the same subtree as the one created for `P2` on the left branch, though it refers to the same error state or propagation. This is due to the fact that the cycle that was broken off for `P2` on the left branch is not yet a cycle on the right side (the ancestor list (`SW`, `P2` does not contain `P1` yet). Thus, we need to regenerate a new fault tree for `P2` on the right hand side. Currently, we only share subtrees if we find that the *Ancestor* list is identical at the two locations referring to the particular component.

More aggressive optimizations for pruning and handling cycles can be applied to sub-trees that are *monotonic* (subtrees for which both positive and negative representations do not appear as basic events). However, we must introduce negation operators for some constructs in the language because not all fault conditions can be represented by monotonic fault trees. Aggressively pruning non-monotonic fault trees is non-trivial and needs to be investigated further. Our approach to handling cyclic error propagations is rather simplistic and needs to be rigorously evaluated.

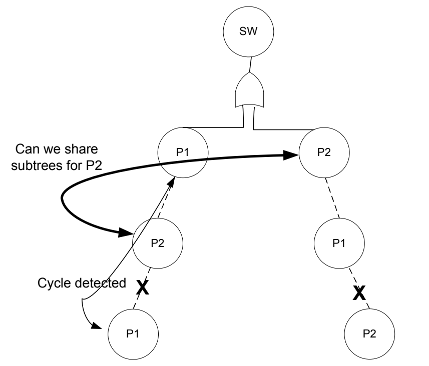


Figure 5. Cycles and Sharing Subtrees

In conclusion, this paper discussed our approach to automatically generating static fault trees for a commercial tool based on AADL models and the advantages and challenges of automatically generating such safety artifacts.

References

[1] P. Bieber, C. Castel, and C. Seguin. Combination of Fault Tree Analysis and Model Checking for Safety Assessment of Complex System. In *Proceedings of the 4th European Dependable Computing Conference*, pages 19 – 31. Springer-Verlag, 2002.

[2] M. Bozzano and A. Villafiorita. Improving System Reliability via Model Checking: the FSAP / NuSMV-SA Safety Analysis Platform. In *SAFECOMP 2003*, pages 49–62, Edinburgh, 2003. Springer.

[3] H. Giese, M. Tichy, and D. Schilling. Compositional Hazard Analysis of UML Component and Deployment Models. In *Computer Safety, Reliability, and Security*, volume 3219 of *LNCS*, pages 166–179. Springer, 2004.

[4] L. Grunske and B. Kaiser. Automatic Generation of Analyzable Failure Propagation Models from Component-level Failure Annotations. *QSIC*, pages 117–123, 2005.

[5] A. Joshi and M. P. Heimdahl. Model-Based Safety Analysis of Simulink Models Using SCADE Design Verifier. In *SAFECOMP*, volume 3688 of *LNCS*, pages 122–135. Springer-Verlag, Sept 2005.

[6] A. Joshi, S. Miller, M. Whalen, and M. P. Heimdahl. A Proposal for Model-Based Safety Analysis. In *Proceedings of 24th DASC*, Nov 2005.

[7] Y. Papadopoulos and M. Maruhn. Model-based Synthesis of Fault Trees from Matlab-Simulink Models. In *(DSN'01)*, July 01 - 04 2001.

[8] A.-E. Rugina, K. Kanoun, and M. Kaâniche. A System Dependability Modeling Framework using AADL and GSPNs. Technical Report 05666, LAAS-CNRS, Nov 2006.

[9] SAE-AS5506. *Architecture Analysis and Design Language*. SAE, Nov 2004.

[10] SAE-AS5506/1. *Architecture Analysis and Design Language Annex Volume 1*. SAE, June 2006.

[11] SAIC. Computer aided fault tree analysis (cafta)product brochure.