Automated Fault Tree Analysis from AADL Models

Peter Feiler and Julien Delange
Carnegie Mellon Software Engineering Institute
4500 5th Avenue, Pittsburgh, PA15213
{phf,delange}@sei.cmu.edu

Abstract

Cyber-physical systems, used in domains such as avionics or medical devices, perform critical functions where a fault might have catastrophic consequences (mission failure, severe injuries, etc.). Their development is guided by rigorous practice standards that prescribe safety analysis methods in order to verify that failure have been correctly evaluated and/or mitigated. This labor-intensive practice typically focuses system safety analysis on system engineering activities.
As reliance on software for system operation grows, embedded software systems have become a major source of hazard contributors. Studies show that late discovery of errors in embedded software system have resulted in costly rework, making up as much as 50% of the total software system cost. Automation of the safety analysis process is key to extending safety analysis to the software system and to accommodate system evolution.
In this paper we discuss three elements that are key to safety analysis automation in the context of fault tree analysis (FTA). First, generation of fault trees from annotated architecture models consistently reflects architecture changes in safety analysis results. Second, use of a taxonomy of failure effects ensures coverage of potential hazard contributors is achieved. Third, common cause failures are identified based on architecture information and reflected appropriately in probabilistic fault tree analysis.
The approach utilizes the SAE Architecture Analysis & Design Language (AADL) standard and the recently published revised Error Model Annex V2 (EMV2) standard to represent annotated architecture models of systems and embedded software systems. The approach takes into account error sources specified with an EMV2 error propagation type taxonomy and occurrence probabilities as well as direct and indirect propagation paths between system components identified in the architecture model to generate a fault graph and apply transformations into a fault tree representation to support common mode analysis, cut set determination and probabilistic analysis.

Categories and Subject Descriptors
Software and its engineering→Software safety
Software and its engineering→System modeling languages
Software and its engineering→Fault tree analysis
Software and its engineering→Software verification and validation

General Terms Algorithms, Reliability, Standardization, Languages.

Keywords safety analysis; fault tree; fault propagation taxonomy; AADL; automation; ARP4761.

1. Introduction

Cyber-physical systems, used in domains such as avionics or medical devices, perform critical functions where a fault might have catastrophic consequences (mission failure, severe injuries, etc.). Their development is guided by rigorous practice standards that prescribe safety analysis methods in order to verify that failure have been correctly evaluated and/or mitigated. Recommended practice standards for system safety assessment and certification, such as SAE ARP4761 [22] provide guidance for validation and verification of safety-critical software and systems. In current practice the safety assessment process is labor-intensive, mostly relying on engineer's ability to correctly interpret a textual specification of the system. This labor-intensive practice typically focuses system safety analysis on system engineering activities.
As reliance on embedded software for system operation grows, embedded software systems have become a major source of hazards [7]. The recent Boeing 787 generator control unit issue is a recent example of a software induced failure potential in a quad redundant system[1]. Studies show that 70% of embedded software system errors are introduced during requirements and architecture design phases and 80% are detected post unit test [16]. With rework cost of 100-1000 times the cost of in-phase correction, qualification related software rework makes up 50% of total system development cost [12]. As result software reliant mission and safety critical systems increasingly become unaffordable, increase safety risks, and delay in product delivery [10].
Automation of the safety analysis process is key to extending safety analysis to the embedded software system and to accommodate system design evolution. In this paper we focus on deductive fault impact analysis that identifies all potential contributors to an undesirable system malfunction or failure, in the form of fault tree analysis (FTA) and Common Cause Analysis (CCA).
We discuss three elements are key to safety analysis automation. First, generation of fault trees from annotated architecture models consistently reflects architecture changes in safety analysis results. We generate fault graphs and fault trees from architecture models of embedded software systems represented in the SAE Architecture Analysis & Design Language (AADL) standard [23] and annotated with fault behavior specifications expressed in the revised SAE AADL Error Model V2 Annex (EMV2) standard [24]. Second, use of a taxonomy of failure effects ensures that coverage of different types of potential hazard contributors is achieved. We utilize a taxonomy of error propagation types that has been defined as part of the EMV2 standard. Third, architecture models help identify sources of common cause failures. They are reflected in a fault graph, which is then transformed and reflected appropriately in a fault tree representation for probabilistic analysis. The approach is supported by the Open Source AADL Tool Environment

[1] http://gizmodo.com/the-boeing-dreamliner-has-a-bug-that-can-make-it-lose-a-1701599388

(OSATE)[2] and a recently developed open source FTA analysis tool that has been integrated with OSATE.

Model-based approaches have been deployed to safety analysis. For example, fault trees [21] are analyzed by tools to determine minimal cut sets (smallest combination of fault events leading to a system failure) and its occurrence probability. State-based behavioral models have been used to specify and verify nominal and fault behavior of systems, e.g., Petri nets [20], AltaRica [3] have been used to generate fault trees and to perform model checking on the specified behavior. To address the challenge of keeping a safety analysis model, such as fault trees [13], consistent with an evolving architecture, their generation from architecture models expressed in AADL [5][14], SysML [11], UML [15], Simulink [13][18] and annotated with fault information have been pursued and complemented with behavioral verification of safety conditions [9].

We have chosen AADL as it allows us to addressing software induced hazards and verify software design against these safety requirements to ensure that potential software errors are avoided or correctly handled. For example, for safety purposes, a software subsystem must be checked to ensure that it does not send more than its bandwidth allocation on a shared network in order to assure timely delivery of data between other subsystems on the same network. Other examples are: to ensure that a change in the deployment configuration of software onto hardware; to balance the workload across processors does not violate assumptions about physical redundancy by placing replicated software instances on different processor; or to isolate subsystems with different criticality level in into different partitions.

We have chosen EMV2 for expressing fault behavior annotations for two reasons. First, EMV2 allows users to specify fault behavior in three levels of abstraction. Second, it supports a type system that is used to represent an error propagation taxonomy, which has been defined as part of the EMV2 standard. The expressive power and semantics of EMV2 have been improved based on experience with the original Error Model Annex standard [5][14]. We have demonstrated that safety analysis reports as defined by the SAE ARP4761 [22] standard from the AADL models augmented with safety information on the wheel braking system example [4]. At the time we utilized composite error state specifications of EMV2, which express error states of a system in terms of error states of its subsystems in a logic equivalent to fault tree logic, to generate fault trees through composition of these specifications. We have also used AADL and EMV2 to diagnose a timing related software safety hazard and evaluated several proposed corrections to the problem [8].

In this paper we present a flow-based approach to generate fault trees. The flow-based approach interprets propagation paths and EMV2 error behavior specifications to trace from an undesirable system level effect or failure mode backwards to identify all possible contributors. The propagation paths are determined by connections and deployment bindings in the architecture model. The EMV2 error behavior specifications can be incoming and outgoing error propagations and related error path and error source specifications, as well as error behavior state machines with transitions triggered by error events and incoming propagations and affecting outgoing propagations. Error types associated with error events, states, and propagations are taken into consideration similar to a typed token systems such as colored Petri nets. In this process we take into account inclusive and exclusive OR, AND as well as priority AND, and k of n voting logic. Common cause contributors are identified in the architecture model by the fan-out of propagation paths, i.e., connections and deployment bindings. Handling of common cause fault events has been identified as an area of future work [14].

The paper proceeds as follow: first, we present the AADL language, its Error annex, and the error propagation taxonomy, and a description of a new open source fault tree analysis tool EMFTA that can process fault graphs. Then, we elaborate the flow-based approach to fault tree generation from AADL models annotated with fault behavior, including the handling of common cause contributors. Finally, we illustrate key aspects of our fault tree generation approach with an example.

2. Architecture and Fault Modeling Notations

In this work we will be utilizing the AADL notation to express the architecture of safety-critical software systems, the Error Model V2 Annex standard used to annotate the AADL model with fault safety related fault information, and the fault tree representation of an open source fault tree editor and analyzer called EMFTA. In the next sections we provide a summary of each.

2.1 AADL

AADL is a modeling language for embedded software systems standardized by SAE International. It allows users to describe the task and communication architecture of embedded software, its deployment on a hardware platform, and its interaction with a physical system within a single and consistent architecture model.

The core language specifies several categories of components. For each one the modeler defines a component type to represent its external interface, and one or more component implementations to represent a blue print in terms of subcomponents. For example, the task and communication architecture of the embedded software is modeled with thread and process components interconnected with port connections, shared data access and remote service call. The hardware platform is modeled as an interconnected set of processor, bus, and memory components. A device component represents a physical subsystem with both logical and physical interfaces to the embedded software system and its hardware platform. The system component is used to organize the architecture into a multi-level hierarchy. Users model the dynamics of the architecture in terms of operational modes and different runtime configurations through the mode concept. Users further characterize components through standardized properties, e.g., by specifying the period, deadline, worst-case execution time for threads. Users specify deployment binding of functional architectures to physical systems and software architectures to hardware platforms.

The language is extensible; users may adapt it to their needs using two mechanisms:

1. *User-defined properties*. New properties can be defined by users to extend the characteristics of the component. This is a convenient way to add specific architecture criteria into the model (for example, criticality of a subprogram or task)
2. *Annex languages*. Specialized languages can be attached to AADL components to augment the component description and specify additional characteristics and requirements (for example, specifying the component behavior). They are referred to as annex languages, meaning that they are added as an additional piece of the component. In this paper we will discuss the Error Model Annex language for specifying fault behavior.

The AADL model, annotated with properties and annex language clauses is the basis for virtual system integration and analysis of functional and non-functional properties along multiple dimensions from the same source, e.g., for performance analysis or safety evaluation. The same model can be used for generating

[2] Available at http://osate.github.io/

code and configuration files that are consistent with the verified model as shown in Figure 1.

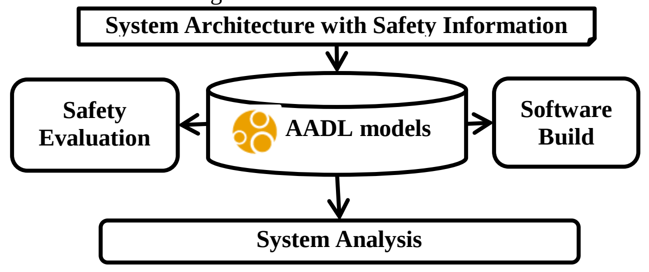

Figure 1 AADL ecosystem

2.2 Error Model Annex V2

EMV2 allows users to specify fault behavior of systems and their subsystems in terms of error propagations and flows, in terms of error behavior states and transitions triggered by error events and incoming propagations, and in terms of system error states as composites of subsystem error states. EMV2 introduces user-definable error types that when associated with error events, propagations, and states, act as a typed token system. A standard set of error types associated with error propagations acts as a taxonomy for characterizing failure effects.
EMV2 supports three levels of abstraction:

- *Focus on error propagation between components*: For each component the user can specify outgoing and incoming error propagations of error types being propagated and of error types expected to be contained. They represent a contract with outgoing propagation and containment specifications indicating guarantees and incoming ones indicating assumptions.
  The error propagation paths between components are determined by connections and deployment bindings. In addition, each component includes a specification of whether it is the source of an error propagation, the sink of an error propagation, or passes on incoming error propagations, possibly transforming the error type into a different one. This level of architecture fault model specification allows for hazard identification, fault impact analysis, and stochastic fault analysis, and aligns with the Fault Propagation and Transformation Calculus (FPTC) [17].
- *Focus on error behavior of a component*: For each component the user can specify an error event, i.e., activation of component-specific faults, recover and repair events, their occurrence probability, how they together with incoming error propagations affect the error state of the component, under what conditions outgoing error propagations occur, and when error behavior is detected and addressed by the component.
- *Focus on the composite error behavior of a component*: For each component with subcomponents the user can specify under what conditions in terms of subcomponent error states the component is in a particular error state. This mapping of subcomponent error state into a component error state abstraction reflects fault tree logic and allows for architecture fault analysis at different levels of the architecture hierarchy.

A more detailed presentation of EMV2 can be found in [6].

2.3 An Error Propagation Taxonomy

The EMV2 standard includes a standard set of error types that represents a taxonomy for characterizing commonly propagated failure effects. This taxonomy draws on previous work on formally specifying error propagation behavior [19][26]. The taxonomy views the interaction between two system components as one component providing a service to the other with the service consisting of a continuous or (as in the case of software) discrete stream of service items, e.g., a sensor providing a sequence of sensor readings. The taxonomy helps modelers ensure that they have covered common potential failure effects. The taxonomy consists of the following error types, which are organized into type hierarchies:

- *Omission and commission errors:* They can be omission and commission of the service as a whole (*ServiceOmission*, *ServiceCommission*), or omission and commission of individual service items (*ItemOmission*, *ItemCommission*). Examples of service omission and commission are loss and unexpected provision of electrical power. An example of item omission is loss of a message in transmission.
- *Timing errors:* They can be item related such as early or late arrival of messages (*EarlyDelivery*, *LateDelivery*), or stream related in the form of rate errors (*HighRate*, *LowRate*, *RateJitter*), or service related such as service starting early or late.
- *Value errors:* They can be for individual items such as *OutOfRange* sensor readings, *OutOfBounds* control state, or stream related such as *NonMonotonic*, *OutOfCalibration*, or *StuckValue*.
- *Replication errors:* They deal with replicates of service and include symmetric and asymmetric value, timing, and omission errors.
- *Concurrency errors:* They deal with access to shared resources by concurrent tasks, e.g., *RaceCondition* and *MutexError*.
- *Authentication and authorization errors:* They deal with security issues.

These error types are associated with outgoing and incoming error propagations defined for each of the interaction points of a component with other components, e.g., ports, or deployment binding points. Error types of outgoing propagations (guarantees) must match those of incoming propagations (assumptions). Users can explicitly error types that are expected not to be propagated. This allows us to ensure that absence of an error type in an error propagation is not an oversight.
Users can define domain specific aliases for these error types, e.g., *NoPower* for *ServiceOmission*, and can extend the taxonomy. Users can also introduce libraries or error types to be associated with error events associated with different types of components, e.g., a particular type of sensor. Those types represent different ways a component can fail. The perceived effect of such failures tends to be one of the error types in the taxonomy.

2.4 Fault Tree Representation and Analysis

The SAE ARP4761 standard describes the Fault Tree Analysis (FTA) [1][2] as a deductive fault impact analysis that focuses on identifying all potential causes, i.e., contributors, that lead to a particular undesired system failure. The Fault Tree Handbook [25] defines a representation for fault trees consisting of alternating events and gates.
Figure 2 illustrates such a fault tree with the root event specifying the system failure is interest, a computer crash. An OR gate indicates that any of three contributors can result in the crash, namely, a broken device, an unhandled interrupt, or a software error. The software error is refined by an AND gate to indicate that a divide by zero occurs and a no recovery handler is present. The figure also shows the occurrence probabilities associated with

leaf events and calculated for enclosing intermediate events including the root event.

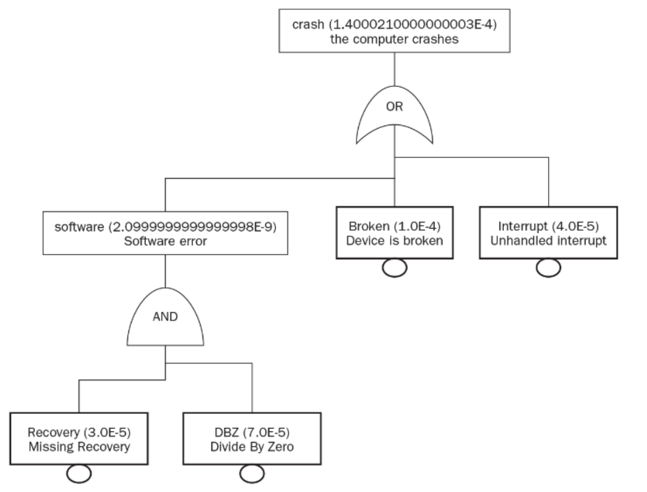

Figure 2 Example of a Fault Tree

We have developed an open source graphical tool to create, visualize and analyze fault trees, EMF-based Fault Tree Analysis (EMFTA). Its development was motivated by the lack of actively maintained open source tools for fault tree analysis. Commercial tools are available for a high cost that can be a barrier for research activities.

EMFTA has been developed on top of Eclipse. It is available as a stand-alone tool on a github repository[3] and released under the BSD license. We also have integrated it with OSATE by combining it with the fault tree generator discussed in this paper. From within OSATE users can invoke fault tree analysis on a system instance model, which will automatically generate the fault tree and open it in EMFTA.

EMFTA provides two representations for creating, visualizing, and analyzing a fault tree:

- *Graphical representation:* tree representation, as shown in Figure 2.
- *Table representation:* spreadsheet-style representation, easier to edit.

Both representation operate on the same underlying fault tree model and are synchronized. A change in the tree representation is automatically reflected in the table representation.

The underlying fault tree model actually represents a fault graph, where common cause (dependent) events and subtrees are not replicated but referenced by different gates. Think of a shared event or subtree as *transfer out* symbol [25] that can be referenced by multiple *transfer in* symbols.

EMFTA includes the following analysis capabilities:

- *Minimal cut set generation:* determines the minimal combinations of events that lead to a system failure event.
- *Occurrence probability computation:* computes the occurrence probability of any intermediate and the root events from leaf event probabilities taking the gate logic into account. It also ensures that the fault tree has a consistent set of probability values associated with each event.

The fault tree model in EMFTA supports the following event types (see also [25]):

- *Basic* to represent leaf events (shown as rectangle with a small circle),
- *Intermediate* to represent the root and subtree events with gates (shown as rectangle),
- *External* to represent events that occur outside the system of interest but contribute to the system failure (shown as rectangle with a small pentagon),
- *Undeveloped* to represent events that at this time are not further developed due to lack of information (shown as rectangle with a small diamond), and
- *Conditioning* to represent a condition that applies to a logic gate such as Priority AND and Inhibit (shown as rectangle with a small oval).

The fault tree model in EMFTA supports the following gate types:

- *AND* to indicate that a fault occurs if all faults represented by subevents occur,
- *OR* to indicate that a fault occurs if at least one of the faults represented by subevents occurs,
- *Exclusive OR* to indicate that a fault occurs if exactly one of the faults represented by subevents occurs,
- *Priority AND* to indicate that a fault occurs if all faults represented by subevents occur in sequence,
- *Intermediate* to indicate that a fault occurs if a single subevent fault occur, i.e., to associate a single subevent with an event,
- *Inhibit* to indicate that a fault occurs if a single subevent fault occurs in the presence of a condition specified by an associated Conditioning event,
- *K of n Voting OR* to indicate that a fault occurs if at least k of the subevent faults occur.

3. Generation of Fault Trees

We generate fault trees from AADL models that are annotated with EMV2 specifications by creating a system instance model from a root system implementation, and by identifying the failure condition (error state or outgoing error propagation and possibly error type) of interest to become the fault tree root event. We support the composition of fault trees from composite error state specifications and flow-based fault tree generation.

The generator supports the creation of fault trees based on composite error state specifications. This may be useful early in development when the user has specified a parts model, i.e., the component implementation declaration only contains subcomponents without connections. In that case, the user can associate error states (failure modes) with each component and define a composite error state specification to indicate which combination of subcomponent error states leads to a particular error state of the enclosing system component. This condition is expressed in terms of a combination of *AND¸ (exclusive) OR, k ORMORE, k ORLESS* operators. The logic expressions of these components are recursively combined into a fault tree.

In the flow-based fault tree generation, the AADL model includes connections and possibly bindings from a functional architecture to a physical architecture, or an application software architecture to a virtual or physical computer platform. These connections and bindings represent propagation paths between components of a system, i.e., determine that outgoing error propagations of one component can impact another component. Furthermore, components of the system are expected to have at least error source, sink, and path specifications. Users may have elaborated the EMV2 specification of a component with a component error behavior in terms of an error behavior state machine with a set of

[3] EMFTA repository: https://github.com/cmu-sei/emfta

error events, transitions, and outgoing propagation condition declarations.
We proceed by first describing how EMV2 constructs are mapped into fault tree fragments, then discussing how common cause (dependent) events and subtrees are easily identified from the AADL model, and outlining the transformations applied to the fault graph to flatten it and minimize multiple references to events, and finally point out special case processing of *exclusive OR* operators in error state and outgoing propagation conditions.

3.1 Mapping of EMV2 Constructs into Fault Tree Elements

The starting point of flow-based fault tree generation is an outgoing error propagation of the root component in an AADL system instance model. It represents the undesirable system failure of interest. If the outgoing error propagation has multiple error types, the user selects the type of failure of interest. It becomes the root event of the fault tree.
We traverse propagation paths and error flows backwards to identify contributors to the failure. The feature of the outgoing propagation has a connection from one or more subcomponents. These are combined with an OR gate to indicate that any of them can lead to a failure. For each subcomponent outgoing error propagation we follow error flows if no component error behavior specification exists. Error sources become *Basic* events. In case of error paths the incoming error propagation of the path is used to follow the propagation path (connection or binding) to its predecessor. Multiple error paths and error sources for the same outgoing propagation are combined by an OR gate. Similarly, multiple connections from a feature of an incoming error propagation are combined with an OR gate.
If a component has an error behavior specification we interpret the error state machine with error events and incoming propagations triggering transitions to different error states, and outgoing propagation condition (OPC) declarations indicating which combination of error states and incoming error propagations result in a particular outgoing error propagation.
First, we elaborate on how OPC are handled. For OPC, the condition expression and the error state are interpreted and combined under a Priority AND gate with the state related subtree occurring before the trigger condition subtree. If the outgoing propagation condition applies to all states (*all* keyword on left) only the condition is interpreted.
The OPC condition, if not empty, identifies incoming error propagations. They are followed to outgoing propagations of connected/bound components by following propagation paths as discussed earlier.
The error state of an OPC or leaf error state of a composite error state specification is elaborated as follows. Any transition with the state as target is processed and multiple transitions are combined by an OR gate. The transition trigger condition identifies error events, which are represented by *Basic* events. Incoming error propagations referenced in the trigger conditions are processed as described earlier. The various subtrees are combined according to the condition logic operators (see below for details). The source state of a transition is processed backwards recursively if it includes transitions by error events and combined by a Priority AND gate. This captures situations where a *failed* state is reached from a *degraded* state through an error event and the *degraded* state was reached by a previously occurring error event.
The logic operators of the composite state condition as well as the trigger condition of error state transitions and outgoing propagation conditions are mapped into fault tree gates as follows: AND maps into an AND gate; *(exclusive)* OR maps into an Exclusive OR gate; 1 ORMORE maps into an *OR* gate; k ORMORE maps into a Voting OR gate; k ORLESS is typically used to specify k or fewer error free states, thus can be mapped into an equivalent Voting OR gate.
Incoming propagation that represent bindings, but are not bound, are mapped into *Undeveloped* events. Similarly, incoming propagations that do not have a propagation path to a sending component are mapped into *Undeveloped* events. These events will be expanded in the future when binding specifications or connections have been added to the AADL model. Incoming propagations that trace back to an incoming propagation of the top system, are mapped to *External* events. For example, the incoming error propagation of the top level system represents an exceptional condition in the operational environment that affects the safe operation of the system, e.g., a failure condition of an external power supply.
Note that as we generate the fault tree, we start with the error type identified by the user and perform the appropriate filtering and mappings according to the specified type set constraints. In other words, the error type being propagated affects which error flows, transitions, and OPCs are included in the generated fault tree.

3.2 Generating a Fault Graph

When generating fault trees we have to be concerned with common cause (dependent) events and subtrees. For example, a power supply supplying multiple sensors becomes a common failure cause. Similarly, a processor failure results in all software tasks executing on that processor to fail, or one task overrunning its execution time budget causing other tasks on the same processor to miss their deadlines. We have explicit knowledge of such common cause propagation paths in the AADL model in the form of multiple connections from the same component or multiple bindings to the same resource, i.e., a fan out of propagation paths.
The generation process utilizes this information to generate a fault graph by having common cause events and subtrees referenced by multiple gates whose events are impacted by the common cause event. The fault graph makes a common cause analysis (CCA) simple in that any event that is the target of multiple gate references is a candidate. Note that this fault graph can be visualized by the EMFTA tool as a graph or as a fault tree by replicating common cause events and subtrees.

3.3 Fault Tree Transformations

We apply two types of transformation to the generated intermediate fault graph: flattening of nested gates of the same type, and moving events as target of multiple references towards the root to reduce or eliminate multiple references.
When flattening nested gates we take advantage of the fact that fault tree gates represent n-ary logical operators. For example the generation process may produce an event with an OR gate, one or more of whose subevents have OR gates themselves. This is the case when tracing back propagation paths across multiple components. These nested OR gates can be reduced to a single OR gate with all the “leaf” subevents. We do so for OR, AND, Exclusive OR, Priority AND gates.
For common cause events it is desirable to move them up towards the root of the fault graph. For example, in a two sensor system the system fails if sensor1 and sensor2 fail or if the power supply to the sensors fails. The initially generated fault graph has an AND gate with an OR gate under each subevent to represent the sensor failure and the common cause power supply failure, i.e., *(s1 OR power) AND (s2 OR power)*. By applying the *Distributive Law* this can be transformed into *power OR (s1 AND s2)*. Note that the *power* event is now referenced only once.
A second transformation that moves common cause events and subtrees towards the root is the application of the *Law of*

Absorption. For example, if the original fault graph has *power AND (s1 OR power)* the result of the transformation is *power*. The same applies for *power OR (s1 AND power)*.

Finally, we apply the *Idempotent Law*, which allows us to reduce multiple instances of the same subevent under a gate to a single subevent, i.e., *power OR power* becomes *power*.

These transformations may turn the fault graph into a fault tree of independent events. This allows for a simplified calculation of occurrence probabilities.

3.4 Processing Exclusive OR Conditions

Users may specify that an error state is reached or a particular error type is propagated out only if one of several incoming propagations or error events occur. For example, a control system may go into degraded mode if it loses one of its sensor input and as result it controls a system with less precision. Therefore, subtrees of an Exclusive OR gate generated from an EMV2 exclusive OR operator cannot contain common cause events or subtrees. This is taken into account when we generate the fault graph. Figure 7 in the next section shows the resulting fault tree for such a scenario.

4. Illustrated Example

We demonstrate the fault tree generation capability on a Global Positioning System (GPS) receiver example[4]. The GPS receiver is shown in Figure 3. It consists of two satellite signal receivers picking up satellite signals. The signals are processed by a processing unit, which provides high precision location information when it has both signals and low precision when one signal is available. The processing unit executes on, i.e., is bound to a CPU. The receivers communicate over a network with the CPU. A power supply supplies both receivers, the network, and CPU. Error propagation paths are determined by the port connections representing data flow, bus access connections that interconnect the hardware, and abstract feature connections for electrical power.

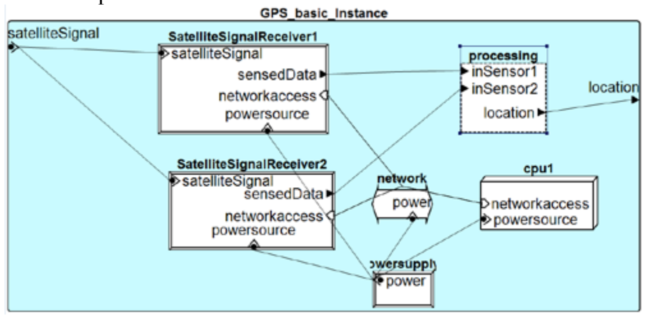

Figure 3 GPS Receiver

Figure 4 shows the AADL and EMV2 error propgation and flow specificaiton of the GPS processing unit. It is defined as an abstract component as we have not yet decided whether it will execute as a thread in the same or a different process on the same processor. The two incoming ports have incoming error propagation declarations indicating that *ServiceOmission* is being propagated. The outgoing propagation declaration for location indicates that *ServiceOmission* as well as *LowPrecisionData* and *IncorrectData* are potentially propagated. The incoming propagation labelled *processor* identifies error propagation due to processor binding. The error paths specify how incoming error propagations are passed on. The error source declaration indicates that the processing unit is the source of low precision and incoorect location data.

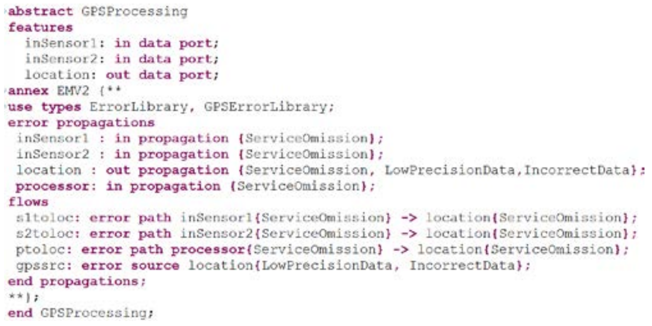
```
abstract GPSProcessing
features
  inSensor1: in data port;
  inSensor2: in data port;
  location: out data port;
annex EMV2 {**
use types ErrorLibrary, GPSErrorLibrary;
error propagations
 inSensor1 : in propagation {ServiceOmission};
 inSensor2 : in propagation {ServiceOmission};
 location : out propagation {ServiceOmission, LowPrecisionData,IncorrectData};
 processor: in propagation {ServiceOmission};
flows
 s1toloc: error path inSensor1{ServiceOmission} -> location{ServiceOmission};
 s2toloc: error path inSensor2{ServiceOmission} -> location{ServiceOmission};
 ptoloc: error path processor{ServiceOmission} -> location{ServiceOmission};
 gpssrc: error source location{LowPrecisionData, IncorrectData};
end propagations;
**};
end GPSProcessing;
```

Figure 4 Error Propagation and Flows for GPS Processing

Figure 5 expands the EMV2 specification for the GPS processing unit by adding error state behavior. This includes a *computeError* error event, which triggers a transition to the *Incorrect* error state, a transition to indicate low precision processing due to single input, and a transition to the *NoService* state if no input is available. Outgoing propagation condition declarations specify when *ServiceOmission*, *LowPrecisionData*, and *IncorrectData* is propagated.

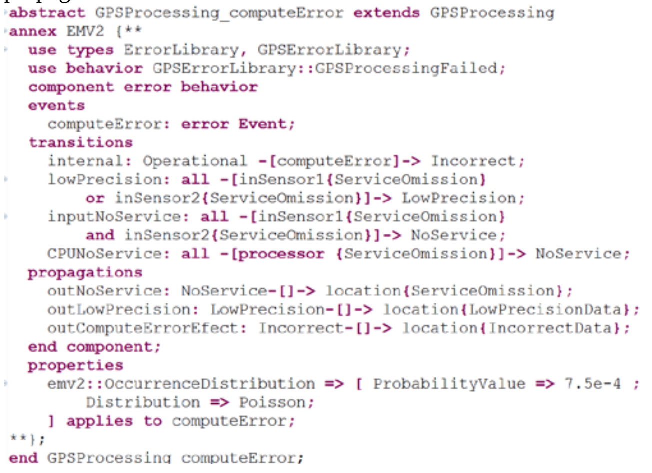
```
abstract GPSProcessing_computeError extends GPSProcessing
annex EMV2 {**
  use types ErrorLibrary, GPSErrorLibrary;
  use behavior GPSErrorLibrary::GPSProcessingFailed;
  component error behavior
  events
    computeError: error Event;
  transitions
    internal: Operational -[computeError]-> Incorrect;
    lowPrecision: all -[inSensor1{ServiceOmission}
        or inSensor2{ServiceOmission}]-> LowPrecision;
    inputNoService: all -[inSensor1{ServiceOmission}
        and inSensor2{ServiceOmission}]-> NoService;
    CPUNoService: all -[processor {ServiceOmission}]-> NoService;
  propagations
    outNoService: NoService-[]-> location{ServiceOmission};
    outLowPrecision: LowPrecision-[]-> location{LowPrecisionData};
    outComputeErrorEfect: Incorrect-[]-> location{IncorrectData};
  end component;
  properties
    emv2::OccurrenceDistribution => [ ProbabilityValue => 7.5e-4 ;
        Distribution => Poisson;
    ] applies to computeError;
**};
end GPSProcessing computeError;
```

Figure 5 Error State Behavior for GPS Processing

Figure 6 shows the fault tree that has been generated and transformed for the GPS receiver system providing no service as failure condition. It shows that this condition occurs when we receive no satellite signal (an *External* event), or the network, or CPU, or power supply fail. It also occurs when both receivers fail. Note that in the initially generated fault graph the power supply failure event and the network failure event were common cause events contributing to *ServiceOmission* as incoming receiver signal error propagation to the GPS processing unit, i.e., they would have been referenced by two separate OR gates (and intermediate event) together one of the receiver failure event under the AND gate. Note that fault tree shows computed occurrence probabilities.

[4] The GPS Receiver example with additional use cases is available at https://github.com/osate/examples/tree/master/SafetyTutorial

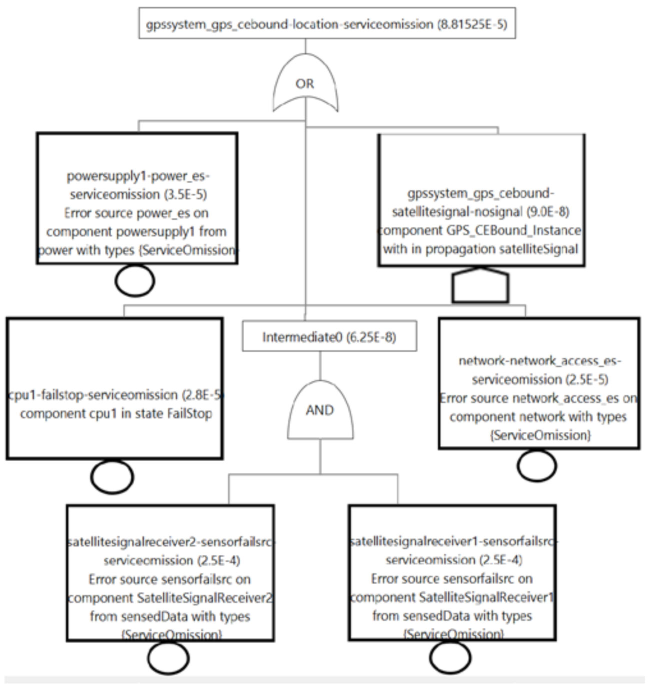
gpssystem_gps_cebound-location-serviceomission (8.81525E-5)
OR
powersupply1-power_es-serviceomission (3.5E-5) Error source power_es on component powersupply1 from power with types {ServiceOmission
gpssystem_gps_cebound-satellitesignal-nosignal (9.0E-8) component GPS_CEBound_Instance with in propagation satelliteSignal
cpu1-failstop-serviceomission (2.8E-5 component cpu1 in state FailStop
Intermediate0 (6.25E-8)
AND
network-network_access_es-serviceomission (2.5E-5) Error source network_access_es on component network with types {ServiceOmission}
satellitesignalreceiver2-sensorfailsrc-serviceomission (2.5E-4) Error source sensorfailsrc on component SatelliteSignalReceiver2 from sensedData with types {ServiceOmission}
satellitesignalreceiver1-sensorfailsrc-serviceomission (2.5E-4) Error source sensorfailsrc on component SatelliteSignalReceiver1 from sensedData with types {ServiceOmission}

Figure 6 Location Service Omission for GPS

Figure 7 shows the generated and transformed fault tree for the same GPS receiver system for the condition that it produces low precision location data. As expected it only consists of an Exclusive OR gate of the two receivers failing themselves.

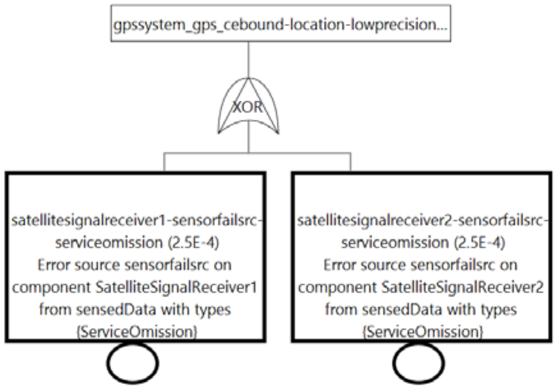
gpssystem_gps_cebound-location-lowprecision...
XOR
satellitesignalreceiver1-sensorfailsrc-serviceomission (2.5E-4) Error source sensorfailsrc on component SatelliteSignalReceiver1 from sensedData with types {ServiceOmission}
satellitesignalreceiver2-sensorfailsrc-serviceomission (2.5E-4) Error source sensorfailsrc on component SatelliteSignalReceiver2 from sensedData with types {ServiceOmission}

Figure 7 Low Precision Location for GPS

5. Conclusion

System safety analysis has been a labor intensive process where safety engineers develop and analyze safety models such as fault trees. As safety-critical systems increasingly rely on software, embedded software systems have become a major hazard contributor. Model-based approaches have been pursued to address the challenge of keeping safety models consistent with an evolving system architecture and design by leveraging system models that are annotated with safety information.

In this paper we have presented an approach of automated safety analysis that combines the generation of safety models, in our case fault trees, with a taxonomy of error propagation types to achieve coverage of potential failure effects, and with a way to take into account common cause failures. The approach utilizes the SAE AADL standard to represent embedded software system architectures and the SAE EMV2 standard, which includes the above mentioned taxonomy, to annotate AADL models with fault behavior information. The error propagation type taxonomy is used to characterize outgoing and incoming propagations to represent failure effects that are and are not expected to be propagated (guarantees) and received (assumptions).

We have presented a flow-based approach to the generation of fault trees from AADL models annotated with fault behavior information expressed in EMV2. This approach supports the interpretation of fault behavior specification at three levels of abstraction using the revised Error Model Annex V2 standard. It takes into account the fault propagation level abstraction of EMV2 specification as well as error behavior specifications expressing failure modes and their triggers in terms of error events and incoming propagations. It also supports the composition of fault trees from composite error state specifications in EMV2.

We have discussed how our approach explicitly represents common cause events by leveraging knowledge about common cause sources from the architecture model, and then applies transformations to flatten the structure and eliminate multiple references to events by moving then closer to the root event. The resulting fault tree is then amenable to provide more accurate occurrence probability calculations. We have also shown that *exclusive OR* operators in EMV2 require special attention in the generation of fault trees.

The fault tree generator has been integrated with OSATE, a tool environment for AADL. We have also developed an open source tool for visualizing and analyzing generated fault trees as well as editing fault trees manually called EMFTA. We have also released EMFTA under the BSD license with the hope that other modeling framework can leverage it and integrate its safety analysis capabilities.

Previous work by us and others has shown the value of automatically generating fault trees from architecture models annotated with fault behavior. We have been able to demonstrate that several methods in support of system safety analysis following best practices such as SAE ARP4761 can be supported from a single model. The AADL-based approach has been applied to various systems, including an aircraft wheel braking system, a situational awareness system, satellite systems, medical devices, and a stepper motor based engine control system. The automation of these safety analyses has allowed users to continuously re-evaluate safety properties as architecture design alternatives are being considered and as architecture designs are refined and evolve.

Acknowledgments

This material is based upon work funded and supported by the Department of Defense under Contract No. FA8721-05-C-0003 with Carnegie Mellon University for the operation of the Software Engineering Institute, a federally funded research and development center.

References

[1] Andrews, J. "Fault Tree Analysis." Proceedings of the 16th International Safety Conference, www. fault-tree. net/papers/andrews-fta-tutor. pdf (Stand 12/2004). 1998.

[2] Barlow, R. E. Fault Tree Analysis. John Wiley & Sons, Inc., 1973.

[3] Bieber, P., C. Castel, and C. Seguin. “Combination of Fault Tree Analysis and Model Checking for Safety Assessment of Complex Systems”, in 4th European Dependable Computing Conference, 2002.

[4] Delange, J., et al. "AADL Fault Modeling and Analysis within an ARP4761 Safety Assessment.", Software Engineering Institute, CMU/SEI-2014-TR-020 (2014).

[5] Ern, B., V. Y. Nguyen, T. Noll. “Characterization of Failure Effects on AADL Models.” Proceedings of the 32nd International Conference on Computer Safety, Reliability and Security (SAFECOMP 2013), Volume 8153 of LNCS, Springer, 2013.

[6] Feiler, P., et.al. “Architecture Fault Modeling and Analysis with the Error Model Annex, Version 2”, Software Engineering Institute, CMU/SEI-2016-TR-009 (2016).

[7] Feiler, P. “Challenges in Validating Safety-critical Embedded Systems.” In SAE International AeroTech Congress, Nov 2009.

[8] Feiler, P., et.al. Architecture-led Diagnosis and Verification of a Stepper Motor Controller. 8th European Congress on Embedded Real Time Software & Systems (ERTS 2016). Jan 2016. http://www.erts2016.org/

[9] Ghassabani, E., A. Gacek, M. Whalen. “Efficient Generation of Inductive Validity Cores for Safety Properties.” ACM SIGSOFT International Symposium on the Foundations of Software Engineering (FSE 2016), November 2016.

[10] Hagen, C., J. Sorenson. “Delivering Military Software Affordably.” Defense AT&L. March-April 2013. http://www.dau.mil/pubscats/ATL%20Docs/Mar_Apr_2013/Hagen_Sorenson.pdf

[11] Hecht, M., et.al. “Using SysML to Automatically Generate of Failure Modes and Effects Analyses”, INCOSE International Symposium, Volume 25, Number 1, 2015.

[12] Helton, S., D. Ward. “Estimating Return on Investment for SAVI: A Model-based Virtual Integration Process.” SAE International AeroTech Congress. Oct 2011.

[13] Joshi, A., S. Miller, M. Whalen, M. Heimdahl. “A Proposal for Model-based Safety Analysis.” In Proceedings of the 24th Digital Avionics Systems Conference (DASC 2005), Oct 2005.

[14] Joshi, A., P. Binns, S. Vestal, “Automatic Generation of Fault Trees from AADL Models”, 1st International Workshop on Aerospace Software Engineering, in conjunction with International Conference on Software Engineering (ICSE), May 2007.

[15] Lauer, C., R. German, and J. Pollmer. "Fault Tree Synthesis from UML Models for Reliability Analysis at Early Design Stages." ACM SIGSOFT Software Engineering Notes 36.1 (2011): 1-8.

[16] National Institute of Standards and Technology (NIST). “The Economic Impacts of Inadequate Infrastructure for Software Testing” Technical report, 2002. http://www.nist.gov/director/prog-ofc/report02-3.pdf.

[17] Paige, R., et.al. “FPTC: Automated Safety Analysis for Domain-Specific Languages.” In Models in Software Engineering. Lecture Notes in Computer Science. Volume 5421. Pages 229–242. Springer-Verlag. 2009

[18] Papadopoulos Y., et.al. “Engineering Failure Analysis & Design Optimisation with HiP-HOPS”, Journal of Engineering Failure Analysis, Elsevier Science, 2011.

[19] Powell, D. “Failure Mode Assumptions and Assumption Coverage.” In Fault-Tolerant Computing, 1992. FTCS-22.Digest of Papers, Twenty-Second International Symposium on, pages 386–395, 1992.

[20] Rugina, A., K. Kanoun, and M. Kaâniche. “A System Dependability Modeling Framework Using AADL and GSPNs.” In *Architecting Dependable Systems IV* Lecture Notes In Computer Science, Vol. 4615. Springer-Verlag. 2007.

[21] Ruijters, E., and M. Stoelinga. "Fault Tree Analysis: A Survey of the State-of-the-art in Modeling, Analysis and Tools." Computer Science Review 15 (2015): 29-62.

[22] SAE International. ARP4761. "Guidelines and methods for conducting the safety assessment process on civil airborne systems and equipment." SAE International (1996): 1-331.

[23] SAE International. AS5506B. “Architecture Analysis and Design Language (AADL)”, 2012. https://saemobilus.sae.org/content/as5506b.

[24] SAE International. AS5506/1A. “SAE Architecture Analysis and Design Language (AADL) Annex Volume1A – Error Model Annex V2”, Sept 2015, https://saemobilus.sae.org/content/as5506/1a.

[25] Vesely, W. E., et al. Fault Tree Handbook. No. NUREG-0492. Nuclear Regulatory Commission Washington DC, 1981.

[26] Walter, C.J., N. Suri. “The Customizable Fault/Error Model for Dependable Distributed Systems.” Theor. Comput. Sci., Vol. 290, No. 2, pp. 1223-1251. Jan 2003. .