

Automatic Generation of Fault Trees from AADL Models

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

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

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

Abstract

¹*Safety-critical systems, such as avionics systems and medical devices, are developed with stringent safety requirements. System safety analysis provides assurance that the system satisfies these safety constraints. Traditionally, safety analyses are performed manually based on various informal requirements and design documents. Much recent work has investigated automating system safety analyses using formal error models linked to system specifications. This integrated 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. This paper extends our soon to appear publication [10] which describes 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. This paper adds more related work and further discusses the couplings among model level semantics and expressiveness, characteristics of an intermediate representation, and underlying mathematical solution techniques that produce the analysis results.*

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 [1] [3] [5] [6] [7] [8] [9] [11] 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. Examples of models with automatic fault tree generation tools, where the system model is encoded in a *behavioral* model are, a NuSMV model for FSAP [3], a Simulink model using HiP-HOPS methodology for SAM [12], and an Altarica model [1].

Since safety analysis is performed in the 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 important to deriving realistic safety analyses. The architecture description language, AADL (Architecture Analysis and Design Language) [15], an SAE standard, has inherent support for describing and binding various system components through the core language. The AADL standard also provides an Error Model Annex [16] sub-language that supports specification of fault and failure information. One of the advan-

¹This paper expands “Automatic Generation of Static Fault Trees from AADL Models,” which is to appear in DSN07-WADS ([10]).

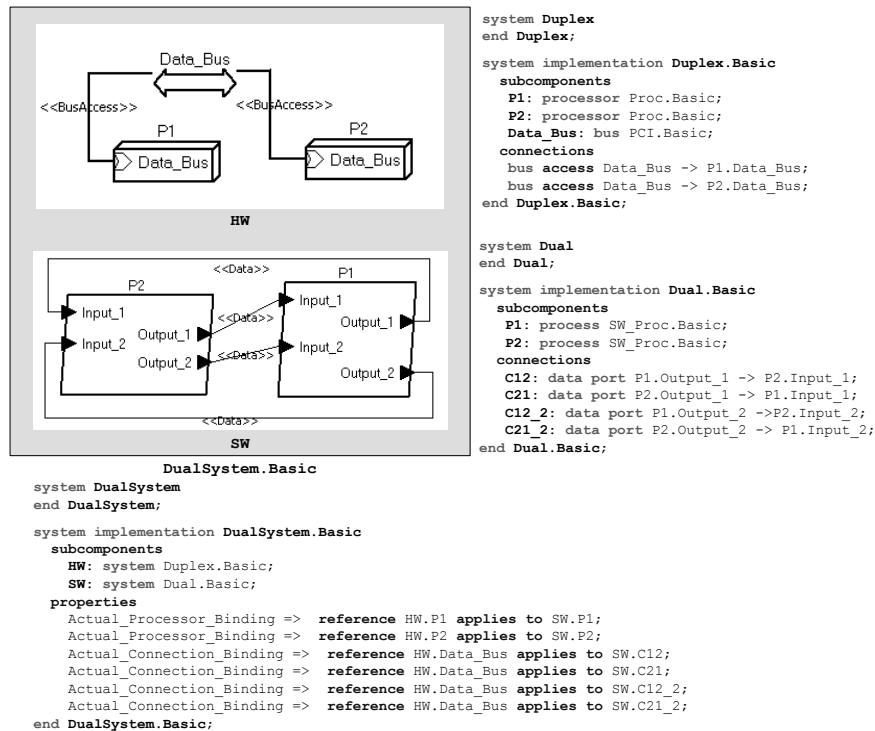


Figure 1. Snippet of a Simple Dually Redundant System in AADL

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 [17]. When solving the fault trees, CAFTA finds the minimal cut sets. Several recent fault tree analysis tools are using Binary Decision Diagrams (BDDs) to represent and then solve the fault trees. In our Discussion and Future Directions section, we briefly highlight some of the links and trade-offs of different internal representations and solution techniques.

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) [14]. A discussion on the automatic generation and analysis of different types of complementary models, including reliability models, from a common AADL specification is described in [2].

The rest of the paper is organized as follows. The next section provides background in modeling using AADL and the Error Model Annex with the help of a small example. The subsequent section sketches our 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, challenges and candidate future directions of this approach in relation to other on-going work.

2. Background

This section gives a brief overview of the AADL standard [15] and then introduces a small running example to illustrate system and error modeling using AADL and the Error Model Annex [16].

2.1. AADL Overview

Components form the central modeling construct in AADL². 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

²Due to lack of space, we only explain the relevant AADL and Error Model Annex constructs. Please refer to the standards [15, 16] for details.

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 subcomponents. 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 [16] of the AADL standard provides a mechanism to ascribe error annotations to AADL components and connections. The Annex sub-language 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 sub-component 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

```
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

Figure 2³. 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 er-

³The terms `loss_of_availability`, `loss_of_integrity`, `loss_of_data`, `corrupted_data`, `error_free` have been 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

ror 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.

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 plugin 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

Our fault tree generation approach involves 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. In the discussion concluding this paper, we show how even some simple error modeling constructs lead to fault “trees” with loops, and we briefly compare our treatment to a more formal treatment for BDDs that we just recently discovered [13].

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 than 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).

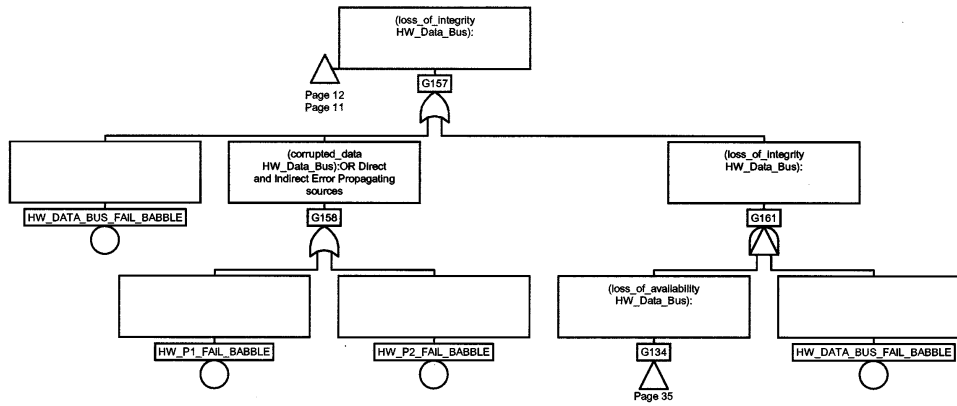


Figure 4. Snapshot of the Auto-generated CAFTA Subtree

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 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 some of the challenges associated with this

optimization in the Discussion and Future Directions 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 an 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 an 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 and Future Directions

Even a simple system architecture can be annotated early on with fairly complex and realistic error models representing the various faults and failure modes. This makes it easy to perform safety analyses from the time the system is being conceptualized and then repeatedly evaluate the safety implications of the design choices as the system evolves. The resulting fault trees are consistent with the architecture design by construction, eliminating a potential source of inconsistency in current manual processes. The fault trees map clearly and intuitively back to the design, which is helpful both during design studies and evolution and during certification reviews. The entire process is automated and reasonably fast, making it possible to use the analysis results during design trade-off studies and resulting in an improved design. A single fault forest is generated for the entire system, reducing the possibility of overlooking common mode faults and enabling improved multi-function, multi-hazard safety analyses.

Our prototype tool targeted CAFTA, a widely-used tool in industry. With proper care in the fault tree generation algorithm, such as careful selection of optimizing transformations and tree structuring and naming conventions, the resulting trees appear similar to ones that might be produced by a manual processes (another study that generated fault trees from annotated UML specifications also found the resultant fault trees were either identical to or similar to fault trees that would have been generated by hand [11]).

Challenges remain before this approach is practically feasible. Despite the intuitive modeling appeal that fault trees offer, it is well known that solutions for large fault trees, where it is not possible to generate all minimal cut sets, are both computationally inefficient and inaccurate. This remains true even when optimizations such as common subtree elimination are applied.

Another challenge we encountered in optimization was the pruning of recursive sub-trees, or equivalently the breaking of loops that resulted from cyclic dependencies in the specifications. Our example illustrates a cyclic dependency. 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

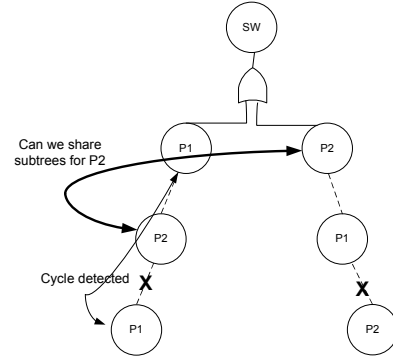


Figure 5. Cycles and Sharing Subtrees

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 a 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*). 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. We also share subtrees if we detect a component that has a subtree generated for a particular error state or propagation. 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, even 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). We consequently 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.

Our prototyped cycle breaking algorithm applies only to monotone functions. For a monotonically increasing function, a working component (*vs.* a failed component) will never decrease overall system performance. Fault trees without negation operators (including NOT, NOR and NAND) are monotonic. Coherent systems are monotonic systems in which all components are relevant. A relevant component's failure will degrade overall system performance. The restrictive class of coherent models precludes many natural system constructs such as voting (*i.e.* 2 or more of 3). Cycle breaking extensions are needed for cyclic non-coherent fault trees.

In another investigation of transforming a system level fault specification to an underlying Boolean expression for

safety/risk analysis [13], looped and hierarchical constructs are addressed distinctly, where a loop is determined by a node having itself as an ancestor, just as in our algorithm. That paper presents a rigorous and comprehensive approach to address looped systems. Algorithms to construct the prime implicants⁴ for looped systems are presented in a Binary Decision Diagram (BDD) framework. It is shown that for purely monotone systems, loops can be handled using only logical constructs. For non-monotone systems, a partial order operator was introduced to eliminate any non-determinism that results in the logical expression of the physical model it is representing. The partial order is a formal way of expressing an implied behavioral preference of the physical system, which the systems engineer might know from experience to hold. Thus, the specific partial order given might not apply to all specifications, it was chosen to produce a desired result for looped bridge structure and several other example networks reported in [13].

As an alternative to generating fault trees directly, one might consider generating BDDs directly, which can then be solved using several of the more recent commercial fault tree analysis tools. BDDs are a much less intuitive representation than traditional fault trees, although their solution times can be much faster and the numeric accuracy of the solution much greater [4, 11, 13]. This raises the very important question of how accurately any underlying target set of mathematical models can practically capture and provide a quantitative safety assessment for an architecture specification (with associated error annotations). Automatic generation of large, complex models may be the only viable approach. However, it can only be effective if the tool developers have an in depth understanding of the system level semantics and how they map to the underlying mathematical models, and the tools then provide a careful and comprehensive treatment of the mappings.

In summary, 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. Note that we have only focused on static fault tree analysis. As the system evolves, more complicated fault tree analyses or analyses of different system aspects may be required. An integrated dynamic fault tree toolset might be a back-end tool for a safety analysis. One such example is DIFTree [4], which mixes several different solution techniques to enable analysis of more complex fault models with varied fault constructs. The AADL and envisioned toolset was designed to integrate with a variety of existing analysis tools.

⁴Prime implicants are frequently the same as minimal cut sets in many practical applications.

References

- [1] P. Bieber, C. Castel, and C. Seguin. Combination of Fault Tree Analysis and Model Checking for Safety Assessment of Complex System. In *4th European Dependable Computing Conference*, pages 19 – 31. Springer-Verlag, 2002.
- [2] P. Binns and S. Vestal. Hierarchical Composition and Abstraction in Architecture Models. In *Architecture Description Languages: IFIP Springer Science + Business Media Inc.*, Volume 176, Chapter 3, Editors: P. Dissaux, M. Filali-Amine, P. Michel, and F. Vernadat, 2005
- [3] M. Bozzano and A. Villaflorita. Improving System Reliability via Model Checking: the FSAP / NuSMV-SA Safety Analysis Platform. In *SAFECOMP 2003*, pages 49–62, Edinburgh, 2003. Springer.
- [4] J.B. Dugan, K.J. Sullivan, and D. Coppit. Developing a Low-Cost, High-Quality Software Tool for Dynamic Fault Tree Analysis. In *IEEE Proceedings of the International Symposium on Software Reliability Engineering*, 1999
- [5] 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.
- [6] L. Grunske and B. Kaiser. Automatic Generation of Analyzable Failure Propagation Models from Component-level Failure Annotations. *QSIC*, pages 117–123, 2005.
- [7] L. Grunske and B. Kaiser. An Automated Dependability Analysis Method for COTS-Based Systems. In *ICCBSS, LNCS 3412*, Eds. X. Franch and D. Port, pp. 178-190, Springer-Verlag, 2005
- [8] 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.
- [9] A. Joshi, S. Miller, M. Whalen, and M. P. Heimdahl. A Proposal for Model-Based Safety Analysis. In *Proceedings of 24th DASC*, Nov 2005.
- [10] A. Joshi, S. Vestal, and P. Binns. Automatic Generation of Static Fault Trees from AADL Models In *Proceedings of the IEEE/IFIP Conference on Dependable Systems and Networks' Workshop on Dependable Systems*, DSN07-WADS, Edinburgh, Scotland-UK, June 2007
- [11] G. Pai and J.B. Dugan. Automatic Synthesis of Dynamic Fault Trees from UML System Models In *IEEE Proceedings of the 13th International Symposium on Software Reliability Engineering*, 2002
- [12] Y. Papadopoulos and M. Maruhn. Model-based Synthesis of Fault Trees from Matlab-Simulink Models. In (*DSN'01*), July 01 - 04 2001.
- [13] A. Rauzy. A New Methodology to Handle Boolean Models with Loops. In *IEEE Transactions on Reliability*, Vol. 52, No. 1, March 2003
- [14] A.-E. Rugina, K. Kanoun, and M. Kaâniche. A System Dependability Modeling Framework using AADL and GSPNs. Technical Report 05666, LAAS-CNRS, Nov 2006.
- [15] SAE-AS5506. *Architecture Analysis and Design Language*. SAE, Nov 2004.
- [16] SAE-AS5506/1. *Architecture Analysis and Design Language Annex Volume 1*. SAE, June 2006.
- [17] SAIC. Computer aided fault tree analysis (cafta)product brochure.