Carnegie Mellon University
Software Engineering Institute

A Semantics of AADL EMV2 and Its Application to Model-Based Fault Tree Generation

Aaron Greenhouse
Jérôme Hugues
Sam Procter
Lutz Wrage
Joseph R. Seibel

September 2025

TECHNICAL REPORT
CMU/SEI-2025-TR-002
DOI: 10.1184/R1/24653841

Software Solutions Division

[DISTRIBUTION STATEMENT A] Approved for public release and unlimited distribution.

http://www.sei.cmu.edu

Copyright 2025 Carnegie Mellon University.

This material is based upon work funded and supported by the Department of Defense under Contract No. FA8702-15-D-0002 with Carnegie Mellon University for the operation of the Software Engineering Institute, a federally funded research and development center.

The view, opinions, and/or findings contained in this material are those of the author(s) and should not be construed as an official Government position, policy, or decision, unless designated by other documentation.

NO WARRANTY. THIS CARNEGIE MELLON UNIVERSITY AND SOFTWARE ENGINEERING INSTITUTE MATERIAL IS FURNISHED ON AN ”AS-IS” BASIS. CARNEGIE MELLON UNIVERSITY MAKES NO WARRANTIES OF ANY KIND, EITHER EXPRESSED OR IMPLIED, AS TO ANY MATTER INCLUDING, BUT NOT LIMITED TO, WARRANTY OF FITNESS FOR PURPOSE OR MERCHANTABILITY, EXCLUSIVITY, OR RESULTS OBTAINED FROM USE OF THE MATERIAL. CARNEGIE MELLON UNIVERSITY DOES NOT MAKE ANY WARRANTY OF ANY KIND WITH RESPECT TO FREEDOM FROM PATENT, TRADEMARK, OR COPYRIGHT INFRINGEMENT.

[DISTRIBUTION STATEMENT A] This material has been approved for public release and unlimited distribution. Please see Copyright notice for non-US Government use and distribution.

This work is licensed under a Creative Commons Attribution-NonCommercial 4.0 International License. Requests for permission for non-licensed uses should be directed to the Software Engineering Institute at permission@sei.cmu.edu.

DM25-0203

Table of Contents

[DISTRIBUTION STATEMENT A] Approved for public release and unlimited distribution.

[DISTRIBUTION STATEMENT A] Approved for public release and unlimited distribution.

[DISTRIBUTION STATEMENT A] Approved for public release and unlimited distribution.

List of Figures

[DISTRIBUTION STATEMENT A] Approved for public release and unlimited distribution.

Abstract

The Architecture Analysis & Design Language (AADL) is an SAE International standard for the design and analysis of both the software and hardware architecture of performance-critical real-time systems. The Error Model Annex, Version 2 (EMV2), extends AADL with concepts to perform safety analyses, such as error types, error propagations, and the impact of errors propagated on components. EMV2 builds on AADL concepts of components and ports to define error propagations and error state machines. These definitions rely on a precise definition of the effect of an error being triggered in this system. The definitions of these concepts rely on powerful abstractions, yet they are mostly defined in plain text. This report first proposes a formal semantics for EMV2. Then, it shows how to leverage this semantics to generate fault trees from an AADL model enriched with EMV2 information. Defining a formal semantics improves the understanding of the EMV2 model, and the precision of model transformation from EMV2 to analysis techniques.

[DISTRIBUTION STATEMENT A] Approved for public release and unlimited distribution.

1 Introduction

The Architecture Analysis & Design Language (AADL) is an SAE International standard for the design and analysis of both the software and hardware architecture of performance-critical real-time systems [26]. AADL describes how components in such systems interact, how data inputs and outputs are connected, and how software components are bound to hardware components. The dynamic behavior of the system can be described via operational modes and mode transitions: that is, AADL can describe the different configurations of the system and the transitions between those configurations.

AADL provides a standard and machine-processable way to model system architecture that permits the analysis of system properties. The language can, for example, describe timing requirements, time and space partitioning, and safety properties. A system designer can then perform analyses of the *composed components* that, for example, determine process schedulability, communication latency, or even size and weight bounds for the physical system. From these analyses, the designer can evaluate architectural tradeoffs and changes.

AADL can be used for multiple activities in multiple development phases, beginning with preliminary system design. The language can be used by multiple tools to automate various levels of modeling, analysis, implementation, integration, verification, and certification. Finally, AADL is extensible: extensions, in the form of new properties and analysis-specific notations, can be associated with components written in the core language. Notational extensions are accessed via `annex` clauses within AADL component declarations. One such extension is the Error Model Annex, Version 2 (EMV2) [27].

EMV2 is intended "to support qualitative and quantitative assessments of system reliability, availability, safety, security, survivability, robustness, and resilience, as well as assure compliance of the system design and implementation to the fault mitigation strategies specified in the annotated architecture model of the embedded software, computer platform, and physical system" [27, E.1.(1)]. The notational language extensions, accessed through `annex EMV2 {** ... **}` clauses in the AADL core text, enable the specification of fault types, fault behavior of individual components, fault propagation affecting related components, aggregation of fault behavior and propagation in terms of the component hierarchy, and specification of fault tolerance strategies expected in the actual system architecture [27, E.1.(3)]. Error modeling is supported at three levels of abstraction:

1. *Error propagation and flow* focuses on components as error sources, sinks, and paths and enables analysis of the effects of the errors propagated by one component on the errors propagated by another component.
2. *Component error behavior* focuses on fault models of individual components. The model identifies faults in the component, determines how they manifest as failure, and describes their interaction with incoming error propagations to produce outgoing error propagations. Furthermore, the different error behavior states of the system and transitions between them are declared.
3. *Composite error behavior* focuses on relating the fault models of a system's (component's) subsystems (subcomponents) to the fault model of the system itself.

Finally, EMV2 is intended to enable the assessments and analyses of *SAE ARP4761 Guidelines and Methods for Conducting the Safety Assessment Process on Civil Airborne Systems and Equipment* [24] to be applicable to AADL models. Such assessments are diverse and include functional hazard assessments, failure mode and effect analyses, fault tree analysis, and common cause analysis.

[DISTRIBUTION STATEMENT A] Approved for public release and unlimited distribution.

1.1 This Report

Performing any one of the analyses of ARP4761 in an effective manner *requires understanding exactly what behavior is described by an EMV2 model*. The purpose of this report is thus to

- provide a formal semantics of EMV2
- demonstrate the usefulness of the semantics to the above intent by relying on the semantics to formally describe how to generate a fault tree from a system with an AADL model incorporating EMV2 error models

This report focuses on the *component error behavior* abstraction layer of EMV2. This layer describes the behavior of each component as a state machine with a complex relationship to internal error events and external error propagations:

> The EMV2 supports the specification of component error behavior in terms of an error behavior state machine with a set of states and transitions that occur under specified conditions, as well as specification error, recover, and repair events that are local to a component. They are associated with components to specify how the error state of a component changes due to error events and error propagations as well as due to repair events. Error events and states can indicate the type of error they represent by referring to error types. The error behavior specification also declares the conditions for outgoing error propagation in terms of the component error behavior state and incoming error propagations. For example, the error state of a component might change due to an error event of the component itself, and/or due to an error propagated into that component from some other component. This allows us to characterize the error behavior of an individual component in terms [of] its own error events and in terms [of the] impact of incoming error propagations from other components, as well as conditions under which outgoing error propagations occur that can impact other components. [27, E.1.(8)]

The many nuances of the relationships and interactions of the features of the component behavior model are complicated and benefit from a formal semantic presentation.

1.2 A Semantics for EMV2

Our goal for the semantics presented herein is to contribute to a framework for reasoning about the error behavior (e.g., propagations and state transitions) of a complete AADL system that contains EMV2 error behavior declarations. Because EMV2 error behaviors define state-based automata, such a system is a *collection* of automata connected to each other via propagations (see Figure 1.1):

- The error behavior of each component as a whole is an automaton (e.g., B, C, D, and S).
- The specific EMV2 component error behavior is *one component* of the component's full error behavior.
- The full behaviors of a component's subcomponents contribute to the component's full error behavior.
- Even the EMV2 events themselves (e.g., E and F) are modeled as automata that contribute to the overall component behavior. An occurrence of an event is represented by an output from the event's automaton.
- Inputs received from parent and sibling automata feed a component's EMV2 behavior automaton *and* its subcomponents' automata.

[DISTRIBUTION STATEMENT A] Approved for public release and unlimited distribution.

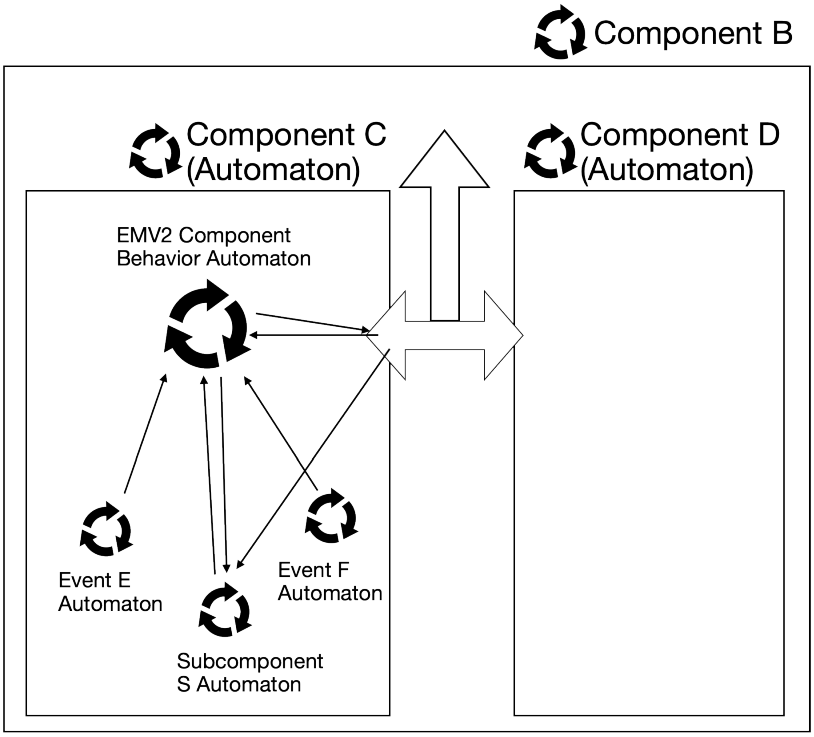

Figure 1.1: Diagram Showing the Relationships Among Behavioral and Event Automata

- Similarly, a component automaton's output feeds the automata of its parent and siblings.

A contribution of this work is to provide a formal description of the EMV2 behavior automaton *associated with each component.* This is an automaton specialized for the specifics of EMV2 that combines concepts from multiple formalisms: probabilistic automata [23, 32], Mealy machines [18], and symbolic automata [9, 33].

The definition of the behavior automaton is built on top of a *denotational semantics* [17, 28] of the component error behavior and error types. EMV2 features that are not currently included in the semantics include the *typed* aspect of the state machine, recover and repair events, type mappings, error detection, composite error behavior, and error flows.

1.2.1 Condition Expressions

The main source of complexity in reasoning about EMV2 behavior is the "condition expression" used to control state transitions, out propagations, and error detections (although these are not dealt with herein). These are rich logical expressions built from more basic "trigger expressions" that test for the occurrence of an internal event or specific incoming error type propagation. These can be combined using the standard Boolean operators. Additional primitive operations, such as "all but n" and "n or less," with their own complex meanings are also available. Furthermore, condition expressions used in transitions may be evaluated with an additional set of *implicit* assumptions about the component's received propagations. All this results in a complex entanglement of event occurrences, in propagations, out propagations, and state transitions.

The fact that transitions (and out propagations) are governed by *expressions* rather than singular input symbols evokes the use of *symbolic automata.* Essentially, the core idea is to eval-

[DISTRIBUTION STATEMENT A] Approved for public release and unlimited distribution.

uate the condition to the set of input symbols that make the condition true and to consider the transition/propagation selected when the current input symbol is one of the satisfying symbols. Considering this, defining a denotational semantics for the condition expressions is a prerequisite to giving a definition of a behavior automaton.

1.2.2 Outline: Semantics

As suggested above, condition expressions depend on internal events, incoming error propagations, and error types and type sets. Condition expressions are thus evaluated in a semantic *environment* that contains information about the currently occurring event and in propagations. It is this necessity that frames the presentation of the semantics:

- Section 3 formalizes the EMV2 type system. This includes declared error types, extended types, type aliases, type sets and type aliases, and type products. Error types and type sets play an integral role in declaring propagations and conditions, so they are presented first.
- Section 4 presents the syntactic and semantic models of the AADL model with EMV2 declarations. These are based on a primitive object-oriented representation.
- Section 4 also describes the semantics of the basic EMV2 component declarations error event, error behavior state, and propagation.
- Section 5 describes the environment of a component—to be used to describe the evaluation of condition expressions.
- Sections 6 to 8 describe the semantics of condition expressions. As stated above, these are unusual: instead of operating on an environment and returning a value, a condition evaluates to the set of environments that cause the expression to be satisfied. Expressions are explained in three parts:
  - Section 6 describes the basic logical operators `or` and `and` and the "trigger expressions."
  - Section 7 describes the complex assumptions required when evaluating conditions for transitions. This section shows how to reduce these requirements to the basic logical operators.
  - Section 8 describes the semantics of the "primitive operators" `all`, `ormore`, and `orless` in terms of basic logical operators.
- Finally, Section 9 describes the specifics of the behavior automaton itself. This is fundamentally based on the
  - use of environments as the *input symbols* to the automaton
  - conversion of condition expressions to sets of environments to evaluate transitions and out propagations

1.3 Generation of Fault Trees from EMV2

The de facto canonical fault tree reference *Fault Tree Handbook with Aerospace Applications* describes fault tree analysis (FTA):

> FTA can be simply described as an analytical technique, whereby an undesired state of the system is specified (usually a state that is critical from a safety or reliability standpoint), and the system is then analyzed in the context of its environment and operation to find all realistic ways in which the undesired event (top event) can occur. The fault tree itself is a graphic model of the various parallel

[DISTRIBUTION STATEMENT A] Approved for public release and unlimited distribution.

> and sequential combinations of faults that will result in the occurrence of the predefined undesired event. The faults can be events that are associated with component hardware failures, human errors, software errors, or any other pertinent events which can lead to the undesired event. A fault tree thus depicts the logical interrelationships of basic events that lead to the undesired event, the top event of the fault tree. [34]

Fundamentally there are two steps to fault tree analysis: (1) production of the fault tree and (2) analysis of the fault tree. Typically, fault trees are produced by hand based on processes intended to coerce the producer to determine all the relevant sources of faults and their effects upon the system. The proliferation of formal architecture descriptions has led to techniques to automatically generate fault trees (e.g., [2, 7, 10, 11, 13, 20, 22, 35]).

Analysis of a fault tree is either qualitative or quantitative:

- *Qualitative* evaluation determines the sets or sequences of events that can cause the top event to become true.
- *Qualitative* evaluation determines the probability of the top event becoming true.

Additional summary material on fault trees is presented in Section 2.1. This report does not concern the analysis of a fault tree. This report does, however, contribute a novel semantics-based technique for generating a fault tree from AADL models with EMV2 component error behavior descriptions.

1.3.1 Related Work

Previous work describes techniques for deriving fault trees from AADL models using both the original error model annex [7, 13, 20] and the current EMV2 [10]. These existing techniques are insufficient for our work:

- Some are based on an obsolete annex language [25] that is significantly different from the current version.
- None of the techniques are grounded in a formal error modeling semantics.
- None of the techniques present a comprehensive description of exactly how the fault tree is generated. Indeed, without an underlying semantic model, this is difficult.

Other work describes techniques for generating fault trees from general descriptions of systems [2, 11, 22, 35]. In most cases, this work is not grounded in an architecture description language or a system modeling approach with well-defined semantics, as is AADL. See Section 13.2.2, however, for a discussion of how techniques from these works may influence future work.

Furthermore, the approach presented herein is based on component fault trees [15, 16] and is, therefore, composable and component oriented.

1.3.2 Outline: Fault Tree Generation

The details of fault tree generation are provided in several sections of this report:

- Background on fault tree analysis is presented in Section 2.1.
- Background on component fault trees is presented in Section 2.2.
- A general description of the fault tree generation process is given in Section 10.
- A formal description of how a component fault tree is derived from each AADL component is given in Section 11.

[DISTRIBUTION STATEMENT A] Approved for public release and unlimited distribution.

- A formal description of how a fault tree for a complete system is generated is given in Section 12.

[DISTRIBUTION STATEMENT A] Approved for public release and unlimited distribution.

2 Background

This section provides an overview of the fault tree concepts utilized by the approach described herein. It includes a brief description of both static and dynamic fault trees, the temporal algebra of Merle, and component fault trees. The section concludes with a summary of the mathematical notation and concepts used throughout.

2.1 Fault Trees

Fundamentally, a fault tree is a graphical representation of a Boolean formula expressing the condition under which the top event occurs: the so-called structure function of the top event. The leaf nodes are known as basic events and represent a fault that requires no further development. Intermediate events are represented by the output of a "gate" that permits or inhibits the passage of fault logic. Standard fault trees use AND and OR gates representing logical conjunction and disjunction. Thus, a fault tree can be thought of as the parse tree of a Boolean expression in which the basic events are Boolean variables. This style of fault tree is known as a *static fault tree*: it is unable to capture time-dependent relationships between events.

2.1.1 Dynamic Fault Trees

A *dynamic fault tree* (DFT) is capable of capturing happens-before relationships between events [8]. The priority AND gate, PAND, is true only when (1) both its inputs are true and (2) the first input became true before the second input became true. Specifically, it is not true when the second input becomes true before the first input. The other standard dynamic gates are

- FDEP, functional dependency, which indicates that a set of output events become true when the input event becomes true
- SEQ, sequence enforcing, which forces the input events to occur only in the given order
- SPARE, spare component, which isolates a spare component from failures until it is activated by the failure of one or more previous components

The only dynamic gate used by the fault trees generated by the process described herein is PAND.

Analysis, both quantitative and qualitative, of dynamic fault trees is more complicated and usually reduced to Markov chains. To achieve improved performance, modern analysis tools for DFTs divide the DFT into static and dynamic subgraphs (e.g., Basgöze et al. [3]).

2.1.2 The Structure Function of Dynamic Fault Trees

While static fault trees are easily understood in terms of Boolean algebra, the time-dependent nature of the dynamic gates in DFTs has historically made them difficult to formally reason about. The development of a formal time-based algebra by Merle [19] has greatly improved the situation by allowing a structure function to be produced from a DFT.[1] Various sorts of analyses can then be performed by manipulating the structure function. Important to the

[1] Recently the authors became aware of a similar effort by Schilling [31]. Although discovered too late to have an influence on the work described herein, it is worth noting that Schilling's formalism *does support* negation and thus the analysis of non-coherent [29] fault trees. As a practical matter, however, there unfortunately does not appear to be any tool support.

[DISTRIBUTION STATEMENT A] Approved for public release and unlimited distribution.

$$d(a+b) = \begin{cases} d(a) & \text{if } d(a) < d(b) \\ d(a) & \text{if } d(a) = d(b) \\ d(b) & \text{if } d(a) > d(b) \end{cases} \qquad d(a \cdot b) = \begin{cases} d(b) & \text{if } d(a) < d(b) \\ d(a) & \text{if } d(a) = d(b) \\ d(a) & \text{if } d(a) > d(b) \end{cases}$$

Figure 2.1: Temporal Definitions of the or *(+) and* and *(·) Operators*

work herein, the theorems for the algebra can be used to prove the correctness of various simplifications of the DFT. A brief summary of the algebra is thus presented here.

2.1.2.1 Temporal Functions

The algebra is based on a Boolean model of events, where an event is a *temporal function* in $\mathbb{R}^+ \cup \{+\infty\} \to \mathbb{B}$, where

- $\mathbb{R}^+$ is the set of non-negative real numbers.
- $+\infty$ represents the time point that is unreachable such that
  - $\forall t \in \mathbb{R}^+ . t < +\infty$
  - $+\infty \not< +\infty$
- $\mathbb{B}$ is the set of Boolean values $\{0, 1\}$, or alternatively $\{\mathsf{false}, \mathsf{true}\}$.

Every event a has a single *date of occurrence*, denoted $d(a)$, such that for $0 \leq t < d(a)$, a(t) = 0, and for $t \geq d(a)$, a(t) = 1. An event function can thus be defined solely by its date of occurrence.

Figure 2.1 shows the definitions of the traditional Boolean disjunction and conjunction operators. The logical identities for + and · are $\bot$ and $\top$, respectively, where $d(\bot) = +\infty$ and $d(\top) = 0$. Proofs of the standard logical theorems proceed following the tripartite scheme shown above, that is, considering when $d(a) < d(b)$, when $d(a) = d(b)$, and when $d(a) > d(b)$. The standard commutative, associative, idempotent, distributive, and identity laws are proven. These operators are sufficient for representing static fault trees.

To represent dynamic fault trees, additional time-sensitive operators are required. The (non-inclusive) before operator $\lhd$ indicates if a occurs before b, or if a occurs and b does not occur at all. The simultaneous operator $\triangle$ indicates whether a and b have the same date of occurrence. These operators are defined in Figure 2.2. An important assumption in fault trees (and for EMV2) is that any two basic events a and b cannot occur simultaneously. This can be expressed using the expression $a \triangle b = \bot$. These two operators can be combined to produce the *inclusive before* operator $\trianglelefteq$: $a \trianglelefteq b = a \lhd b + a \triangle b$, or equivalently

$$d(a \trianglelefteq b) = \begin{cases} d(a) & \text{if } d(a) < d(b) \\ d(a) & \text{if } d(a) = d(b) \\ +\infty & \text{if } d(a) > d(b) \end{cases}$$

Specifically, $a \trianglelefteq b$ occurs when a occurs before b, when a occurs at the same time as b, or when a occurs and b does not occur at all. Generally speaking, the inclusive before is preferred to the non-inclusive before during the creation of a structure function from a DFT. It is shown, however, that during reduction of the structure function to a canonical form, the inclusive before operators can be eliminated; the canonical form contains non-inclusive before operators only.

Merle proves a number of basic theorems and simplification rules about the three temporal operators [19]. These are not shown here—those that are necessary for later proofs are provided when needed.

[DISTRIBUTION STATEMENT A] Approved for public release and unlimited distribution.

$$d(a \lhd b) = \begin{cases} d(a) & \text{if } d(a) < d(b) \\ +\infty & \text{if } d(a) = d(b) \\ +\infty & \text{if } d(a) > d(b) \end{cases} \qquad d(a \triangle b) = \begin{cases} +\infty & \text{if } d(a) < d(b) \\ d(a) & \text{if } d(a) = d(b) \\ +\infty & \text{if } d(a) > d(b) \end{cases}$$

Figure 2.2: Temporal Definitions of the Before *(◁) and* Simultaneous *(△) Operators*

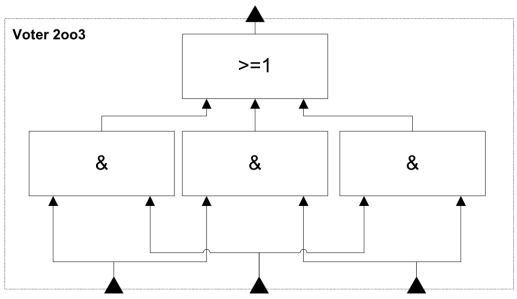

Figure 2.3: A CFT for a 2 out of 3 Voting Component (Source: Kaiser et al. [15].[3])

2.1.2.2 Behavior Model of Dynamic Gates

Most importantly, Merle provides so-called behavior models of dynamic fault tree gates using the temporal algebra [19]. The most important of these for the work described herein is the model of the priority AND (PAND) gate, traditionally defined using language such as "A and B when A is before B." Merle, however, has formally defined the meaning of *before*—the operator $\trianglelefteq$—and thus the PAND gate whose inputs are represented by the structure functions A and B can be represented by the structure function $B \cdot (A \trianglelefteq B)$.

Merle is able to formally prove using the temporal algebra the long-held belief that the functional dependency gate (FDEP) is logically equivalent to the OR gate.[2]

Merle relies on Boudali and associates [4] to conclude that sequence (SEQ) gates can be replaced with cold spare (SPARE) gates and therefore provides no formal modeling of the SEQ gate. This conclusion is also made by Pai and Dugan [21]. It is, however, *incorrect* as shown by Junges and colleagues [14].

2.2 Component Fault Trees

The process to generate a fault tree from an AADL and EMV2 model described herein is based on using *component fault trees* (CFTs) [15, 16]. As the name implies, a CFT brings hierarchical modules to the structure of a fault tree. Generally speaking, a CFT is a directed graph built from fault tree gates and specialized nodes known as ports. An out port represents a logical flow out of the CFT, and an in port represents a logical flow into a CFT. An example CFT with three in ports and one out port that abstracts a 2-of-3 gate is shown in Figure 2.3. Internal to a CFT are standard fault tree gates and events. Every event is internal to some CFT; events internal to one component are stochastically independent from those that are internal to another component. Additionally, a CFT may contain another CFT in the same way that a fault tree gate node is used: subgraphs flow into the in ports of the nested CFT, and the out ports of the nested CFT can flow into multiple parent nodes. Pragmatically speaking, each out port of a CFT is a Boolean function (i.e., a *predicate*) parameterized by the CFT's in ports.

[2] The authors believe, however, that there is value in maintaining a distinction between the FDEP and OR gates during specification or construction of the fault tree because they represent fundamentally different design intents.

[DISTRIBUTION STATEMENT A] Approved for public release and unlimited distribution.

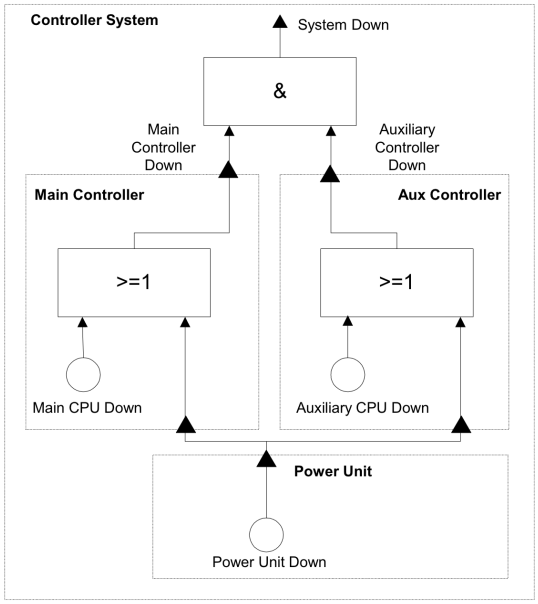

Figure 2.4: A CFT with Three Sub-CFTs. "Main Controller" and "Aux Controller" are two different instances of the same component type (Source: Kaiser et al. [15].[3])

By connecting together the in and out ports of different CFTs, a component fault tree for the entire system is produced. Each component type in a model is associated with a CFT that abstracts its fault behavior. CFTs are connected based on the architectural structure of the system. If a particular component type appears multiple times in the model, multiple copies (instances) of the associated CFT are introduced into the final fault tree. The CFT in Figure 2.4 is constructed from three CFTs and an AND gate; in particular "Main Controller" and "Aux Controller" are two instances of the *same* CFT component.

In summary, CFTs

- mirror and exploit hierarchical modularization in system architectures. This makes them particularly well suited for analyzing AADL models.
- enable *reusable* fault tree modules

The overall vision is that a component should ship with a CFT that describes its behavior, and the system integrator would incorporate the CFTs of all the system components into the CFT of the system as a whole. Unfortunately, as is described in Section 10.4.2, this is not achievable for AADL models, although a different sort of reuse is presented.

2.2.1 Obtaining a CEG from a CFT

A traditional *cause–effect graph* (CEG, fault tree as a graph) can be derived from any out port of any CFT; the chosen port is known as the *root port*. A depth-first search (e.g., Aho et al. [1, §6.5]) from the out port into the CFT is used to construct the fault tree. The search recursively proceeds into any sub-CFT it encounters. A particular CFT may be visited multiple times as the search enters through different output ports. A CFT is exited as the search travels through an input port, where the search then continues in the internals of the surrounding

[3]Copyright © 2003, Australian Computer Society, Inc. This paper appeared at the 8th Australian Workshop on Safety Critical Systems and Software (SCS'03), Canberra. Conferences in Research and Practice in Information Technology, Vol. 33. P. Lindsay & T. Cant, Eds.

[DISTRIBUTION STATEMENT A] Approved for public release and unlimited distribution.

CFT. The CEG includes the event and gate nodes only; ports are elided and, as previously described, nested CFTs are traversed into.

2.2.2 Formalism

A CFT is formalized as a 4-tuple by Kaiser and colleagues [15]. The formalism is presented here with a some notational modifications to make it consistent with the rest of the notation used herein. A CFT is a 4-tuple (N, G, S, E), where

- N is a set of *simple nodes* partitioned into the disjoint subsets N_{Intern} of internal events, N_{In} of input ports, and N_{Out} of output ports.
- G is the set of gates, where each $g \in G$ has a single output port $g.\mathsf{out}$, one or more input ports $g.\mathsf{in}_i$ where $i \in \mathbb{N}$, and a Boolean formula, such as $\mathsf{g.out} = \mathsf{g.in}_1 \wedge \mathsf{g.in}_2$, where g is a specific gate in G. Set $G_{\mathsf{In}} = \bigcup_{g \in G} \{g.\mathsf{in}_i\}$ and set $G_{\mathsf{Out}} = \bigcup_{g \in G} \{g.\mathsf{out}\}$.
- S is the set of subcomponents (i.e., nested CFTs). Each $s \in S$ has one or more output ports $s.\mathsf{out_i}$, one or more input ports $s.\mathsf{in_j}$, and a mapping between the subcomponent's ports and those of the component's associated CFT. Set $S_{\mathsf{In}} = \bigcup_{s \in S} \{s.\mathsf{in_i}\}$ and set $S_{\mathsf{Out}} = \bigcup_{s \in S} \{s.\mathsf{out_i}\}$.
- $E \subseteq (N_{\mathsf{Intern}} \cup N_{\mathsf{In}} \cup G_{\mathsf{Out}} \cup S_{\mathsf{Out}}) \times (N_{\mathsf{Out}} \cup G_{\mathsf{In}} \cup S_{\mathsf{In}})$ is the set of edges: for $(s, d) \in E$, s is the source of the edge and d is the destination of the edge. That is, edges point towards the output ports. No two edges may share the same target: $\nexists s_1, s_2, d. (s_1, d) \in E \wedge (s_2, d) \in E \wedge s_1 \neq s_2$. It is forbidden for edges to form a cycle; therefore, an output port cannot be connected to an input port of the same subcomponent.

It must be emphasized that the g and s are *instances* of specific types of gates and components, respectively. The mapping between a component instance s and its fault tree is not further described by Kaiser and colleagues [15] but can easily be imagined. For reasons described in Section 11, the CFTs developed in this work have a one-to-one mapping to components, and thus such a mapping is unnecessary.

2.3 Notation and Definitions

The notation and terminology used throughout are heavily influenced by Hopcroft and Ullman [12].

2.3.1 Conventions

As is standard, pure mathematical sets are written in "blackboard bold":

- $\mathbb{N}$ is the set of natural numbers: $\{0, 1, 2, 3, \ldots\}$.
- $\mathbb{R}$ is the set of real numbers.
- $\mathbb{B}$ is the set of Boolean values: $\{\mathsf{true}, \mathsf{false}\}$.

When identifying items in the semantic space, sets are written in calligraphy (e.g., $\mathcal{A}$ is the set of all symbols). Specific symbols and other mathematical relations are written in sans serif. Semantic functions are generally written using bracket notation (e.g., $[\![\cdot]\!]_{\mathsf{F}}$).

The names of sets and functions relating to the abstract syntactic space are written in **boldface**. Declared names and other items representing syntax are written in `typewriter text`.

[DISTRIBUTION STATEMENT A] Approved for public release and unlimited distribution.

2.3.2 Symbols

The definition from Hopcroft and Ullman is used here:

> A "symbol" is an abstract entity that we shall not define formally, just as "point" and "line" are not defined in geometry. Letters and digits are examples of frequently used symbols. [12]

The set of all symbols is herein denoted $\mathcal{A}$.

2.3.3 Relations

For a relation $X : A \to B$, equivalently $X : A \times B$,

- $\mathsf{dom}\ X = \{a \mid (a, b) \in X\}$
- $\mathsf{ran}\ X = \{b \mid (a, b) \in X\}$
- $(a, b) \in X \Leftrightarrow a\ X\ b \Leftrightarrow X(a) = b$
- $\mathsf{defined}\ X(a) = \exists b \in B.\, (a, b) \in X$
- $\mathsf{undefined}\ X(a) = \forall (a', b') \in X.\, a' \neq a$

2.3.4 Cross Products

Here $\prod$ is used with sets to mean the repeated application of the cross product $\times$: when $S_1, \ldots, S_n$ are all sets

$$\prod_{i=1}^{n} S_i = S_1 \times \cdots \times S_n$$

2.3.5 Uniquely Define

When

- S and T are sets
- $f : S \to T$ is a function

then f *uniquely maps* S to T if and only if

$$\forall s, s' \in S.\, f(s) = t \wedge f(s') = t \Rightarrow s = s'$$

When $|S| = |T|$, the function f is said to *uniquely define* set T.

2.3.6 List Notation

The set of lists over a set S is $\mathsf{list}(S)$. An element $l \in \mathsf{list}(S)$ can be written $l = [e_1, \ldots, e_n]$ and is a finite ordered sequence such that $e_i \in S$.

- The number of elements of l is written $|l| = n$.
- e_1 is the first element of l and may be written $\mathsf{first}(l)$.
- e_n is the last element of l and may be written $\mathsf{last}(l)$.

A sublist $[s_1, \ldots, s_m] \in l \Leftrightarrow \exists i \geq 1. \forall 0 \leq j < m.\, e_{i+j} = s_{1+j}$.

[DISTRIBUTION STATEMENT A] Approved for public release and unlimited distribution.

2.4 Record Notation

Record notation is used to compactly refer to values in tuples, for example,

$$\langle f_1 \mapsto v_1, \ldots, f_n \mapsto v_n \rangle$$

represents a tuple where the position representing field f_i is set to value v_i, and any field not explicitly named is set to an empty value (described below). The field names f_i are simply distinct *symbols* from an index set. Specifically, a set of n-records is described by the structure (R, I, E, ϵ), where

- R is the underlying set of n-tuples.[4]
- $I \subseteq \mathcal{A}$ is the index set:
  - $|I| = n$
  - Values for field $f \in I$ must come from the set R_f. Therefore, set R is isomorphic to the set $\prod_{f \in I} R_f$.
- E is the set of values that represent empty fields: $|E| \leq n$ because fields are allowed to share empty values.
- $\epsilon : I \to E$ maps each field to the empty value for that field; it must be that $\epsilon(f) \in R_f$.

As suggested above, the order of the values in the tuple is arbitrary, as long as a consistent mapping from field names to tuple positions is maintained. This level of detail is not elaborated here. The upshot is that the order of sets used in the cross product to describe R is also arbitrary. Here a "labeled" cross product notation is used to associate indices with sets:

$$R = f_1 : S_1 \times \cdots \times f_n : S_n$$

This defines a set R of n-tuples with indices $\{f_1, \ldots, f_n\}$ where $R_{f_i} = S_i$. With this notation, R and I are simultaneously implicitly defined, and the structure of the record is instead the triple

$$(f_1 : S_1 \times \cdots \times f_n : S_n, E, \epsilon)$$

Array-style notation is used to manipulate a record. For $r \in R, f \in I, v \in R_f$,

- $r[f] \in R_f$ is the value of the f field of r.
- $r[f] \leftarrow v$ is the tuple $r' \in R$ such that $r'[f] = v$ and $\forall f' \in I.\, f' \neq f \Rightarrow r'[f'] = r[f']$.

The above notation $\langle f_1 \mapsto v_1, \ldots, f_m \mapsto v_m \rangle$ can now be formally defined as representing the n-tuple $r \in R$ such that

- $m \leq n$
- $\forall i, j.\, i \neq j \Rightarrow f_i \neq f_j$, that is, fields names cannot be repeated
- $r[f_i] = v_i$
- $\forall f \in I \setminus \{f_1, \ldots, f_n\}.\, r[f] = \epsilon(f)$

Note that $\langle\rangle$ is the record where all the fields are set to the appropriate empty value: $\forall f \in I.\, \langle\rangle[f] = \epsilon(f)$.

A record r can also be considered a function $r : I \to \bigcup_{f \in I} R_f$:

$$r(f) = v \Leftrightarrow r[f] = v \Leftrightarrow \langle \ldots, f \mapsto v, \ldots \rangle \Leftrightarrow (f, v) \in r$$

[4] R for **R**ecords.

[DISTRIBUTION STATEMENT A] Approved for public release and unlimited distribution.

Similarly, a function $g : I \to \bigcup_{f \in I} R_f$ can be converted to a record $\mathsf{record}(g)$ such that

$$\forall f \in I. \begin{cases} f \in \mathsf{dom}\ g & \Rightarrow \quad \mathsf{record}(g)[f] = g(f) \\ f \notin \mathsf{dom}\ g & \Rightarrow \quad \mathsf{record}(g)[f] = \epsilon(f)) \end{cases}$$

2.4.1 Trees

A tree (V, E, r) is a structure of nodes and directed edges with a distinguished root node. All nodes are reachable from the root node by following edges. Specifically

- Set V is the set of *nodes* or *vertices* in the tree.
- Relation $E \subseteq V \times V$ is the set of *edges* in the tree. When $(v_1, v_2) \in E$, there is an edge from node v_1 to node v_2. Vertex v_1 is a *parent* of vertex v_2, which is a *child* of v_1.
  - No node is a child of itself: $\forall v \in V. (v, v) \notin E$.
  - A node has at most one parent: $\forall v \in V.\ \nexists p_1, p_2 \in V. (p_1, v) \in E \wedge (p_2, v) \in E$.
- $r \in V$ is the root node of the tree. It is the child of no vertex, and all other nodes are reachable from it.
  - $\nexists v \in V. (v, r) \in E$
  - $\forall v \in V. \mathsf{reachable}(r, v)$, where $\mathsf{reachable}(n, \hat{n}) \Leftrightarrow n = \hat{n} \vee \exists n' \in V. (n\ E\ n' \wedge \mathsf{reachable}(n', \hat{n}))$

[DISTRIBUTION STATEMENT A] Approved for public release and unlimited distribution.

3 Error Types

Error types are declared in AADL packages and are universal for all components. Named error types, as well as predeclared error type sets, can be declared in these so-called error type libraries. In addition, names that alias error types and error type sets can be declared. $\mathbf{N}$ is the set of syntactic names and is based on the definition of names in the AADL specification[1] [26, §15.3]. Its meaning should be clear, and the set is not discussed further except to say it is a flat set: package name prefixes are incorporated into the names in $\mathbf{N}$.

3.1 Declared Error Types

Error library declarations belong to the syntactic set $\mathbf{L}$ and are described by the abstract production rule **LibElement**:

$$
\begin{array}{rcl}
\textbf{LibElement} & ::= & \mathbf{N} \texttt{ : type ;} \\
& | & \mathbf{N} \texttt{ : type extends } \mathbf{N} \texttt{ ;} \\
& | & \mathbf{N} \texttt{ renames type } \mathbf{N} \texttt{ ;} \\
& | & \mathbf{N} \texttt{ : type set } \mathbf{TypeSet} \texttt{ ;} \\
& | & \mathbf{N} \texttt{ renames type set } \mathbf{N} \texttt{ ;}
\end{array}
$$

The above rules make use of the production rule for type sets, where $\mathbf{S} : \mathbf{TypeSet}$. This production is described in a Section 3.3. Type libraries are unified into the simple set $\mathbf{D}_{\mathsf{Lib}} \subseteq 2^{\mathbf{L}}$—recall names are assumed to be prefixed with package names. A number of syntactic sets, shown in Figure 3.1, are derived from the declared library elements:

- $\mathbf{D}_{\mathsf{Type}} \subseteq \mathbf{N}$ is the set of declared error type names.
- $\mathbf{D}_{\mathsf{extends}} \subseteq \mathbf{N} \times \mathbf{N}$ is the set of declared error type extensions.
- $\mathbf{D}_{\mathsf{Set}} \subseteq \mathbf{N} \times \mathbf{S}$ is the set of declared error type sets.
- $\mathbf{D}_{\mathsf{Renames}} \subseteq \mathbf{N} \times \mathbf{N}$ is the set of declared error type aliases.
- $\mathbf{D}_{\mathsf{Renames\_TS}} \subseteq \mathbf{N} \times \mathbf{N}$ is the set of declared type set aliases.

Declared error types, type sets, type aliases, and type set aliases all share the same namespace [27, E.5.(N20), E.5.(N21)]. Therefore, the intersections of all pairwise combinations of $\mathbf{D}_{\mathsf{Type}}$ and $\mathsf{dom}\ \mathbf{D}_{\mathsf{Extends}}$, $\mathsf{dom}\ \mathbf{D}_{\mathsf{Set}}$, $\mathsf{dom}\ \mathbf{D}_{\mathsf{Renames}}$, and $\mathsf{dom}\ \mathbf{D}_{\mathsf{Renames\_TS}}$ are the empty set $\emptyset$.

3.1.1 Semantic Error Types

Set $\mathcal{E} \subseteq \mathcal{A}$ is a set of symbols representing the declared error types. The semantic function $[\![\cdot]\!]_{\mathsf{Type}} : \mathbf{D}_{\mathsf{Type}} \to \mathcal{E}$ uniquely defines $\mathcal{E}$. Herein it is assumed that the name `x` maps to the symbol x. Names from a `renames` declaration do not specify symbols in the semantic space; see below. Declarations in the EMV2 model establish *subtype* as an irreflexive function $\prec: \mathcal{E} \times \mathcal{E}$ [27, E.5 (6)]:

- $e_1 \prec e_2$ iff e_1 is (declared to be) a subtype of e_2.
- $(e, e) \notin \prec$

Specifically,

$$(n, n') \in \mathbf{D}_{\mathsf{Extends}} \Leftrightarrow [\![n]\!]_{\mathsf{Type}} \prec [\![n']\!]_{\mathsf{Type}}$$

[1] Cf. the Java class `String`.

[DISTRIBUTION STATEMENT A] Approved for public release and unlimited distribution.

$$
\begin{array}{rcl}
\mathbf{D}_{\mathsf{Type}} & = & \{n \mid n : \texttt{ type } ; \in \mathbf{D}_{\mathsf{Llb}}\} \cup \{n \mid n : \texttt{ type extends } n' ; \in \mathbf{D}_{\mathsf{Llb}}\} \\
\mathbf{D}_{\mathsf{Extends}} & = & \{(n, n') \mid n : \texttt{ type extends } n' ; \in \mathbf{D}_{\mathsf{Lib}}\} \\
\mathbf{D}_{\mathsf{Set}} & = & \{(n, s) \mid n : \texttt{ type set } s ; \in \mathbf{D}_{\mathsf{Lib}}\} \\
\mathbf{D}_{\mathsf{Renames}} & = & \{(n, n') \mid n \texttt{ renames type } n' ; \in \mathbf{D}_{\mathsf{Lib}}\} \\
\mathbf{D}_{\mathsf{Renames\_TS}} & = & \{(n, n') \mid n \texttt{ renames type set } n' ; \in \mathbf{D}_{\mathsf{Lib}}\}
\end{array}
$$

Figure 3.1: Sets of Type Declarations Derived from $\mathbf{D}_{\mathsf{Lib}}$

3.1.2 Type References

As a first step to interpreting type aliases, the set of all *type references* $\mathbf{R}_{\mathsf{Type}}$ is defined:

$$
\begin{array}{rcl}
\mathbf{R}_{\mathsf{Type}} & = & \mathbf{D}_{\mathsf{Type}} \cup \mathsf{dom}\ \mathbf{D}_{\mathsf{Renames}} \\
\mathbf{R}_{\mathsf{TSE}} & = & \mathbf{R}_{\mathsf{Type}} \cup \mathsf{dom}\ \mathbf{D}_{\mathsf{Set}} \cup \mathsf{dom}\ \mathbf{D}_{\mathsf{Renames\_TS}}
\end{array}
$$

The second set of "type set element" references includes the names of type sets and type set aliases and is used in a later section. The semantic function $[\![\cdot]\!]_{\mathsf{Ref}} : \mathbf{R}_{\mathsf{Type}} \to \mathcal{E}$ evaluates syntactic references to error types and type aliases to semantic error symbols:

$$
[\![n]\!]_{\mathsf{Ref}} = \begin{cases} [\![n]\!]_{\mathsf{Type}} & \text{when } n \in \mathbf{D}_{\mathsf{Type}} \\ [\![n']\!]_{\mathsf{Ref}} & \text{when } (n, n') \in \mathbf{D}_{\mathsf{Renames}} \end{cases}
$$

That is $(n, n') \in \mathbf{D}_{\mathsf{Renames}} \Rightarrow [\![n]\!]_{\mathsf{Ref}} = [\![n']\!]_{\mathsf{Ref}}$, cf. [27, E.5.(9), E.5.(24)]. Note that the aliased name n' is evaluated as a *reference* using $[\![\cdot]\!]_{\mathsf{Ref}}$ because aliases are allowed to be aliased. Also note that function $[\![\cdot]\!]_{\mathsf{Ref}}$ specifically *does not uniquely define* $\mathcal{E}$, which is, of course, the whole point of aliases.

References can also be resolved in the syntactic space using $\mathbf{Res} : \mathbf{R}_{\mathsf{Type}} \to \mathbf{D}_{\mathsf{Type}}$:

$$
\mathbf{Res}(n) = \begin{cases} n & \text{when } n \in \mathbf{D}_{\mathsf{Type}} \\ \mathbf{Res}(n') & \text{when } (n, n') \in \mathbf{D}_{\mathsf{Renames}} \end{cases}
$$

3.1.3 Type Containment

The relations $<: \mathcal{E} \times \mathcal{E}$ and $\leq: \mathcal{E} \times \mathcal{E}$ extend the subtype relation $\prec$ [27, E.5 (28)]:

$$
\begin{array}{rcl}
e < e' & \Leftrightarrow & (e \prec e') \ \vee\ \exists \hat{e}.\, (e \prec \hat{e} \wedge \hat{e} < e') \\
e \leq e' & \Leftrightarrow & (e = e') \ \vee\ (e < e')
\end{array}
$$

The *root* of a declared error type e is given by $\mathsf{r} : \mathcal{E} \to \mathcal{E}$ and is the type that satisfies the property [27, E.5 (29)]:

$$
(e \leq \mathsf{r}(e)) \ \wedge\ (\nexists e'.\, \mathsf{r}(e) < e')
$$

The *type containment* relation $\sqsubseteq$, more fully defined in Section 3.4, extends the type descendent relation to type products and type sets. For declared error types $e_1, e_2 \in \mathcal{E}$, it is simply [27, E.5 (30)]

$$
e_1 \sqsubseteq e_2 \Leftrightarrow e_1 \leq e_2
$$

3.1.4 Extension and Containment in the Syntactic Space

Because

- the semantic set $\mathcal{E}$ is uniquely defined by $[\![\cdot]\!]_{\mathsf{Type}}$ from the set $\mathbf{D}_{\mathsf{Type}}$
- the definition of the semantic $\prec$ relation is in terms of $\mathbf{D}_{\mathsf{Extends}}$

[DISTRIBUTION STATEMENT A] Approved for public release and unlimited distribution.

the *semantic* notion of $<$ can be directly related to the *syntactic* space:

$$e_1 < e_2 \Leftrightarrow \exists\{(d_1, d_1'), \ldots, (d_n, d_n')\} \subseteq \mathbf{D}_{\mathsf{Extends}} \cdot [\![d_1]\!]_{\mathsf{Type}} = e_1 \wedge [\![d_n']\!]_{\mathsf{Type}} = e_2 \wedge d_i' = d_{i+1}$$

That is, a chain of declared type extensions exists in the syntax between the type names denoted by e_1 and e_2. This syntactic relationship is identified with the relation **descendant** : $\mathbf{D}_{\mathsf{Type}} \times \mathbf{D}_{\mathsf{Type}}$, i.e., $[\![n_1]\!]_{\mathsf{Type}} < [\![n_2]\!]_{\mathsf{Type}} \Leftrightarrow n_1$ **descendant** n_2, from which the relation **extends** : $\mathbf{D}_{\mathsf{Type}} \times \mathbf{D}_{\mathsf{Type}}$ can be defined:

$$\begin{array}{rcl} [\![n_1]\!]_{\mathsf{Type}} \leq [\![n_2]\!]_{\mathsf{Type}} & \Leftrightarrow & n_1 \textbf{ extends } n_2 \\ n_1 \textbf{ extends } n_2 & \Leftrightarrow & n_1 = n_2 \vee n_1 \textbf{ descendant } n_2 \end{array}$$

Finally, because semantic type containment is based on $\leq$, i.e., $e_1 \sqsubseteq e_2 \Leftrightarrow e_1 \leq e_2$, it too can be related to the syntactic space:

$$[\![n_1]\!]_{\mathsf{Type}} \sqsubseteq [\![n_2]\!]_{\mathsf{Type}} \Leftrightarrow n_1 \textbf{ extends } n_2 \Leftrightarrow n_1 \sqsubseteq n_2$$

This bidirectionally of containment between the semantic and syntactic space may seem pedantic, but it is useful for showing *what operations can be performed by an analysis tool solely based on the syntax of the model*. For example, negating a trigger of a condition expression is discussed in Section 8.3.

3.2 Type Products

Error type products belong to set $\mathbf{P}$ and are built using the abstract production rule **TypeProduct**:

$$\begin{array}{rcl} \textbf{TypeProduct} & ::= & \mathbf{R}_{\mathsf{Type}} * \mathbf{R}_{\mathsf{Type}} \\ & | & \mathbf{R}_{\mathsf{Type}} * \textbf{TypeProduct} \end{array}$$

The rule makes use of the set of declared type names $\mathbf{R}_{\mathsf{Type}}$ to indicate that the names used in a type product must be names of error types or error type aliases and *not* type sets.

$\mathcal{P}$ is the set of all semantic type products; it is more specifically defined below. The semantic function $[\![\cdot]\!]_{\mathsf{Product}} : \mathbf{P} \rightarrow \mathcal{P}$ generates tuples of error type symbols from syntactic products of type names:

$$\begin{array}{rcl} [\![n_1 * n_2]\!]_{\mathsf{Product}} & = & ([\![n_1]\!]_{\mathsf{Ref}}, [\![n_2]\!]_{\mathsf{Ref}}) \\ [\![n * p]\!]_{\mathsf{Product}} & = & ([\![n]\!]_{\mathsf{Ref}}, e_2, \ldots, e_n) \text{ where } (e_2, \ldots, e_n) = [\![p]\!]_{\mathsf{Product}} \end{array}$$

Note that names are evaluated to error type symbols using the *reference* semantics to account for type aliases. Alternatively, type aliases can be handled in the syntactic space first. The syntactic function **Product** : $\mathbf{P} \rightarrow \mathbf{P}$ converts an arbitrary syntactic type product to a type product whose type name references are fully resolved.

$$\begin{array}{rcl} [\![p \in \mathbf{P}]\!]_{\mathsf{Product}} & = & [\![\textbf{Product}(p)]\!]_{\mathsf{Product2}} \\ \textbf{Product}(n_1 * n_2) & = & \textbf{Res}(n_1) \parallel * \parallel \textbf{Res}(n_2) \\ \textbf{Product}(n * p) & = & \textbf{Res}(n) \parallel * \parallel \textbf{Product}(p) \\ [\![n_1 * n_2]\!]_{\mathsf{Product2}} & = & ([\![n_1]\!]_{\mathsf{Type}}, [\![n_2]\!]_{\mathsf{Type}}) \\ [\![n * p]\!]_{\mathsf{Product2}} & = & ([\![n]\!]_{\mathsf{Type}}, e_2, \ldots, e_n) \text{ where } (e_2, \ldots, e_n) = [\![p]\!]_{\mathsf{Product2}} \end{array}$$

Here $\parallel$ is the string concatenation operator, and the function $[\![\cdot]\!]_{\mathsf{Product2}}$ is like $[\![\cdot]\!]_{\mathsf{Product}}$ from above, but the syntactic type names are directly converted into semantic types using $[\![\cdot]\!]_{\mathsf{Type}}$ instead of being interpreted as aliases. This alternative formulation is important for considering type product containment in the syntactic space (see below).

[DISTRIBUTION STATEMENT A] Approved for public release and unlimited distribution.

$$
\begin{array}{rcl}
 & & \mathbf{S} : \mathbf{TypeSet} \\
 & & \mathbf{S}_{\mathsf{Elements}} : \mathbf{TypeSetElements} \\
 & & \mathbf{S}_{\mathsf{Element}} : \mathbf{TypeSetElement} \\
\mathbf{TypeSet} & ::= & \{\ \mathbf{TypeSetElements}\ \} \\
\mathbf{TypeSetElements} & ::= & \mathbf{TypeSetElement} \\
 & | & \mathbf{TypeSetElement}\ ,\ \mathbf{TypeSetElements} \\
\mathbf{TypeSetElement} & ::= & \mathbf{R}_{\mathsf{TSE}} \\
 & | & \mathbf{TypeProduct}
\end{array}
$$

Figure 3.2: Definition of EMV2 Type Sets

The set of products of n types is $\mathcal{P}_n = \overbrace{\mathcal{E} \times \cdots \times \mathcal{E}}^{n \geq 2}$. Now obviously any type product constructed in an AADL model will be of finite size, and there is a least $m \geq 2$ such that all constructed type products have m or fewer elements. Furthermore, EMV2 requires that all elements of a type product be descended from different root types [27, E.5 (8)]. This is expressed by the predicate product:

$$\mathsf{product}((e_1, \ldots, e_n)) \Leftrightarrow \forall i, j \leq n.\, i \neq j \Rightarrow \mathsf{r}(e_i) \neq \mathsf{r}(e_j)$$

Finally, it can be declared that the set of all product types is

$$\mathcal{P} = \left\{ p \in \bigcup_{n=2}^{m} \mathcal{P}_n \mid \mathsf{product}(p) \right\}$$

3.3 Type Sets

An error type set is a set of one or more error types or type products. Set $\mathcal{T} = \mathcal{E} \cup \mathcal{P}$ is the set of semantic type set elements (i.e., types that may be part of a type set). As seen above, error type sets are built using the abstract production rule **TypeSet** defined in Figure 3.2. When a type set element is a name, it is allowed to reference a declared error type, a type alias, a declared type set, or a type set alias. A type set should not be declared to contain itself, and cycles of type set references must be avoided. Herein it is assumed that these cases are prevented prior to semantic analysis.

$\mathcal{S}$ is the set of all type sets; it is defined more specifically below. A second type reference semantic function $[\![\cdot]\!]_{\mathsf{TSE}} : \mathbf{R}_{\mathsf{TSE}} \to \mathcal{S}$ interprets references to type sets and their aliases. Type set aliases are handled by the same clause that handles error type aliases; see also EMV2 [27, E.5.(9), E.5.(25)]. Note that this function *always* evaluates to a set, whereas $[\![\cdot]\!]_{\mathsf{Ref}}$ evaluates to a semantic error symbol. (The function $[\![\cdot]\!]_{\mathsf{Set}}$ is defined below.)

$$[\![n]\!]_{\mathsf{TSE}} = \begin{cases} \{[\![n]\!]_{\mathsf{Type}}\} & \text{when } n \in \mathbf{D}_{\mathsf{Type}} \\ [\![ts]\!]_{\mathsf{Set}} & \text{when } (n, ts) \in \mathbf{D}_{\mathsf{Set}} \\ [\![n']\!]_{\mathsf{TSE}} & \text{when } (n, n') \in \mathbf{D}_{\mathsf{Renames}} \cup \mathbf{D}_{\mathsf{Renames\_TS}} \end{cases}$$

The semantic functions $[\![\cdot]\!]_{\mathsf{Set}} : \mathbf{S} \to \mathcal{S}$, $[\![\cdot]\!]_{\mathsf{Elements}} : \mathbf{S}_{\mathsf{Elements}} \to \mathcal{S}$, and $[\![\cdot]\!]_{\mathsf{Element}} : \mathbf{S}_{\mathsf{Element}} \to \mathcal{S}$ are used to build sets of error type symbols and tuples from syntactic type sets:

$$
\begin{array}{rcl}
[\![\{\ elements\ \}]\!]_{\mathsf{Set}} & = & [\![elements]\!]_{\mathsf{Elements}} \\
[\![t]\!]_{\mathsf{Elements}} & = & [\![t]\!]_{\mathsf{Element}} \\
[\![t\ ,\ elements]\!]_{\mathsf{Elements}} & = & [\![t]\!]_{\mathsf{Elements}} \cup [\![elements]\!]_{\mathsf{Elements}} \\
[\![n \in \mathbf{N}]\!]_{\mathsf{Element}} & = & [\![n]\!]_{\mathsf{TSE}} \\
[\![p \in \mathbf{P}]\!]_{\mathsf{Element}} & = & \{[\![p]\!]_{\mathsf{Product}}\}
\end{array}
$$

[DISTRIBUTION STATEMENT A] Approved for public release and unlimited distribution.

As with type products, this process can be arranged so that all the alias resolution occurs in the syntactic space. The syntactic function $\mathbf{Set} : \mathbf{S} \to \mathbf{S}$, defined in the next paragraph, converts an arbitrary syntactic type set to a syntactic type set where all the names have been resolved, including those nested in referenced type sets and those in member type products.

$$\begin{array}{rcl}
[\![s \in \mathbf{S}]\!]_{\mathsf{Set}} & = & [\![elements]\!]_{\mathsf{Elements}} \\
 & \text{where} & \{\ elements\ \} = \mathbf{Set}(s) \\
[\![t]\!]_{\mathsf{Elements}} & = & [\![t]\!]_{\mathsf{Element}} \\
[\![t\ ,\ elements]\!]_{\mathsf{Elements}} & = & [\![t]\!]_{\mathsf{Elements}} \cup [\![elements]\!]_{\mathsf{Elements}} \\
[\![n \in \mathbf{D}_{\mathsf{Type}}]\!]_{\mathsf{Element}} & = & \{[\![n]\!]_{\mathsf{Type}}\} \\
[\![p \in \mathbf{P}]\!]_{\mathsf{Element}} & = & \{[\![p]\!]_{\mathsf{Product2}}\}
\end{array}$$

The definition of **Set** depends on $\mathbf{resolve} : \mathbf{S}_{\mathsf{Elements}} \to \mathbf{S}_{\mathsf{Elements}}$ and $\mathbf{tse} : \mathbf{S}_{\mathsf{Element}} \to \mathbf{S}_{\mathsf{Elements}}$:

$$\begin{array}{rcl}
\mathbf{Set}(\{elements\ \}) & = & \{\ \|\ \mathbf{resolve}(elements)\ \|\ \} \\
\mathbf{resolve}(n) & = & \mathbf{tse}(n) \\
\mathbf{resolve}(n\ ,\ elements) & = & \mathbf{resolve}(n)\ \|\ ,\ \|\ \mathbf{resolve}(elements) \\
\mathbf{tse}(n \in \mathbf{D}_{\mathsf{Type}}) & = & n \\
\mathbf{tse}(n \in \mathrm{dom}\ \mathbf{D}_{\mathsf{Renames}}) & = & \mathbf{tse}(\mathbf{D}_{\mathsf{Renames}}(n)) \\
\mathbf{tse}(n \in \mathrm{dom}\ \mathbf{D}_{\mathsf{Set}}) & = & n_1, \ldots, n_k \text{ where } \{\ n_1, \ldots, n_k\ \} = \mathbf{Set}(\mathbf{D}_{\mathsf{Set}}(n)) \\
\mathbf{tse}(n \in \mathrm{dom}\ \mathbf{D}_{\mathsf{Renames_TS}}) & = & \mathbf{tse}(\mathbf{D}_{\mathsf{Renames_TS}}(n)) \\
\mathbf{tse}(p \in \mathbf{P}) & = & \mathbf{Product}(p)
\end{array}$$

The point of this alternative presentation is to be able to argue that all type sets $s \in \mathcal{S}$ originate from type sets $\{t_1,\ \ldots,\ t_n\} \in \mathbf{S}$ in the syntactic space where $\forall i.\, t_i \in \mathbf{D}_{\mathsf{Type}} \vee (t_i = t'_1 * \cdots * t'_n,$ where $\forall j.\, t'_j \in \mathbf{D}_{\mathsf{Type}})$.

Like type products, type sets must satisfy a predicate as well, in this case typeset. This predicate is defined in the next section, but we can use it now to define the set of all type sets:

$$\mathcal{S} = \left\{ s \in 2^{\mathcal{T}} \mid \mathsf{typeset}(s) \right\}$$

3.4 Type Containment—Concluded

Type containment $\sqsubseteq$ is extended to compare two type products with the same number of elements [27, E.5 (32)]:

$$(e_1, \ldots, e_n) \sqsubseteq (e'_1, \ldots, e'_n) \Leftrightarrow \forall i.\, \exists j.\, e_i \leq e'_j$$

Observe that the order of the elements in the products *does not matter*, hence the use of $\exists j$; see also EMV2 [27, E.5 (18)].

Next, containment is extended to type set elements versus type sets [27, E.5 (33)]. For $e \in \mathcal{E}$, $p \in \mathcal{P}$, and $s \in \mathcal{S}$,

$$\begin{array}{rcl}
e \sqsubseteq s & \Leftrightarrow & \exists e' \in s.\, e \sqsubseteq e' \\
p \sqsubseteq s & \Leftrightarrow & \exists p' \in s.\, p \sqsubseteq p'
\end{array}$$

Finally, containment between two type sets is definable [27, E.5 (35)]. For $s, s' \in \mathcal{S}$,

$$s \sqsubseteq s' \ \Leftrightarrow\ \forall t \in s.\, \exists t' \in s'.\, t \sqsubseteq t'$$

A type set is required to contain unique elements only [27, E.5 (35)]. That is, no element should be contained in any other element. This is captured by the predicate typeset—referenced in the previous section—finally defined here. For $s \in \mathcal{S}$,

$$\mathsf{typeset}(s) \ \Leftrightarrow\ \forall t, t' \in s.\, t \not\sqsubset t' \wedge t' \not\sqsubset t$$

The relation $\sqsubset$ is not explicitly defined, but its meaning should be clear based on the meaning of $\sqsubseteq$.

[DISTRIBUTION STATEMENT A] Approved for public release and unlimited distribution.

3.4.1 Type Product and Type Set Containment in the Syntactic Space

Because

- type product containment is based on $\leq$
- type aliases can be resolved in the syntactic space prior to interpretation in the semantic space

it is the case that for $n_i, n'_i \in \mathbf{D}_{\mathsf{Type}}$,

$$(\llbracket n_1 \rrbracket_{\mathsf{Type}}, \ldots, \llbracket n_m \rrbracket_{\mathsf{Type}}) \sqsubseteq (\llbracket n'_1 \rrbracket_{\mathsf{Type}}, \ldots, \llbracket n'_m \rrbracket_{\mathsf{Type}}) \Leftrightarrow \forall i.\, \exists j.\, n_i \sqsubseteq n'_j$$

Thus, type product containment can be extended to the syntactic space:

$$(n_1 * \cdots * n_m \sqsubseteq n'_1 * \cdots * n'_m) \;\Leftrightarrow\; \forall i.\, \exists j.\, n_i \sqsubseteq n'_j$$

Containment in type sets can now be extended to the syntactic space because all semantic type sets can come from syntactic type sets without aliases as members or aliases as components of any member type product. For error types and type sets, it is the case that

$$\begin{array}{ll} & \llbracket n \rrbracket_{\mathsf{Type}} \sqsubseteq \{\llbracket n'_1 \rrbracket_{\mathsf{Type}}, \ldots, \llbracket n'_m \rrbracket_{\mathsf{Type}}, \llbracket p'_1 \rrbracket_{\mathsf{Product}}, \ldots, \llbracket p'_k \rrbracket_{\mathsf{Product}}\} \\ \Leftrightarrow & \exists i.\, n \sqsubseteq n'_i \\ \Leftrightarrow & n \sqsubseteq \{n'_1, \ldots, n'_m, p'_1, \ldots, p'_k\} \end{array}$$

For error type products and type sets, it is the case that

$$\begin{array}{ll} & \llbracket p \rrbracket_{\mathsf{Product}} \sqsubseteq \{\llbracket n'_1 \rrbracket_{\mathsf{Type}}, \ldots, \llbracket n'_m \rrbracket_{\mathsf{Type}}, \llbracket p'_1 \rrbracket_{\mathsf{Product}}, \ldots, \llbracket p'_k \rrbracket_{\mathsf{Product}}\} \\ \Leftrightarrow & \exists j.\, p \sqsubseteq p'_j \\ \Leftrightarrow & p \sqsubseteq \{n'_1, \ldots, n'_m, p'_1, \ldots, p'_k\} \end{array}$$

[DISTRIBUTION STATEMENT A] Approved for public release and unlimited distribution.

4 AADL Components

This work describes the behavior of an AADL system based on the *instantiated* model of the component. It follows the general practice that only AADL instance models are analyzable. A basic mathematical model of the instance model is given as well as attributes that can be derived from the model. Naturally, it is assumed that the model contains valid AADL and EMV2 syntax (e.g., type sets and type products satisfy specific predicates); see Sections 3.2 and 3.4. When appropriate, footnotes indicate analogous classes in the EMV2 instance model or Java Runtime Environment.

The *syntactic space* for this work is the AADL model with EMV2 annex clauses. It can be thought of as a sort of parse tree. It is easy to become confused here: again, the syntactic AADL model is an AADL instance model. Therefore, *the syntactic components do not directly correspond to classifier definitions in the textual AADL.* In particular, there are no declarative entities that directly provide names for the instantiated components. Except for types, names are replaced with the named syntactic object. Additionally, the instance model aggregates information within a single component that may otherwise be distributed across component classifiers in the classifier hierarchy. This includes, for example, feature, transition, and outgoing propagation declarations.

The complete structure of the AADL+EMV2 model is not of interest here and is not presented in a comprehensive manner. Later sections introduce abstract production rules that describe some of the structure rooted at particular objects in the syntactic AADL component model. Instead, the AADL+EMV2 model is abstracted to a simple object-oriented model, where elements are "objects" from the set $\mathbf{O}$,[1] and information about an object is retrieved using a family of functions (i.e., one function for each "field" of the object). These are represented as functions that operate on components in the AADL+EMV2 model.

Finally, it is assumed the model has already been projected into a specific system operation mode (i.e., that it represents a specific configuration of components in the system).

The following first describes the syntactic and semantic models of the component hierarchy and then describes the semantics of error event, behavior state, and propagation declarations.

4.1 Component Hierarchy

4.1.1 Syntactic AADL Component Instances

Set $\mathbf{C} \subseteq \mathbf{O}$ is the set of all possible syntactic AADL components.[2] A *syntactic AADL instance model* is the tree $(\mathbf{K}, \mathbf{sub}, \mathbf{k_{Top}})$ where

- set $\mathbf{K} \subseteq \mathbf{C}$ is the set of component instances in the model
- relation $\mathbf{sub} : \mathbf{K} \times \mathbf{K}$ is the subcomponent relation between component instances: k_1 **sub** k_2 if and only if k_2 is a subcomponent of k_1
- component instance $\mathbf{k_{Top}} \in \mathbf{K}$ is the root node, that is, the component instance of the declarative component implementation that was instantiated

The function $\mathbf{D_{Sub}} : \mathbf{K} \to 2^{\mathbf{K}}$ gives the set of subcomponents of a component k: $\mathbf{D_{Sub}}(k) = \{k' \mid k \ \mathbf{sub} \ k'\}$.[3] To make the presentation herein more compact, a superscripted notation (or

[1] Cf. the Java class `Object`.
[2] Cf. `ComponentInstance` in the EMV2 instance meta model.
[3] D for **D**eclared.

[DISTRIBUTION STATEMENT A] Approved for public release and unlimited distribution.

sometimes subscripted when there is no additional label) is used for the field functions: for example, $\mathbf{D}^k_{\mathsf{Sub}} \equiv \mathbf{D}_{\mathsf{Sub}}(k)$. Additional syntactic functions are introduced as needed throughout.

4.1.2 Semantic Components

The component instance tree is mapped into a semantic component tree, where components are simply represented as symbols. Given a syntactic AADL instance model $(\mathbf{K}, \mathbf{sub}, \mathbf{k}_{\mathsf{Top}})$, the corresponding semantic component hierarchy is the tree $(\mathcal{K}, \gg, \square_{\mathsf{Top}})$ where

- Set $\mathcal{K} \subseteq \mathcal{A}$—the nodes of the tree—is the set of component instances represented by symbols.
  - Elements of $\mathcal{K}$ are referenced by the variable $\square$, because components are typically represented graphically by boxes.
  - The semantic function $[\![\cdot]\!]_{\mathsf{K}} : \mathbf{K} \to \mathcal{K}$ converts syntactic components to semantic components. Function $[\![\cdot]\!]_{\mathsf{K}}$ uniquely defines $\mathcal{K}$.
- The subcomponent relation $\gg$: $\mathcal{K} \times \mathcal{K}$—the edges of the tree—is derived from the **sub** relation:
$$\forall k_1, k_2 \in \mathbf{K}.\, [\![k_1]\!]_{\mathsf{K}} \gg [\![k_2]\!]_{\mathsf{K}} \Leftrightarrow k_1 \ \mathbf{sub}\ k_2$$
- Node $\square_{\mathsf{Top}} \in \mathcal{K}$ is the *root* of the tree: $\square_{\mathsf{Top}} = [\![\mathbf{k}_{\mathsf{Top}}]\!]_{\mathsf{K}}$.

Similar to the syntactic sets, most of the semantic sets and functions are specific to a particular semantic component. For example, the set of semantic error events $\mathcal{V}$ depends on the events declared in a specific component. Thus, $\mathcal{V}$ is really a function in $\mathcal{K} \to 2^{\mathcal{A}}$. As with syntactic relations, a subscript (or in some case a superscript) on the set is used to indicate which component is the source of the information, for example, $\mathcal{V}_{\square} = \mathcal{V}(\square)$.

The semantic equivalent to $\mathbf{D}^k_{\mathsf{Sub}}$ is the semantic relation $\mathsf{Sub} : \mathcal{K} \to 2^{\mathcal{K}}$:

- $\mathsf{Sub}(\square) = \{\square' \mid \square \gg \square'\}$

4.2 Component Declarations

4.2.1 Error Events

The syntactic domain $\mathbf{D}^k_{\mathsf{Event}} \subseteq \mathbf{O}$ is the set of error event objects declared in component instance k.[4] For component instance k with $\square = [\![k]\!]_{\mathsf{K}}$, the semantic set $\mathcal{V}_{\square} \subseteq \mathcal{A}$ is a set of symbols representing the error events.[5] These are used to indicate an *error event instance*. Note that recover and repair events are deliberately being ignored and are not included in the semantics.

The semantic function $[\![\cdot]\!]^{\square}_{\mathsf{Event}} : \mathbf{D}^k_{\mathsf{Event}} \to \mathcal{V}_{\square}$ maps a syntactic event to its semantic symbol. The function $[\![\cdot]\!]^{\square}_{\mathsf{Event}}$ uniquely defines $\mathcal{V}_{\square}$. Herein, the canonical mapping is that a syntactic event named `e` is mapped to the symbol e. For example, if a model declares an event named `Failure`, then in the corresponding automata the symbol Failure is used.

4.2.2 Error Behavior States

For each component instance k, there is the syntactic structure $(\mathbf{D}^k_{\mathsf{State}}, q^k_0)$ where

- the syntactic domain $\mathbf{D}^k_{\mathsf{State}} \subseteq \mathbf{O}$ is the set of error behavior state objects in component instance k[6]

[4] Cf. `ErrorEventInstance` in the EMV2 instance meta model.
[5] V for eVent. $\mathcal{E}$ is already used for the set of error types.
[6] Cf. `StateInstance` in the EMV2 instance meta model.

[DISTRIBUTION STATEMENT A] Approved for public release and unlimited distribution.

- the state $q_0^k \in \mathbf{D}^k_{\mathsf{State}}$ is the initial state of k

When $\square = [\![k]\!]_{\mathsf{K}}$, the semantic set $\mathcal{Q}_\square \subseteq \mathcal{A}$ is the set of symbols representing the error behavior states.[7] The semantic function $[\![\cdot]\!]^\square_{\mathsf{State}} : \mathbf{D}^k_{\mathsf{State}} \to \mathcal{Q}_\square$ maps a behavior object to its semantic symbol and uniquely defines $\mathcal{Q}_\square$. Herein, the canonical mapping is that a state named `s` is mapped to the symbol s. For example, if a model declares a state named `Normal`, then in the corresponding automata the symbol Normal is used.

4.2.3 Propagation Points

Each component has propagation points through which error types may propagate in or out. Herein, these are collectively referred to as the propagation points of a component. These propagation points are represented as fields in the environment (described in Section 5.1) but require some semantic preliminaries to be explained first, beginning with the two syntactic domains $\mathbf{D}^k_{\mathsf{In}} \subseteq \mathbf{O}_{\mathsf{PP}} \subset \mathbf{O}$ and $\mathbf{D}^k_{\mathsf{Out}} \subseteq \mathbf{O}_{\mathsf{PP}} \subset \mathbf{O}$. These are the sets of objects representing the in and out propagation points in component k.[8]

Sets $\mathbf{D}^k_{\mathsf{In}}$ and $\mathbf{D}^k_{\mathsf{Out}}$ ultimately derive from the `error propagations` clause in the EMV2 declarative model [27, E.7]. The specification refers to items declared therein as "error propagations." This is distinguished from a propagation action of a component's behavior, which is referred to as an "outgoing propagation declaration" or "outgoing propagation condition" in the specification [27, E.10]. A propagation object may correspond to a feature of a component, a user-declared propagation point, a feature in a (nested) feature group, or a binding-related point such as `processor` or `bindings`. Because the assumption is that the model derives from a legal AADL+EMV2 model, there is no need herein to formalize which propagation points appear in the syntactic model.

Note that a single propagation point may be both an in and an out propagation point. So it is possible that $\mathbf{D}^k_{\mathsf{In}} \cap \mathbf{D}^k_{\mathsf{Out}} \neq \emptyset$. From these sets, and the set $\mathbf{D}^k_{\mathsf{Sub}}$, the set of propagation *references* in component k, $\mathbf{R}^k_{\mathsf{PP}} \subseteq \mathbf{O}$ can be constructed:

$$\begin{array}{rcl} \mathbf{R}^k_{\mathsf{PP}} & = & \mathbf{D}^k_{\mathsf{In}} \cup \mathbf{R}^k_{\mathsf{Out}} \\ \mathbf{R}^k_{\mathsf{Out}} & = & \bigcup_{s \in \mathbf{D}^k_{\mathsf{Sub}}} \mathbf{D}^s_{\mathsf{Out}} \end{array}$$

The set $\mathbf{R}^k_{\mathsf{Out}}$ is the set of all out propagation objects of subcomponents of k and corresponds to the declarative syntax `s.f` in condition expressions. Sometimes it is necessary to know the subcomponent that contains the propagation reference. The function $\mathbf{PP}_{\mathsf{K}} : \mathbf{R}^k_{\mathsf{PP}} \to \mathbf{K}$ returns that component. In particular,

$$\mathbf{PP}_{\mathsf{K}}(f) = \begin{cases} k & \text{when } f \in \mathbf{D}^k_{\mathsf{In}} \\ s & \text{when } f \in \mathbf{D}^s_{\mathsf{Out}} \text{ where } s \in \mathbf{D}^k_{\mathsf{Sub}} \end{cases}$$

In addition, there are syntactic sets that describe the types a component is declared to propagate:

- $\mathbf{D}_{\triangleright\mathsf{PP}} : (\bigcup_{k \in \mathbf{K}} \mathbf{D}^k_{\mathsf{In}}) \to \mathbf{S}$ maps a propagation object representing an *in* propagation point to its declared set of propagated types.
- $\mathbf{D}_{\mathsf{PP}\triangleright} : (\bigcup_{k \in \mathbf{K}} \mathbf{D}^k_{\mathsf{Out}}) \to \mathbf{S}$ maps a propagation object representing an *out* propagation point to its declared set of propagations.

Note the following:

- The sets do not need to be parameterized by the component k because each component has a unique set of propagation objects. The domain of the mappings is thus the set of all in/out propagation points used by components in the syntactic model.

[7] Q is used because it is the canonical set of states in an automaton; see Section 9.

[8] Cf. `ErrorPropagationInstance` in the EMV2 instance meta model.

[DISTRIBUTION STATEMENT A] Approved for public release and unlimited distribution.

- The codomain of both these mappings is *still a syntactic set* whose semantic value must be obtained using the semantic function $[\![\cdot]\!]_{\mathsf{Set}}$.

[DISTRIBUTION STATEMENT A] Approved for public release and unlimited distribution.

5 Environments

EMV2 condition expressions refer to event occurrences and error type propagations. These can be viewed as references to the values of variables. Thus, an environment is needed to evaluate a condition. Later sections describe how the environment is actually an input symbol to the automata representing the component's error behavior. Furthermore, as becomes evident in Section 6, evaluation has a non-traditional meaning: it produces a set of environments.

To begin, the set $\Gamma_\square$ is the set of environments for component $\square = [\![k]\!]_{\mathsf{K}}$. Specific environments are identified using the variable $\gamma \in \Gamma_\square$. The elements of set $\Gamma_\square$ are structured as records with the index set $\mathcal{I}_\square$, so we have

$$(\Gamma_\square, \mathcal{I}_\square, E_\square, \epsilon_\square)$$

In the following, the different components of the record definition are described in more detail.

5.1 Fields — $\mathcal{I}_\square$

Generally speaking, the environment contains

- one field indicating the current event instance, if any
- one field for each propagation point of the component, each indicating that point's currently propagated type

The *semantic function* $[\![\cdot]\!]^\square_{\mathsf{Field}} : \mathbf{R}^k_{\mathsf{PP}} \to \mathcal{I}_\square$ maps references to propagation points to fields of the environment. The index set $\mathcal{I}_\square$ and semantic function $[\![\cdot]\!]^\square_{\mathsf{Field}}$ are concurrently defined below:

- For the symbol $\mathsf{event} \in \mathcal{I}_\square$, the value of this field represents an event instance: it is a member of $\mathcal{V}_\square$, the set of symbols representing different errors. There needs to be only one such field because events cannot happen simultaneously [27, E.8.1.(2)]. This field is used solely by the semantics, and it cannot be described syntactically: $\forall x \in \mathbf{R}^k_{\mathsf{PP}}. [\![x]\!]^\square_{\mathsf{Field}} \neq \mathsf{event}$.
- For each $f \in \mathbf{R}^k_{\mathsf{PP}}$ there is a unique symbol $i \in \mathcal{I}_\square$ such that $[\![f]\!]^\square_{\mathsf{Field}} = i$. The value of this field is an error type, that is, a member of $\mathcal{T}$. But more specifically, this value can be restricted to be from the set of types that the field can actually propagate:
  - For $f \in \mathbf{D}^k_{\mathsf{In}}$ the set of types is $[\![\mathbf{D}_{\triangleright\mathsf{PP}}(f)]\!]_{\mathsf{Set}}$.
  - For $f \in \mathbf{R}^k_{\mathsf{Out}}$ the set of types is $[\![\mathbf{D}_{\mathsf{PP}\triangleright}(f)]\!]_{\mathsf{Set}}$.
- Finally, the function $[\![\cdot]\!]^\square_{\mathsf{Field}}$ uniquely defines the set $\mathcal{I}_\square \setminus \{\mathsf{event}\}$.

While the choice of symbols used for the non event members of the index set is arbitrary, we use the following canonical mapping in the examples below:

- For $\mathtt{f} \in \mathbf{D}^k_{\mathsf{In}}$, $[\![\mathtt{f}]\!]^\square_{\mathsf{Field}} = \mathsf{f}$ (e.g., feature name `input` is mapped to the symbol input).
- For $(\mathtt{s}, \mathtt{f}) \in \mathbf{R}^k_{\mathsf{Out}}$, $[\![(\mathtt{s}, \mathtt{f})]\!]^\square_{\mathsf{Field}} = \mathsf{s\_f}$ (e.g., feature name `output` of the component referenced by the subcomponent name `sub` is mapped to the symbol sub_output).

5.2 Empty Values — $E_\square$ and $\epsilon_\square$

The sets $\mathcal{T}$ and $\mathcal{V}_\square$ do not contain elements that can be used to signify an empty field value. Thus, two fresh symbols ϵ_{Type} and $\epsilon_{\mathsf{Event}}$ are introduced:

[DISTRIBUTION STATEMENT A] Approved for public release and unlimited distribution.

- $\epsilon_{\mathsf{Type}} \notin \mathcal{T}$
- $\forall \square \in \mathcal{K}.\, \epsilon_{\mathsf{Event}} \notin \mathcal{V}_{\square}$

Now

$$E_{\square} = \{\epsilon_{\mathsf{Type}}, \epsilon_{\mathsf{Event}}\}$$

The mapping $\epsilon_{\square}$ is defined:

- $\epsilon_{\square}(\mathsf{event}) = \epsilon_{\mathsf{Event}}$
- $\forall f \in \mathbf{R}^{k}_{\mathsf{PP}}.\, \epsilon_{\square}([\![f]\!]^{\square}_{\mathsf{Field}}) = \epsilon_{\mathsf{Type}}$

5.3 Environment Structure — $\Gamma_{\square}$

Finally, the structure of the records can be fully specified based on the above details. Incorporating the empty values and the labeled cross product, the set $\Gamma_{\square}$ is

$$\begin{array}{rcl} \Gamma_{\square} & = & \mathsf{event} : (\mathcal{V}_{\square} \cup \{\epsilon_{\mathsf{Event}}\}) \\ & \times & \prod_{f \in \mathbf{D}^{k}_{\mathsf{In}}} [\![f]\!]^{\square}_{\mathsf{Field}} : ([\![\mathbf{D}_{\triangleright \mathsf{PP}}(f)]\!]_{\mathsf{Set}} \cup \{\epsilon_{\mathsf{Type}}\}) \\ & \times & \prod_{f \in \mathbf{R}^{k}_{\mathsf{Out}}} [\![f]\!]^{\square}_{\mathsf{Field}} : ([\![\mathbf{D}_{\mathsf{PP}\triangleright}(f)]\!]_{\mathsf{Set}} \cup \{\epsilon_{\mathsf{Type}}\}) \end{array}$$

[DISTRIBUTION STATEMENT A] Approved for public release and unlimited distribution.

6 Condition Expressions, Part 1: Basic Expressions

Enough mathematical machinery has been developed to finally present the semantics of EMV2 conditions, the Boolean expressions of the language. Here, a non-traditional meaning is given to expressions: instead of evaluating to a Boolean true or false value, *conditions evaluate to the set of environments in which the condition is true.* This interpretation is used because the automata constructed for the component's error behavior is based on a syntactic automata [9, 33], and this meaning is consistent with the semantics required in such a scenario.

6.1 The Basic Expression Language

This section presents the semantics of a condition expression language containing basic expressions over EMV2 triggers. The full grammar for EMV2 error conditions contains additional primitive operators `all`, `orless`, and `ormore` and includes both inclusive or—`or`—and exclusive or—`xor`.

The basic expressions

- exclude exclusive or
- exclude the primitive operators
- do not enforce the silencing constraints of EMV2 [27, E.8.2.(8)], but do provide the building blocks to do so
- contain an expression to express an empty event that is not expressible in EMV2

Sections 7 and 8 give the semantics of the actual EMV2 condition expressions in terms of the semantics of the basic expressions. In particular, we show how to enforce [27, E.8.2.(8)] and convert the primitive operators.

We again emphasize that the model is based on the instantiated AADL and EMV2 information. Thus the abstract production rules actually describe the expression structure as it is in the instantiation. This is most obvious in the case of propagation targets (production rule $\mathbf{Trigger\_Prop}_k$), which have different syntactic forms (e.g., `f` and `s.f`) in the declarative model to name local or subcomponent propagation points. In the production rule, however, the propagation point object, which already abstracts away this distinction, is directly used.

The syntactic domains and abstract production rules for the simple language are shown in Figures 6.2 and 6.3. Many of the rules are subscripted by a syntactic component $k \in \mathcal{K}$ indicating that the rule has specific versions for each component. This is because the syntactic sets of objects are used in the rules to indicate namespaces: for example, abstract production rule $\mathbf{Trigger\_Event}_k$ uses $\mathbf{D}^k_{\mathsf{Event}}$ instead of $\mathbf{O}$ to indicate that the event should be an event declared in component k.

Note as well the following:

- The basic expression language uses keywords and symbols that are slightly different from those in the EMV2 condition language to emphasize that the languages are different.
- Following the conventions of denotational semantics, the abstract production rules are written ambiguously (e.g., the production $\mathbf{Condition}_k ::= \mathbf{Condition}_k + \mathbf{Condition}_k$), but it is expected that expressions are parsed and evaluated using *left association.* See Schmidt [28, pp. 6–9] for a discussion of this issue. So the condition `A + B + C` has the parse tree in Figure 6.1(a) and *not* the parse tree in Figure 6.1(b).

[DISTRIBUTION STATEMENT A] Approved for public release and unlimited distribution.

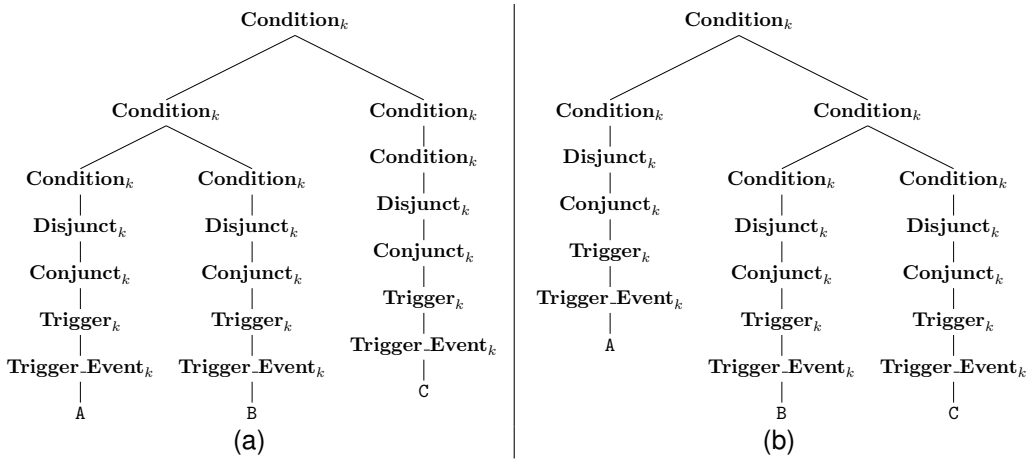

Figure 6.1: The (a) Correct Left-Associative Parsing of `A + B + C` *and (b) Incorrect Right-Associative Parsing*

$$\mathbf{C}_k : \mathbf{Condition}_k \qquad \mathbf{T}^k_{\mathsf{Event}} : \mathbf{Trigger\_Event}_k$$
$$\mathbf{C}^k_{+} : \mathbf{Disjunct}_k \qquad \mathbf{T}^k_{\mathsf{PP}} : \mathbf{Trigger\_Prop}_k$$
$$\mathbf{C}^k_{\&} : \mathbf{Conjunct}_k \qquad \mathbf{S}_{\mathsf{NoError}} : \mathbf{TypeSet\_or\_NoError}$$
$$\mathbf{T}_k : \mathbf{Trigger}_k$$

Figure 6.2: The Syntactic Domains for the Simple Condition Language

$$\begin{array}{rcl}
\mathbf{Condition}_k & ::= & \mathbf{Disjunct}_k \mid \mathbf{Condition}_k + \mathbf{Condition}_k \\
\mathbf{Disjunct}_k & ::= & \mathbf{Conjunct}_k \mid \mathbf{Disjunct}_k \ \& \ \mathbf{Disjunct}_k \\
\mathbf{Conjunct}_k & ::= & (\ \mathbf{Condition}_k\ ) \mid \mathbf{Trigger}_k \\
\mathbf{Trigger}_k & ::= & \mathbf{Trigger_Event}_k \\
 & \mid & \mathbf{Trigger_Prop}_k \\
\mathbf{Trigger_Event}_k & ::= & \texttt{event}\ \mathbf{D}^k_{\mathsf{Event}} \\
 & \mid & \texttt{noevent} \\
\mathbf{Trigger_Prop}_k & ::= & \texttt{in}\ \mathbf{D}^k_{\mathsf{In}}\ \mathbf{TypeSet_or_NoError} \\
 & \mid & \texttt{out}\ \mathbf{R}^k_{\mathsf{Out}}\ \mathbf{TypeSet_or_NoError} \\
\mathbf{TypeSet_or_NoError} & ::= & \mathbf{TypeSet} \mid \{\ \texttt{noerror}\ \} \mid \epsilon \\
\mathbf{TypeSet} & ::= & \text{(Defined in Section 3.3)}
\end{array}$$

Figure 6.3: The Production Rules for the Simple Condition Language

[DISTRIBUTION STATEMENT A] Approved for public release and unlimited distribution.

The following sections describe the semantics bottom-up, starting with the specific trigger expressions. It is assumed that the condition is in component k and that $\square = [\![k]\!]_{\mathsf{K}}$.

6.2 Event Trigger

An event trigger condition is satisfied when an instance of the named event occurs (i.e., the named event is the value of the event field of the environment). As stated above, the meaning of a condition is the set of environments in which the condition is satisfied. Thus for event triggers the semantic function is $[\![\cdot]\!]^{\square}_{\mathsf{Trigger\_E}} : \mathbf{T}^{k}_{\mathsf{Event}} \to 2^{\Gamma_\square}$:

$$\mathsf{instance} : (\mathcal{V}_\square \cup \epsilon_{\mathsf{Event}}) \to 2^{\Gamma_\square}$$

$$\begin{array}{rcl} \mathsf{instance}(\sigma) & = & \{\gamma \in \Gamma_\square \mid \gamma[\mathsf{event}] = \sigma\} \\ [\![\texttt{event } e]\!]^{\square}_{\mathsf{Trigger\_E}} & = & \mathsf{instance}([\![e]\!]^{\square}_{\mathsf{Event}}) \\ [\![\texttt{noevent}]\!]^{\square}_{\mathsf{Trigger\_E}} & = & \mathsf{instance}(\epsilon_{\mathsf{Event}}) \end{array}$$

6.3 Propagation Trigger

A propagation trigger is satisfied when the error type propagated by the named propagation point is contained in the given type set. The catch is that there may not be a type set specified by the trigger:

- It could be that {noerror} is explicitly given, in which case the named propagation point must not be propagating an error (i.e, it is propagating ϵ_{Type}).
- It could be that no set is given at all, in which case the type set is assumed to be the type set specified in the declared propagations of the propagation point (i.e, from $\mathbf{D}_{\triangleright\mathsf{PP}}$ or $\mathbf{D}_{\mathsf{PP}\triangleright}$).

The semantic function $[\![\cdot]\!]_{\mathsf{TS}} : \mathbf{S}_{\mathsf{NoError}} \to \mathcal{S} \to (\mathcal{T} \cup \epsilon_{\mathsf{Type}}) \to \mathbb{B}$ evaluates a type set specification to a function that can be used to test a propagated type against the specification. The curried function has two arguments and a Boolean return value:

- The first semantic type set argument is a default set to use for comparison, in case the type set is not specified as part of the trigger. See the case for ϵ.
- The second type argument is the propagated type (or no type at all) to test.

For the three cases of the **TypeSet_or_NoError** production rule, the functions are

$$\begin{array}{rcl} [\![ts \in \mathbf{S}]\!]_{\mathsf{TS}} & = & \lambda\, default.\ \lambda\, v.\ v \sqsubseteq [\![ts]\!]_{\mathsf{Set}} \\ [\![\{\ \texttt{noerror}\ \}]\!]_{\mathsf{TS}} & = & \lambda\, default.\ \lambda\, v.\ v = \epsilon_{\mathsf{Type}} \\ [\![\epsilon]\!]_{\mathsf{TS}} & = & \lambda\, default.\ \lambda\, v.\ v \sqsubseteq default \end{array}$$

Specifically,

- when a syntactic type set is provided, the comparison function simply tests if the propagated type value is contained in the corresponding semantic type set.
- when {noerror} is specified, the comparison function tests if the propagated type value is the empty type value ϵ_{Type}.
- when the type set is not specified at all (ϵ), the propagated type value is tested against the provided default set. The functions below ensure that the default value is derived from the declared propagations of the correct propagation point.

[DISTRIBUTION STATEMENT A] Approved for public release and unlimited distribution.

The semantic function $[\![\cdot]\!]^{\square}_{\mathsf{Trigger\_P}} : \mathbf{T}^k_{\mathsf{PP}} \to 2^{\Gamma\square}$ is built from $[\![\cdot]\!]_{\mathsf{TS}}$:

$$\begin{array}{rcl}
[\![\texttt{in}\ f\ \mathit{TSorNE}]\!]^{\square}_{\mathsf{Trigger_P}} & = & \{\gamma \in \Gamma\square \mid \phi(d)(\gamma[i])\} \\
 & & \text{where} \quad \phi = [\![\mathit{TSorNE}]\!]_{\mathsf{TS}} \\
 & & \qquad\quad d = [\![\mathbf{D}_{\triangleright\mathsf{PP}}(f)]\!]_{\mathsf{Set}} \\
 & & \qquad\quad i = [\![f]\!]^{\square}_{\mathsf{Field}} \\
[\![\texttt{out}\ f\ \mathit{TSorNE}]\!]^{\square}_{\mathsf{Trigger_P}} & = & \{\gamma \in \Gamma\square \mid \phi(d)(\gamma[i])\} \\
 & & \text{where} \quad \phi = [\![\mathit{TSorNE}]\!]_{\mathsf{TS}} \\
 & & \qquad\quad d = [\![\mathbf{D}_{\mathsf{PP}\triangleright}(f)]\!]_{\mathsf{Set}} \\
 & & \qquad\quad i = [\![f]\!]^{\square}_{\mathsf{Field}}
\end{array}$$

6.4 Triggers

A trigger condition is an event or propagation trigger. The semantic function $[\![\cdot]\!]^{\square}_{\mathsf{Trigger}} : \mathbf{T}_k \to 2^{\Gamma\square}$ evaluates a trigger based on its category:

$$\begin{array}{rcl}
[\![t \in \mathbf{T}^k_{\mathsf{Event}}]\!]^{\square}_{\mathsf{Trigger}} & = & [\![t]\!]^{\square}_{\mathsf{Trigger_E}} \\
[\![t \in \mathbf{T}^k_{\mathsf{PP}}]\!]^{\square}_{\mathsf{Trigger}} & = & [\![t]\!]^{\square}_{\mathsf{Trigger_P}}
\end{array}$$

6.5 Conjuncts

A conjunctive term is either a parenthesized condition or a trigger condition. The semantic function $[\![\cdot]\!]^{\square}_{\&} : \mathbf{C}^k_{\&} \to 2^{\Gamma\square}$ evaluates a conjunct in the obvious way:

$$\begin{array}{rcl}
[\![(\ c\ )]\!]^{\square}_{\&} & = & [\![c]\!]^{\square}_{\mathsf{Cond}} \\
[\![t \in \mathbf{T}_k]\!]^{\square}_{\&} & = & [\![t]\!]^{\square}_{\mathsf{Trigger}}
\end{array}$$

6.6 Disjuncts

A disjunctive term is a conjunction and is evaluated by the semantic function $[\![\cdot]\!]^{\square}_{+} : \mathbf{C}^k_{+} \to 2^{\Gamma\square}$ as an intersection of sets:

$$\begin{array}{rcl}
[\![c_1\ \&\ c_2]\!]^{\square}_{+} & = & [\![c_1]\!]^{\square}_{+} \cap [\![c_2]\!]^{\square}_{+} \\
[\![c]\!]^{\square}_{+} & = & [\![c]\!]^{\square}_{\&}
\end{array}$$

Recall that the operator + is treated as left-associative.

6.7 Condition

Finally, a full condition is a disjunction evaluated by the semantic function $[\![\cdot]\!]^{\square}_{\mathsf{Cond}} : \mathbf{C}_k \to 2^{\Gamma\square}$ as a union of sets:

$$\begin{array}{rcl}
[\![c_1\ +\ c_2]\!]^{\square}_{\mathsf{Cond}} & = & [\![c_1]\!]^{\square}_{\mathsf{Cond}} \cup [\![c_2]\!]^{\square}_{\mathsf{Cond}} \\
[\![c]\!]^{\square}_{\mathsf{Cond}} & = & [\![c]\!]^{\square}_{+}
\end{array}$$

Recall that the operator & is treated as left-associative.

[DISTRIBUTION STATEMENT A] Approved for public release and unlimited distribution.

7 Condition Expressions, Part 2: Transition Conditions

As mentioned previously, the basic expression language of Section 6 is not the condition language actually used by EMV2. To repeat,

- it contains a `noevent` trigger that is not expressible in EMV2
- EMV2 supports both the inclusive `or` operator and the exclusive `xor` operator
- EMV2 imposes constraints on the propagation points not explicitly mentioned in a condition [27, E.8.2.(8)]
- EMV2 features three logical primitives that must be interpreted: `and`, `orless`, and `ormore`

The syntactic domains and abstract production rules for this language are shown in Figures 7.1 and 7.2, based on the grammar in the EMV2 specification. Although, here, unlike in the specification, the grammar rules are used to impose the correct order of operations as specified in EMV2 [27, E.8.(L30)]. Note the following:

- The names of the syntactic domains are decorated with $\hat{}$ and the names of the production rules are prefixed with $\diamond$ to distinguish them from those of the simple language.
- As before, the production rules are ambiguous, but the operators `and`, `or`, and `xor` are left-associative.

Here natural numbers are added to the grammar via the syntactic domain $\mathbf{Natural} \subset \mathbf{O}$, which contains objects representing natural numbers.[1] The semantic function $[\![\cdot]\!]_{\mathsf{N}} : \mathbf{N} \to \mathbb{N}$ maps the object representation into a mathematical natural number.

In later sections, condition expressions are retrieved from objects representing transitions and outgoing propagation declarations. This can be thought of as obtaining an object representing an element of $\hat{\mathbf{C}}_k$, where the abstract production rules describe the structure of the object tree rooted at that node.[2]

The rest of this section develops the translation function $\lfloor\cdot\rfloor^k_{\mathsf{Cond}} : \hat{\mathbf{C}}_k \to \mathbf{C}_k$ that converts from strings in the EMV2 condition language to strings in the basic expression language.

7.1 Rule E.8.2.(8): Silencing Unused Propagations

Before a translation to the simple language can be constructed, it is necessary to understand what is required of the translation. The primary intricacy of the translation is to enforce a requirement described in EMV2:

[1] Cf. `Java.lang.Long` in the EMV2 instance meta model.
[2] Cf. `ConditionExpressionInstance` in the EMV2 instance meta model.

$$
\begin{array}{cc}
\hat{\mathbf{C}}_k : \diamond\mathbf{Condition}_k & \hat{\mathbf{T}}^k_{\mathsf{List}} : \diamond\mathbf{Triggers}_k \\
\hat{\mathbf{C}}^k_{\mathsf{xor}} : \diamond\mathbf{Disjunct}_k & \hat{\mathbf{T}}_k : \diamond\mathbf{Trigger}_k \\
\hat{\mathbf{C}}^k_{\mathsf{and}} : \diamond\mathbf{Conjunct}_k & \hat{\mathbf{T}}^k_{\mathsf{Event}} : \diamond\mathbf{Trigger\_Event}_k \\
\hat{\mathbf{M}}_k : \diamond\mathbf{Term}_k & \hat{\mathbf{T}}^k_{\mathsf{PP}} : \diamond\mathbf{Trigger\_Prop}_k \\
\hat{\mathbf{P}}_k : \diamond\mathbf{Primitive}_k & \mathbf{Natural} : \text{See Main Text}
\end{array}
$$

Figure 7.1: The Syntactic Domains for EMV2 Condition Expressions

[DISTRIBUTION STATEMENT A] Approved for public release and unlimited distribution.

$$
\begin{array}{rcl}
\diamond\mathbf{Condition}_k & ::= & \diamond\mathbf{Disjunct}_k \\
& | & \diamond\mathbf{Condition}_k \texttt{ or } \diamond\mathbf{Condition}_k \\
& | & \diamond\mathbf{Condition}_k \texttt{ xor } \diamond\mathbf{Condition}_k \\
\diamond\mathbf{Disjunct}_k & ::= & \diamond\mathbf{Conjuct}_k \mid \diamond\mathbf{Disjunct}_k \texttt{ and } \diamond\mathbf{Disjunct}_k \\
\diamond\mathbf{Conjunct}_k & ::= & (\ \diamond\mathbf{Condition}_k\ ) \mid \diamond\mathbf{Term}_k \\
\diamond\mathbf{Term}_k & ::= & \diamond\mathbf{Primitive}_k \mid \diamond\mathbf{Trigger}_k \\
\diamond\mathbf{Primitive}_k & ::= & \texttt{all } (\ \diamond\mathbf{Triggers}_k) \\
& | & \texttt{all} - \mathbf{Natural}\ (\ \diamond\mathbf{Triggers}_k) \\
& | & \mathbf{Natural}\ \texttt{ormore}\ (\ \diamond\mathbf{Triggers}_k) \\
& | & \mathbf{Natural}\ \texttt{orless}\ (\ \diamond\mathbf{Triggers}_k) \\
\diamond\mathbf{Triggers}_k & ::= & \diamond\mathbf{Trigger}_k \mid \diamond\mathbf{Trigger}_k, \diamond\mathbf{Triggers}_k \\
\diamond\mathbf{Trigger}_k & ::= & \diamond\mathbf{Trigger\_Event}_k \\
& | & \diamond\mathbf{Trigger\_Prop}_k \\
\diamond\mathbf{Trigger\_Event}_k & ::= & \mathbf{D}^k_{\mathsf{Event}} \\
\diamond\mathbf{Trigger\_Prop}_k & ::= & \texttt{in } \mathbf{D}^k_{\mathsf{In}}\ \mathbf{TypeSet\_or\_NoError} \\
& | & \texttt{out } \mathbf{R}^k_{\mathsf{Out}}\ \mathbf{TypeSet\_or\_NoError} \\
\mathbf{TypeSet\_or\_NoError} & ::= & \text{(Defined in Figure 6.3)}
\end{array}
$$

Figure 7.2: The Abstract Production Rules for EMV2 Condition Expressions

> If an alternative transition condition specifies a single error propagation point, e.g., `port1{BadValue}`, by itself, then *all other incoming error propagation points must not have a propagation present.* [27, E.8.2.(8)] [emphasis added]

This requirement is qualified by the following:

> Note: we chose to interpret *listing a single error propagation point as all others being error free*, because modelers often assume that they are dealing with one incoming error propagation at a time. Optionally, the user can explicitly indicate that the other error propagation points have **NoError**. [27, E.8.2.(9)] [emphasis added]

This is an interesting restriction because it requires knowing *what is not specified* by a condition. Fortunately, the universe of propagation points available to a condition in component instance k is already known: $\mathbf{R}^k_{\mathsf{PP}}$. Thus, knowing that a solitary propagation f is referenced, then the set of propagation points not referenced is the set $\overline{f} = \mathbf{R}^k_{\mathsf{PP}} \setminus \{f\}$. A propagation trigger that tests for `noerror` and the `noevent` trigger are known as *silent triggers*. Thus inserting implied silent triggers is henceforth known as *silencing*.

There is significant ambiguity in the EMV2 standard regarding the interpretation of condition expressions. This is revisited in Sections 8.3.1 and 9.5.1; the problems it causes for fault trees is discussed in Section 10.5. Difficulty in understanding the intended meaning of condition expressions is caused by the standard using simplistic examples to define the meaning of expressions, leaving questions about the interpretation in more complex cases. The sole definition of `and` in EMV2, for example, is

> If the alternative transition condition specifies `port1BadValue` **and** `port2BadValue`, then the condition is satisfied if error propagations are present on both ports. [27, E.8.2.(8)]

Generally, the descriptions use propagation triggers only, do not explore cases where logical operators are nested, do not fully elaborate the meanings of the logical primitives, and do not show primitives combined with logical operators. This makes understanding the full role of events difficult. Of particular interest here is that it is unclear if silencing in the case of a singular propagation trigger is meant to silence events or not. It is also unclear if a solitary ref-

[DISTRIBUTION STATEMENT A] Approved for public release and unlimited distribution.

$$\mathsf{satisfiedBy}_{\square}(c) \quad = \quad \begin{cases} [\![(\parallel \lfloor c \rfloor^{k}_{\mathsf{Cond}} \parallel) \,\&\, (\parallel \mathbf{silence}_k(\overline{f}) \parallel)]\!]^{\square}_{\mathsf{Cond}} \text{ when } c \in \hat{\mathbf{T}}_{\mathsf{k}} \text{ and } \overline{f} \neq \emptyset \\ [\![\lfloor c \rfloor^{k}_{\mathsf{Cond}}]\!]^{\square}_{\mathsf{Cond}} \text{ otherwise} \end{cases}$$
$$\text{where } \overline{f} = \mathbf{L}_{\mathsf{k}} \setminus \mathbf{specified}_k(c)$$

Figure 7.3: Semantic Helper Function $\mathsf{satisfiedBy}_{\square}$

$$\begin{aligned}
\mathbf{specified}_k(C_1 \texttt{ or } C_2) &= \mathbf{specified}_k(C_1) \cup \mathbf{specified}_k(C_2) \\
\mathbf{specified}_k(C_1 \texttt{ xor } C_2) &= \mathbf{specified}_k(C_1) \cup \mathbf{specified}_k(C_2) \\
\mathbf{specified}_k(C_1 \texttt{ and } C_2) &= \mathbf{specified}_k(C_1) \cup \mathbf{specified}_k(C_2) \\
\mathbf{specified}_k(\texttt{((} C \texttt{))}) &= \mathbf{specified}_k(C) \\
\mathbf{specified}_k(\texttt{all (} \mathit{Triggers} \texttt{))}) &= \mathbf{specified}_k(\mathit{Triggers}) \\
\mathbf{specified}_k(\texttt{all} - N \texttt{ (} \mathit{Triggers} \texttt{))}) &= \mathbf{specified}_k(\mathit{Triggers}) \\
\mathbf{specified}_k(N \texttt{ ormore (} \mathit{Triggers} \texttt{))}) &= \mathbf{specified}_k(\mathit{Triggers}) \\
\mathbf{specified}_k(N \texttt{ orless (} \mathit{Triggers} \texttt{))}) &= \mathbf{specified}_k(\mathit{Triggers}) \\
\mathbf{specified}_k(t \texttt{ , } \mathit{Triggers}) &= \mathbf{specified}_k(t) \cup \mathbf{specified}_k(\mathit{Triggers}) \\
\mathbf{specified}_k(e) &= \{\mathsf{evt}\} \\
\mathbf{specified}_k(\texttt{in } f \ ts) &= \{(\mathsf{in}, f)\} \\
\mathbf{specified}_k(\texttt{out } f \ ts) &= \{(\mathsf{out}, f)\}
\end{aligned}$$

Figure 7.4: Determining the Triggers Specified in an Expression

erence to an event instance should silence all the propagations. *Herein the decision is made to apply silencing in both of these cases.*

This requirement applies to an *alternative transition condition* [27, E.8.2.(7)], which is really the condition as a whole that appears in a particular transition declaration. It must be enforced, therefore, at a level *above* the $\diamond\mathbf{Condition}_k$ production rule. For a component instance k where $\square = [\![k]\!]_{\mathsf{K}}$, a new semantic helper function $\mathsf{satisfiedBy}_{\square} : \hat{\mathbf{C}}_k \rightarrow 2^{\Gamma\square}$ is introduced in Figure 7.3. It determines the set of environments that satisfy a particular alternate transition condition. When the input condition expression $c \in \hat{\mathbf{T}}_{\mathsf{k}}$, that is, is a solitary event or propagation, silencing is applied; otherwise the condition is translated without modification. Specifically, a modified basic expression in $\mathbf{C}_{\mathsf{k}}$ is constructed and interpreted using $[\![\cdot]\!]^{\square}_{\mathsf{Cond}}$—the symbol $\parallel$ is the string concatenation operator.

The set $\mathbf{L}_{\mathsf{k}}$ is the set of labeled propagation points (i.e., they are "tagged" with in or out): $\mathbf{L}_{\mathsf{k}} = (\{\mathsf{in}\} \times \mathbf{D}^k_{\mathsf{In}}) \cup (\{\mathsf{out}\} \times \mathbf{R}^k_{\mathsf{Out}}) \cup \{\mathsf{evt}\}$. These tags are necessary to reconstruct the correct syntactic trigger when generating the silenced expression (see below). The syntactic function $\mathbf{silence}_k$—defined in next section—generates a conjunction of empty triggers expressed in the simple condition language from the given set of triggers. The syntactic function $\mathbf{specified}_k : (\hat{\mathbf{C}}_k \cup \hat{\mathbf{T}}^k_{\mathsf{List}}) \rightarrow 2^{\mathbf{L}_{\mathsf{k}}}$ in Figure 7.4 "analyzes" a condition and returns a set consisting of all the triggers used by the condition. Considering that silencing applies only to solitary triggers, this function is definitely overkill. However, silencing is also used to define exclusive or; thus arbitrary expressions may require silencing.

Finally, the syntactic function $\mathbf{silence}_k : 2^{\mathbf{L}_{\mathsf{k}}} \rightarrow (\mathbf{C}^k_+ \cup \{\epsilon\})$ in Figure 7.5, intended to operate in conjunction with the results of the $\mathbf{specified}_k$ function, generates conjunctions of silent triggers in the basic expression language based on the input set. It bears repeating that the output of this function is a *string* in the set of strings $\mathbf{C}^k_+ \cup \{\epsilon\}$.

[DISTRIBUTION STATEMENT A] Approved for public release and unlimited distribution.

$$
\begin{aligned}
\mathbf{silence}_k(\{l_1, \ldots, l_n\}) &= \mathbf{silence}_k(\{l_1\}) \parallel \& \parallel \mathbf{silence}_k(\{l_2, \ldots, l_n\}) \\
\mathbf{silence}_k(\{(\mathsf{in}, f)\}) &= \texttt{in} \parallel f \parallel \{\ \texttt{noerror}\ \} \\
\mathbf{silence}_k(\{(\mathsf{out}, f)\}) &= \texttt{out} \parallel f \parallel \{\ \texttt{noerror}\ \} \\
\mathbf{silence}_k(\{\mathsf{evt}\}) &= \texttt{noevent} \\
\mathbf{silence}_k(\emptyset) &= \epsilon
\end{aligned}
$$

Figure 7.5: Function to Silence Triggers

7.1.1 Silencing Example

Consider the instantiation of `Example.i` in Listing 7.1; that is, k is the component instance for the instantiation of `Example.i`, and $\square$ is the associated semantic component. Ignoring the implicit propagation points, such as `bindings`, the set of labeled propagation point references is $\mathbf{L}_\mathsf{k} = \{(\mathsf{in}, \mathtt{in1}), (\mathsf{in}, \mathtt{in2}), \mathsf{evt}\}$. Consider how $\mathsf{satisfiedBy}_\square$, $\mathbf{specified}_k$, and $\mathbf{silence}_k$ interact and evaluate the alternate condition expression of transition `t1`.

```
1  package Q
2  public
3    annex EMV2 {**
4      error types
5        T: type;
6      end types;
7
8      error behavior Simple_Behavior
9      events
10       X: error event;
11     states
12       S1: initial state;
13       S2: state;
14       S3: state;
15     end behavior;
16   **};
17
18   system Example
19     features
20       in1: in event port;
21       in2: in event port;
22     annex EMV2 {**
23     use types Q;
24       use behavior Q::Simple_Behavior;
25
26       error propagations
27         in1: in propagation {T};
28         in2: in propagation {T};
29       end propagations;
30
31       component error behavior
32       transitions
33         t1: S1 -[X]-> S2;
34         t2: S1 -[X and in1{T} and int{T}]-> S3;
35       end component;
36     **};
37   end Example;
```


[DISTRIBUTION STATEMENT A] Approved for public release and unlimited distribution.

```
38
39    system implementation Example.i
40    end Example.i;
41 end Q;
```

Listing 7.1: An Example EMV2 Model to Demonstrate Silencing

Condition `t1` is the solitary event instance `X`: the condition `X` is the EMV2 condition language. According to rule E.8.2.(8), then, it must be assumed that there are no propagations on `in1` and `in2`.

- The set of specified propagation points in the condition is $\mathbf{specified}_k(\texttt{event X}) = \{\mathsf{evt}\}$.
- The set of unspecified propagation points is then $\overline{f} = \mathbf{L_k} \setminus \mathbf{specified}_k(\texttt{event X}) = \{(\mathsf{in}, \texttt{in1}), (\mathsf{in}, \texttt{in2}), \mathsf{evt}\} \setminus \{\mathsf{evt}\} = \{(\mathsf{in}, \texttt{in1}), (\mathsf{in}, \texttt{in2})\}$.
- The resulting expression of silenced propagations is then $\mathbf{silence}_k(\overline{f}) = \mathbf{silence}_k(\{(\mathsf{in}, \texttt{in1}), (\mathsf{in}, \texttt{in2})\}) = \texttt{in in1 \{noerror\} \& in in2 \{noerror\}}$.

Condition `t2`, `X and in1{T} and in2{T}`, is `X and in in1{T} and in in2{T}` in the EMV2 expression language. It is not a solitary trigger, so silencing is not applied to it.

7.2 Translation to Basic Expressions

It remains to define $\lfloor\cdot\rfloor^k_{\mathsf{Cond}} : \hat{\mathbf{C}}_k \to \mathbf{C}_k$, introduced for $\mathsf{satisfiedBy}_\square$. The scheme is to translate from the EMV2 condition language to the basic expression language and rely on the semantic functions of the basic expression language.

Event and propagation triggers are translated directly without any additional meaning. The translation functions $\lfloor\cdot\rfloor^k_{\mathsf{Trigger\_E}} : \hat{\mathbf{T}}^k_{\mathsf{Event}} \to \mathbf{T}^k_{\mathsf{Event}}$ and $\lfloor\cdot\rfloor^k_{\mathsf{Trigger\_P}} : \hat{\mathbf{T}}^k_{\mathsf{PP}} \to \mathbf{T}^k_{\mathsf{PP}}$ for event and propagation triggers, respectively, are shown below:

$$\begin{aligned} \lfloor e \rfloor^k_{\mathsf{Trigger\_E}} &= \texttt{event } e \\ \lfloor \texttt{in } f \ \mathit{TSorNE} \rfloor^k_{\mathsf{Trigger\_P}} &= \texttt{in } f \ \mathit{TSorNE} \\ \lfloor \texttt{out } f \ \mathit{TSorNE} \rfloor^k_{\mathsf{Trigger\_P}} &= \texttt{out } f \ \mathit{TSorNE} \end{aligned}$$

The translation function $\lfloor\cdot\rfloor^k_{\mathsf{Trigger}} : \hat{\mathbf{T}}_k \to \mathbf{T}_k$ translates a trigger based on its category:

$$\begin{aligned} \lfloor t \in \hat{\mathbf{T}}^k_{\mathsf{Event}} \rfloor^k_{\mathsf{Trigger}} &= \lfloor t \rfloor^k_{\mathsf{Trigger\_E}} \\ \lfloor t \in \hat{\mathbf{T}}^k_{\mathsf{PP}} \rfloor^k_{\mathsf{Trigger}} &= \lfloor t \rfloor^k_{\mathsf{Trigger\_P}} \end{aligned}$$

Translation of primitives is discussed in Section 8.

The translation function $\lfloor\cdot\rfloor^k_{\mathsf{Term}} : \hat{\mathbf{M}}_k \to \mathbf{C}^k_{\&}$ translates a trigger based on its category. Terms are translated directly into conjuncts in the basic expression language because the basic expression language does not have terms.

$$\begin{aligned} \lfloor t \in \hat{\mathbf{T}}_k \rfloor^k_{\mathsf{Term}} &= \lfloor t \rfloor^k_{\mathsf{Trigger}} \\ \lfloor p \in \hat{\mathbf{P}}_k \rfloor^k_{\mathsf{Term}} &= \lfloor p \rfloor^k_{\mathsf{Primitive}} \end{aligned}$$

A conjunctive term is either a parenthesized condition or a term. The translation function $\lfloor\cdot\rfloor^k_{\mathsf{and}} : \hat{\mathbf{C}}^k_{\mathsf{and}} \to \mathbf{C}^k_{\&}$ evaluates as a straightforward translation:

$$\begin{aligned} \lfloor (\ c\ ) \rfloor^k_{\mathsf{and}} &= (\ \Vert\ \lfloor c \rfloor^k_{\mathsf{Cond}}\ \Vert\ ) \\ \lfloor m \in \hat{\mathbf{M}}_k \rfloor^k_{\mathsf{and}} &= \lfloor m \rfloor^k_{\mathsf{Term}} \end{aligned}$$

[DISTRIBUTION STATEMENT A] Approved for public release and unlimited distribution.

A disjunctive term is a possible conjunction and is translated by the function $\lfloor\cdot\rfloor^k_{\mathsf{xor}} : \hat{\mathbf{C}}^k_{\mathsf{xor}} \to \mathbf{C}^k_+$ in the expected manner:

$$\begin{array}{rcl} \lfloor c_1 \texttt{ and } c_2\rfloor^k_{\mathsf{xor}} & = & \lfloor c_1\rfloor^k_{\mathsf{xor}} \parallel \ \& \ \parallel \lfloor c_2\rfloor^k_{\mathsf{xor}} \\ \lfloor c\rfloor^k_{\mathsf{xor}} & = & \lfloor c\rfloor^k_{\mathsf{and}} \end{array}$$

Finally, a full condition is a possible disjunction and is translated by the semantic function $\lfloor\cdot\rfloor^k_{\mathsf{Cond}} : \hat{\mathbf{C}}_k \to \mathbf{C}_k$. As mentioned previously, the EMV2 specification includes both the `or` operator as inclusive or and the `xor` operator as an exclusive or. The exclusive or operator is defined as follows:

> If each port is referenced by itself in a separate alternative transition condition, i.e., `port1{BadValue} xor port2{BadValue}`, then the transition condition is satisfied if `port1` has an error propagation present and `port2` does not have an error propagation present, and vice versa, *but is not satisfied when both ports have an error propagation present (exclusive **or** of alternatives).* [27, E.8.2.(8)] [emphasis added]

The simple language does not have an exclusive or operator; instead the translation takes advantage of the logical equivalence

$$a \text{ xor } b = (a \wedge \neg b) \vee (\neg a \wedge b)$$

As with $\mathsf{satisfiedBy}_\square$, a silenced term is taken to mean a term with no propagation. Specifically, the propagations used by each side of the disjunction are computed using $\mathbf{specified}_k$, and then $\mathbf{silence}_k$ is used to generate the appropriate conjunction of empty propagations. Unlike with $\mathsf{satisfiedBy}_\square$, the set of specified propagations is used directly without being converted to an inverse set.

$$\begin{array}{rcl} \lfloor c_1 \texttt{ xor } c_2\rfloor^k_{\mathsf{Cond}} & = & (\parallel \lfloor c_1\rfloor^k_{\mathsf{Cond}} \parallel \& \parallel n_2 \parallel) \ + \ (\parallel n_1 \parallel \& \parallel \lfloor c_2\rfloor^k_{\mathsf{Cond}} \parallel) \\ & & \text{where} \begin{cases} n_1 = \mathbf{silence}_k(\mathbf{specified}_k(c_1)) \\ n_2 = \mathbf{silence}_k(\mathbf{specified}_k(c_2)) \end{cases} \\ \lfloor c_1 \texttt{ or } c_2\rfloor^k_{\mathsf{Cond}} & = & \lfloor c_1\rfloor^k_{\mathsf{Cond}} \parallel \ + \ \parallel \lfloor c_2\rfloor^k_{\mathsf{Cond}} \\ \lfloor c\rfloor^k_{\mathsf{Cond}} & = & \lfloor c\rfloor^k_{\mathsf{xor}} \end{array}$$

7.3 Example: Translation and Interpretation

The example in Listing 7.2 helps demonstrate the translation of the exclusive or and generally demonstrates applying the semantic functions. Consider the condition of transition `t1`. Let k be the component instance for the instantiation of `Example2.i`; let $\square$ be the associated semantic component. Ignoring the implicit propagation points, the set of propagation point references is $\mathbf{R}^k_{\mathsf{PP}} = \{\texttt{in1}, \texttt{in2}, \texttt{in3}, \texttt{in4}\}$.

```
1  system Example2
2  features
3    in1: in event port;
4    in2: in event port;
5    in3: in event port;
6    in4: in event port;
7  annex EMV2 {**
8    use types Q;
9    use behavior Q::Simple_Behavior;
10
```


[DISTRIBUTION STATEMENT A] Approved for public release and unlimited distribution.

```
    error propagations
      in1: in propagation {T};
      in2: in propagation {T};
      in3: in propagation {T};
      in4: in propagation {T};
    end propagations;

    component error behavior
    transitions
      t1: S0 -[in1{T} and in2{T} xor in3{T} and in4{T}]-> S3;
    end component;
  **};
  end Example2;

  system implementation Example2.i
  end Example2.i;
```

Listing 7.2: An Example Using Exclusive Or

The meaning of the condition begins with it being considered as an alternative transition condition and thus processed by $\mathsf{satisfiedBy}_{\square}$:

$$\begin{aligned} & \mathsf{satisfiedBy}_{\square}(\texttt{in in1\{T\} and in in2\{T\} xor in in3\{T\} and in in4\{T\}}) \\ = \; & [\![\lfloor \texttt{in in1\{T\} and in in2\{T\} xor in in3\{T\} and in in4\{T\}} \rfloor^{k}_{\mathsf{Cond}}]\!]^{\square}_{\mathsf{Cond}} \end{aligned}$$

The translation of the condition `in in1{T} and in in2{T} xor in in3{T} and in in4{T}` is

$$\begin{aligned} & \lfloor \texttt{in in1\{T\} and in in2\{T\} xor in in3\{T\} and in in4\{T\}} \rfloor^{k}_{\mathsf{Cond}} \\ = \; & (\, \| \, \lfloor \texttt{in in1\{T\} and in in2\{T\}} \rfloor^{k}_{\mathsf{Cond}} \, \| \, \& \, \| \\ & \mathbf{silence}_k(\mathbf{specified}_k(\texttt{in in3\{T\} and in in4\{T\}})) \, \| \,) \; + \; (\, \| \\ & \mathbf{silence}_k(\mathbf{specified}_k(\texttt{in in1\{T\} and in in2\{T\}})) \, \| \, \& \, \| \\ & \lfloor \texttt{in in3\{T\} and in in4\{T\}} \rfloor^{k}_{\mathsf{Cond}} \, \| \,) \\ = \; & (\, \| \, \lfloor \texttt{in in1\{T\} and in in2\{T\}} \rfloor^{k}_{\mathsf{Cond}} \, \| \, \& \, \| \\ & \mathbf{silence}_k(\{(\mathsf{in}, \texttt{in3}), (\mathsf{in}, \texttt{in4})\}) \, \| \,) \; + \; (\, \| \\ & \mathbf{silence}_k(\{(\mathsf{in}, \texttt{in1}), (\mathsf{in}, \texttt{in2})\}) \, \| \, \& \, \| \\ & \lfloor \texttt{in in3\{T\} and in in4\{T\}} \rfloor^{k}_{\mathsf{Cond}} \, \| \,) \\ = \; & (\, \| \, \lfloor \texttt{in in1\{T\} and in in2\{T\}} \rfloor^{k}_{\mathsf{Cond}} \, \| \, \& \, \| \\ & \texttt{in in3 \{noerror\}} \, \& \, \texttt{in in4 \{noerror\}} \, \| \,) \; + \; (\, \| \\ & \texttt{in in1 \{noerror\}} \, \& \, \texttt{in in2 \{noerror\}} \, \| \, \& \, \| \\ & \lfloor \texttt{in in3\{T\} and in in4\{T\}} \rfloor^{k}_{\mathsf{Cond}} \, \| \,) \\ = \; & (\, \| \, \lfloor \texttt{in in1\{T\}} \rfloor^{k}_{\mathsf{xor}} \, \| \, \& \, \| \, \lfloor \texttt{in in2\{T\}} \rfloor^{k}_{\mathsf{xor}} \, \| \, \& \, \| \\ & \texttt{in in3 \{noerror\}} \, \& \, \texttt{in in4 \{noerror\}} \, \| \,) \; + \; (\, \| \\ & \texttt{in in1 \{noerror\}} \, \& \, \texttt{in in2 \{noerror\}} \, \| \, \& \, \| \\ & \lfloor \texttt{in in3\{T\}} \rfloor^{k}_{\mathsf{xor}} \, \| \, \& \, \| \, \lfloor \texttt{in in4\{T\}} \rfloor^{k}_{\mathsf{xor}} \, \| \,) \\ = \; & (\, \| \, \texttt{in in1\{T\}} \, \| \, \& \, \| \, \texttt{in in2\{T\}} \, \| \, \& \, \| \\ & \texttt{in in3 \{noerror\}} \, \& \, \texttt{in in4 \{noerror\}} \, \| \,) \; + \; (\, \| \\ & \texttt{in in1 \{noerror\}} \, \& \, \texttt{in in2 \{noerror\}} \, \| \, \& \, \| \\ & \texttt{in in3\{T\}} \, \| \, \& \, \| \, \texttt{in in4\{T\}} \, \| \,) \\ = \; & (\texttt{in in1\{T\}} \, \& \, \texttt{in in2\{T\}} \, \& \, \texttt{in in3 \{noerror\}} \, \& \, \texttt{in in4 \{noerror\}}) + \\ & (\texttt{in in1 \{noerror\}} \, \& \, \texttt{in in2 \{noerror\}} \, \& \, \texttt{in in3\{T\}} \, \& \, \texttt{in in4\{T\}}) \end{aligned}$$

[DISTRIBUTION STATEMENT A] Approved for public release and unlimited distribution.

Evaluation of the basic expression condition yields the following:

$$
\begin{aligned}
& [\![(\texttt{in in1\{T\}} \;\&\; \texttt{in in2\{T\}} \;\&\; \texttt{in in3 \{noerror\}} \;\&\; \texttt{in in4 \{noerror\}}) + \\
& (\texttt{in in1 \{noerror\}} \;\&\; \texttt{in in2 \{noerror\}} \;\&\; \texttt{in in3\{T\}} \;\&\; \texttt{in in4\{T\}})]\!]^{\square}_{\mathsf{Cond}} \\
= \;& [\![(\texttt{in in1\{T\}} \;\&\; \texttt{in in2\{T\}} \;\&\; \texttt{in in3 \{noerror\}} \;\&\; \texttt{in in4 \{noerror\}})]\!]^{\square}_{\mathsf{Cond}} \cup \\
& [\![(\texttt{in in1 \{noerror\}} \;\&\; \texttt{in in2 \{noerror\}} \;\&\; \texttt{in in3\{T\}} \;\&\; \texttt{in in4\{T\}})]\!]^{\square}_{\mathsf{Cond}} \\
= \;& [\![\texttt{in in1\{T\}}]\!]^{\square}_{+} \cap [\![\texttt{in in2\{T\}} \;\&\; \texttt{in in3 \{noerror\}} \;\&\; \texttt{in in4 \{noerror\}}]\!]^{\square}_{+} \cup \\
& [\![\texttt{in in1 \{noerror\}}]\!]^{\square}_{+} \cap [\![(\texttt{in in2 \{noerror\}} \;\&\; \texttt{in in3\{T\}} \;\&\; \texttt{in in4\{T\}}]\!]^{\square}_{+} \\
= \;& [\![\texttt{in in1\{T\}}]\!]^{\square}_{+} \cap [\![\texttt{in in2\{T\}}]\!]^{\square}_{+} \cap [\![\texttt{in in3 \{noerror\}} \;\&\; \texttt{in in4 \{noerror\}}]\!]^{\square}_{+} \cup \\
& [\![\texttt{in in1 \{noerror\}}]\!]^{\square}_{+} \cap [\![(\texttt{in in2 \{noerror\}}]\!]^{\square}_{+} \cap [\![\texttt{in in3\{T\}} \;\&\; \texttt{in in4\{T\}}]\!]^{\square}_{+} \\
= \;& [\![\texttt{in in1\{T\}}]\!]^{\square}_{+} \cap [\![\texttt{in in2\{T\}}]\!]^{\square}_{+} \cap [\![\texttt{in in3 \{noerror\}}]\!]^{\square}_{+} \cap [\![\texttt{in in4 \{noerror\}}]\!]^{\square}_{+} \cup \\
& [\![\texttt{in in1 \{noerror\}}]\!]^{\square}_{+} \cap [\![(\texttt{in in2 \{noerror\}}]\!]^{\square}_{+} \cap [\![\texttt{in in3\{T\}}]\!]^{\square}_{+} \cap [\![\texttt{in in4\{T\}}]\!]^{\square}_{+}
\end{aligned}
$$

Each propagation trigger t evaluates to a set of environments via $[\![t]\!]^{\square}_{+} = [\![t]\!]^{\square}_{\mathsf{Trigger_P}}$:

$$
\begin{aligned}
= \;& \{\gamma \in \Gamma_{\square} \mid \gamma[\mathsf{in1}] \sqsubseteq \{\mathsf{T}\}\} \cap \{\gamma \in \Gamma_{\square} \mid \gamma[\mathsf{in2}] \sqsubseteq \{\mathsf{T}\}\} \cap \\
& \{\gamma \in \Gamma_{\square} \mid \gamma[\mathsf{in3}] = \epsilon_{\mathsf{Type}}\} \cap \{\gamma \in \Gamma_{\square} \mid \gamma[\mathsf{in4}] = \epsilon_{\mathsf{Type}}\} \cup \\
& \{\gamma \in \Gamma_{\square} \mid \gamma[\mathsf{in1}] = \epsilon_{\mathsf{Type}}\} \cap \{\gamma \in \Gamma_{\square} \mid \gamma[\mathsf{in2}] = \epsilon_{\mathsf{Type}}\} \cap \\
& \{\gamma \in \Gamma_{\square} \mid \gamma[\mathsf{in3}] \sqsubseteq \{\mathsf{T}\}\} \cap \{\gamma \in \Gamma_{\square} \mid \gamma[\mathsf{in4}] \sqsubseteq \{\mathsf{T}\}\} \\
= \;& \{\gamma \in \Gamma_{\square} \mid \gamma[\mathsf{in1}] \sqsubseteq \{\mathsf{T}\}\} \cap \{\gamma \in \Gamma_{\square} \mid \gamma[\mathsf{in2}] \sqsubseteq \{\mathsf{T}\}\} \cap \\
& \{\gamma \in \Gamma_{\square} \mid \gamma[\mathsf{in3}] = \epsilon_{\mathsf{Type}} \wedge \gamma[\mathsf{in4}] = \epsilon_{\mathsf{Type}}\} \cup \\
& \{\gamma \in \Gamma_{\square} \mid \gamma[\mathsf{in1}] = \epsilon_{\mathsf{Type}}\} \cap \{\gamma \in \Gamma_{\square} \mid \gamma[\mathsf{in2}] = \epsilon_{\mathsf{Type}}\} \cap \\
& \{\gamma \in \Gamma_{\square} \mid \gamma[\mathsf{in3}] \sqsubseteq \{\mathsf{T}\} \wedge \gamma[\mathsf{in4}] \sqsubseteq \{\mathsf{T}\}\} \\
= \;& \{\gamma \in \Gamma_{\square} \mid \gamma[\mathsf{in1}] \sqsubseteq \{\mathsf{T}\}\} \cap \{\gamma \in \Gamma_{\square} \mid \gamma[\mathsf{in2}] \sqsubseteq \{\mathsf{T}\} \wedge \gamma[\mathsf{in3}] = \epsilon_{\mathsf{Type}} \wedge \gamma[\mathsf{in4}] = \epsilon_{\mathsf{Type}}\} \cup \\
& \{\gamma \in \Gamma_{\square} \mid \gamma[\mathsf{in1}] = \epsilon_{\mathsf{Type}}\} \cap \{\gamma \in \Gamma_{\square} \mid \gamma[\mathsf{in2}] = \epsilon_{\mathsf{Type}} \wedge \gamma[\mathsf{in3}] \sqsubseteq \{\mathsf{T}\} \wedge \gamma[\mathsf{in4}] \sqsubseteq \{\mathsf{T}\}\} \\
= \;& \{\gamma \in \Gamma_{\square} \mid \gamma[\mathsf{in1}] \sqsubseteq \{\mathsf{T}\} \wedge \gamma[\mathsf{in2}] \sqsubseteq \{\mathsf{T}\} \wedge \gamma[\mathsf{in3}] = \epsilon_{\mathsf{Type}} \wedge \gamma[\mathsf{in4}] = \epsilon_{\mathsf{Type}}\} \cup \\
& \{\gamma \in \Gamma_{\square} \mid \gamma[\mathsf{in1}] = \epsilon_{\mathsf{Type}} \wedge \gamma[\mathsf{in2}] = \epsilon_{\mathsf{Type}} \wedge \gamma[\mathsf{in3}] \sqsubseteq \{\mathsf{T}\} \wedge \gamma[\mathsf{in4}] \sqsubseteq \{\mathsf{T}\}\}
\end{aligned}
$$

Now the environments can be fully expanded based on the facts

- $\mathcal{I}_{\square} = \{\mathsf{in1}, \mathsf{in1}, \mathsf{in1}, \mathsf{in1}, \mathsf{evt}\}$
- $\mathcal{V}_{\square} = \{\mathsf{X}\}$
- $\mathbf{D}_{\triangleright \mathsf{PP}}(f) = \{\mathsf{T}\}$ for each f in $\{\texttt{in1},\ \texttt{in2},\ \texttt{in3},\ \texttt{in4}\}$.

$$
\begin{aligned}
= \;& \{\langle \mathsf{in1} \mapsto \mathsf{T}, \mathsf{in2} \mapsto \mathsf{T}, \mathsf{in3} \mapsto \epsilon_{\mathsf{Type}}, \mathsf{in4} \mapsto \epsilon_{\mathsf{Type}}, \mathsf{event} \mapsto \epsilon_{\mathsf{Event}}\rangle, \\
& \langle \mathsf{in1} \mapsto \mathsf{T}, \mathsf{in2} \mapsto \mathsf{T}, \mathsf{in3} \mapsto \epsilon_{\mathsf{Type}}, \mathsf{in4} \mapsto \epsilon_{\mathsf{Type}}, \mathsf{event} \mapsto \mathsf{X}\rangle\} \cup \\
& \{\langle \mathsf{in1} \mapsto \epsilon_{\mathsf{Type}}, \mathsf{in2} \mapsto \epsilon_{\mathsf{Type}}, \mathsf{in3} \mapsto \mathsf{T}, \mathsf{in4} \mapsto \mathsf{T}, \mathsf{event} \mapsto \epsilon_{\mathsf{Event}}\rangle, \\
& \langle \mathsf{in1} \mapsto \epsilon_{\mathsf{Type}}, \mathsf{in2} \mapsto \epsilon_{\mathsf{Type}}, \mathsf{in3} \mapsto \mathsf{T}, \mathsf{in4} \mapsto \mathsf{T}, \mathsf{event} \mapsto \mathsf{X}\rangle\} \\
= \;& \{\langle \mathsf{in1} \mapsto \mathsf{T}, \mathsf{in2} \mapsto \mathsf{T}, \mathsf{in3} \mapsto \epsilon_{\mathsf{Type}}, \mathsf{in4} \mapsto \epsilon_{\mathsf{Type}}, \mathsf{event} \mapsto \epsilon_{\mathsf{Event}}\rangle, \\
& \langle \mathsf{in1} \mapsto \mathsf{T}, \mathsf{in2} \mapsto \mathsf{T}, \mathsf{in3} \mapsto \epsilon_{\mathsf{Type}}, \mathsf{in4} \mapsto \epsilon_{\mathsf{Type}}, \mathsf{event} \mapsto \mathsf{X}\rangle, \\
& \langle \mathsf{in1} \mapsto \epsilon_{\mathsf{Type}}, \mathsf{in2} \mapsto \epsilon_{\mathsf{Type}}, \mathsf{in3} \mapsto \mathsf{T}, \mathsf{in4} \mapsto \mathsf{T}, \mathsf{event} \mapsto \epsilon_{\mathsf{Event}}\rangle, \\
& \langle \mathsf{in1} \mapsto \epsilon_{\mathsf{Type}}, \mathsf{in2} \mapsto \epsilon_{\mathsf{Type}}, \mathsf{in3} \mapsto \mathsf{T}, \mathsf{in4} \mapsto \mathsf{T}, \mathsf{event} \mapsto \mathsf{X}\rangle\}
\end{aligned}
$$

Thus, the alternate condition expression `in in1{T} and in in2{T} xor in in3{T} and in in4{T}` describes the environments in which

- both `in1` and `in2` are propagating `T` and both `in3` and `in4` are silent
- both `in1` and `in2` are silent and both `in3` and `in4` are propagating `T`

Note that it is irrelevant whether event `X` has occurred or not.

[DISTRIBUTION STATEMENT A] Approved for public release and unlimited distribution.

8 Condition Expressions, Part 3: Translation of Primitives

The logical primitives are translated into expressions based on conjunction, disjunction in the simple language, and a basic concept of negating a trigger. The translations all rely on determining combinations of set elements. Let $\mathsf{choose}(k, \{x_1, \ldots, x_n\})$ be the function that returns the set of all k-combinations of $\{x_1, \ldots, x_n\}$. (The particular algorithm used to compute choose is not important and is not further discussed.). For example,

$$\begin{array}{lcl} \mathsf{choose}(2, \{1,2,3,4\}) & = & \{\{1,2\},\{1,3\},\{1,4\},\{2,3\},\{2,4\},\{3,4\}\} \\ \mathsf{choose}(3, \{1,2,3,4\}) & = & \{\{1,2,3\},\{1,2,4\},\{1,3,4\},\{2,3,4\}\} \end{array}$$

Below, the translation function $\lfloor\cdot\rfloor^k_{\mathsf{Primitive}} : \hat{\mathbf{P}}_k \to \mathbf{C}^k_{\&}$ is defined. The translations are more complex than might be expected because of the strict requirements of triggers that *must not be satisfied*. The simplest case `ormore` is presented first.

8.1 The Primitive `ormore`

The standard defines `ormore`:

> If the alternative transition condition specifies `1 ormore (port1{BadValue}, port2{BadValue})`, then the condition is satisfied if error propagations are present on either port or on both ports. In other words, if one conditions or more are true. [27, E.8.2.(8)]

From this can be extrapolated the definition that `k ormore ...` is satisfied if k or more conditions (i.e., triggers) are satisfied. Importantly, *no triggers are required to be unsatisfied.* Let $\mathbf{DNFChoose}_k(kk, \{t_1, \ldots, t_n\})$, where $t_i \in \hat{\mathbf{T}}_k$, be the function that generates disjunctive normal form from the kk combinations of triggers; see Figure 8.1. A strict literal translation of kk `ormore` $(t_1, \ldots, t_n)$ is

$$\begin{array}{ll} & \mathbf{DNFChoose}_k(kk, \{t_1, \ldots, t_n\}) \\ + & \mathbf{DNFChoose}_k(kk+1, \{t_1, \ldots, t_n\}) \\ + & \cdots \\ + & \mathbf{DNFChoose}_k(n, \{t_1, \ldots, t_n\}) \end{array}$$

It is easy to see, however, that once at least kk triggers are satisfied, the subexpression $\mathbf{DNFChoose}_k(kk, \{t_1, \ldots, t_n\})$ will always be satisfied—indeed one of its conjuncts will have to be satisfied—and the condition as a whole will be satisfied. Thus, the translation can just be simplified to $\mathbf{DNFChoose}_k(kk, \{t_1, \ldots, t_n\})$.

The translation of `ormore` is formally defined in Figure 8.1.

- $\mathbf{DNFChoose}_k : \mathbb{N} \times 2^{\hat{\mathbf{T}}_k} \to \mathbf{C}_k$ is described above.
- $\mathbf{DNF}_k : 2^{2^{\hat{\mathbf{T}}_k}} \to \mathbf{C}_k$ constructs a syntactic disjunction by first building a disjunction from each mathematical set s_o of triggers representing a combination of kk triggers.
- $\mathbf{conjoin}_k : 2^{\hat{\mathbf{T}}_k} \to \mathbf{C}^k_+$ builds a syntactic conjunction from a mathematical set of triggers.
- $\mathbf{triggers}_k : \hat{\mathbf{T}}^k_{\mathsf{List}} \to 2^{\hat{\mathbf{T}}_k}$ converts a syntactic sequence of triggers to a mathematical set of triggers.

[DISTRIBUTION STATEMENT A] Approved for public release and unlimited distribution.

$$\lfloor kk \texttt{ ormore (} \mathit{Triggers} \texttt{)} \rfloor^k_{\mathsf{Primitive}} \quad = \quad (\ \| \ \mathbf{DNFChoose}_k([\![kk]\!]_{\mathsf{N}}, \mathbf{triggers}_k(\mathit{Triggers}))\ \| \ )$$

$$\mathbf{DNFChoose}_k(kk, \{t_1, \ldots, t_n\}) \quad = \quad \mathbf{DNF}_k(\mathsf{choose}(kk, \{t_1, \ldots, t_n\}))$$

$$\begin{array}{rcl} \mathbf{DNF}_k(\{s\}) & = & \mathbf{conjoin}_k(s) \\ \mathbf{DNF}_k(\{s_1, \ldots, s_m\}) & = & \mathbf{conjoin}_k(s_1)\ \| \ +\ \| \ \mathbf{DNF}_k(\{s_2, \ldots s_m\}) \end{array}$$

$$\begin{array}{rcl} \mathbf{conjoin}_k(\{t\}) & = & \lfloor t \rfloor^k_{\mathsf{Trigger}} \\ \mathbf{conjoin}_k(\{t_1, \ldots, t_m\}) & = & \lfloor t_1 \rfloor^k_{\mathsf{Trigger}}\ \| \ \&\ \| \ \mathbf{conjoin}_k(\{t_2, \ldots t_m\}) \end{array}$$

$$\begin{array}{rcl} \mathbf{triggers}_k(t) & = & \{t\} \\ \mathbf{triggers}_k(t\ ,\ \mathit{Triggers}) & = & \{t\} \cup \mathbf{triggers}_k(\mathit{Triggers}) \end{array}$$

Figure 8.1: The Translation of the Primitive `ormore`

8.2 The Primitive `orless`

The standard defines `orless`:

> If the alternative transition condition specifies `1 orless (port1{BadValue}, port2{BadValue})`, then the condition is satisfied if at most one error propagation is present on either port or on both ports. In other words, if one conditions [sic] or less is true.

Based on the translation for `ormore`, it is easy to conclude naively that the translation for `orless` should be

$$\mathbf{DNFChoose}_k(1, \{t_1, \ldots, t_n\}) + \cdots + \mathbf{DNFChoose}_k(k, \{t_1, \ldots, t_n\})$$

However, this is incorrect because it *does not* enforce that *no more than* k are satisfied. Consider, for example, `2 ormore`(A, B, C, D), where A, B, C, and D are trigger expressions. The above would translate the `ormore` expression to

$$A + B + C + D + AB + AC + AD + BC + BD + DE$$

This may look reasonable, but it is not: this expression is satisfied, for example, when all four of A, B, C, and D, are satisfied. *But this is not what is required*: the expression must be *unsatisfied* when three or four of A, B, C, or D are satisfied. In this case, the translation of `2 ormore`(A, B, C, D) should be

$$\begin{array}{c} A\&\overline{B}\&\overline{C}\&\overline{D} + \overline{A}\&B\&\overline{C}\&\overline{D} + \overline{A}\&\overline{B}\&C\&\overline{D} + \overline{A}\&\overline{B\&\overline{C}}\&D + A\&B\&\overline{C}\&\overline{D} + \\ A\&\overline{B}\&C\&\overline{D} + A\&\overline{B}\&\overline{C}\&D + \overline{A}\&B\&C\&\overline{D} + \overline{A}\&B\&\overline{C}\&D + \overline{A}\&\overline{B}\&C\&D \end{array}$$

which simplifies to

$$\begin{array}{c} \overline{A}\&\overline{B}\&(C + D) + \overline{A}\&\overline{C}\&(B + D) + \overline{A}\&\overline{D}\&(B + C) + \\ \overline{B}\&\overline{C}\&(A + D) + \overline{B}\&\overline{D}\&(A + C) + \overline{C}\&\overline{D}\&(A + B) \end{array}$$

Here $\overline{X}$ is the negation of trigger X. The exact meaning of this for triggers is given below; for now, it is sufficient to (1) consider the normal logical meaning of negation and (2) point out that negation is not silencing (see Section 7.1 and Section 8.3.1).

The above can be generalized: k `orless` $(t_1, \ldots, t_n\ )$ means $n - k$ of the triggers must be unsatisfied and at least one of the remaining triggers is satisfied. The formal translation is in Figure 8.2:

[DISTRIBUTION STATEMENT A] Approved for public release and unlimited distribution.

$$
\begin{array}{rcl}
\lfloor kk\ \texttt{orless}\ (\ \mathit{Triggers}\ )\rfloor^{k}_{\mathsf{Primitive}} & = & (\ \|\ \mathbf{buildOrless}_k(\mathsf{choose}(n - [\![kk]\!]_{\mathsf{N}}, trg), trg)\ \|\ ) \\
& \text{where} & trg = \mathbf{triggers}(\mathit{Triggers}) \\
& & n = |trg|
\end{array}
$$

$$
\begin{array}{rcl}
\mathbf{buildOrless}_k(\{s\}, trg) & = & (\ \|\ \mathbf{negateSet}_k(s)\ \|\ )\ \& \\
& & (\ \|\ \mathbf{disjoin}_k(trg \setminus s)\ \|\ ) \\
\mathbf{buildOrless}_k(\{s_1, \ldots, s_m\}, trg) & = & \mathbf{buildOrless}_k(\{s_1\}, trg)\ \|\ +\ \| \\
& & \mathbf{buildOrless}_k(\{s_2, \ldots, s_m\}, trg)
\end{array}
$$

$$
\begin{array}{rcl}
\mathbf{disjoin}_k(\{t\}) & = & \lfloor t \rfloor^{k}_{\mathsf{Trigger}} \\
\mathbf{disjoin}_k(\{t_1, \ldots, t_m\}) & = & \lfloor t_1 \rfloor^{k}_{\mathsf{Trigger}}\ \|\ +\ \|\ \mathbf{disjoin}_k(\{t_2, \ldots t_m\})
\end{array}
$$

$$
\begin{array}{rcl}
\mathbf{negateSet}_k(\{t\}) & = & (\ \|\ \mathbf{negate}_k(t)\ \|\ ) \\
\mathbf{negateSet}_k(\{t_1, \ldots, t_m\}) & = & \mathbf{negateSet}_k(\{t_1\})\ \|\ \&\ \|\ \mathbf{negateSet}_k(\{t_2, \ldots, t_m\})
\end{array}
$$

Figure 8.2: The Translation of the Primitive `orless`

- Recall that choose evaluates to a set of sets so that variables s_i are sets.
- Function $\mathbf{buildOrLess}_{k:}2^{2^{\hat{\mathbf{T}}_k}} \times 2^{\hat{\mathbf{T}}_k} \rightarrow \mathbf{C}^k_+$ builds the overall disjunction based on the choices of sets of $(n-k)$ triggers that must be *unsatisfied*.
- Function $\mathbf{disjoin}_{k:}2^{\hat{\mathbf{T}}_k} \rightarrow \mathbf{C}_k$ simply builds a disjunction of triggers from a set of triggers.
- Function $\mathbf{negateSet}_k : 2^{\hat{\mathbf{T}}_k} \rightarrow \mathbf{C}^k_+$ builds a conjunction of the *negation* of all the triggers in the input set.

The next section defines $\mathbf{negate}_k$—used by $\mathbf{negateSet}_k$—and the semantics of negating a trigger. To complete this section, it is enough to say that the negation of a trigger should be satisfied if and only if *the trigger is not satisfied*. For example, the trigger `in in1{T}` is satisfied by the set of environments

$$\{\gamma \in \Gamma_\square \mid \gamma[\mathsf{in1}] \sqsubseteq \{\mathsf{T}\}\}$$

The negation of the trigger `in1{T}` is thus

$$\{\gamma \in \Gamma_\square \mid \gamma[\mathsf{in1}] \not\sqsubseteq \{\mathsf{T}\}\}$$

8.3 Negation of Triggers

From a mathematical point of view, the negation of a trigger is straightforward; see the above example. It is illuminating, however, to present a more "positive" denotation of negation. Fortunately, the universe of possible values for environment fields is both known and finite. For example, the universe of values for an in propagation is given by the declared in-propagation set. So if

1. the propagation point `in1`, above, is declared to have the possible in propagations of `X`, `Q`, and `T`
2. error types `X`, `Q`, and `T` are in separate hierarchies (cf. type containment)

then the negation of the trigger `in1{T}` can be denoted as

$$\{\gamma \in \Gamma_\square \mid \gamma[\mathsf{in1}] \sqsubseteq \{\mathsf{X}, \mathsf{Q}\} \vee \gamma[\mathsf{in1}] = \epsilon_{\mathsf{Type}}\}$$

The set includes the possibility that the propagation point has no propagation: the value ϵ_{Type}. Working backward, this set of environments can be seen to come from the basic expression `in in1{X,Q} + in in1{noerror}`.

[DISTRIBUTION STATEMENT A] Approved for public release and unlimited distribution.

$$
\begin{array}{rcl}
\mathbf{negate}_k(e \in \mathbf{D}^k_{\mathsf{Event}}) & = & \mathbf{disjoinEvents}_k(\mathbf{D}^k_{\mathsf{Event}} \setminus \{e\}) \ \| \ \texttt{noevent} \\
\mathbf{negate}_k(\texttt{in}\ f\ tse) & = & \mathbf{constructLocal}(f,\ ne,\ inverted) \\
\multicolumn{3}{r}{\text{where } (ne,\ inverted) = \mathbf{invertTSE}(\mathbf{Set}(\mathbf{D}_{\triangleright\mathsf{PP}}(f)), tse)} \\
\mathbf{negate}_k(\texttt{out}\ f\ tse) & = & \mathbf{constructSub}(f,\ ne,\ inverted) \\
\multicolumn{3}{r}{\text{where } (ne,\ inverted) = \mathbf{invertTSE}(\mathbf{Set}(\mathbf{D}_{\mathsf{PP}\triangleright}(f)), tse)}
\end{array}
$$

$$
\begin{array}{rcl}
\mathbf{invertTSE}_k(d, \epsilon) & = & (\mathsf{true}, \emptyset) \\
\mathbf{invertTSE}_k(\{elements\}, \{\texttt{noerror}\}) & = & (\mathsf{false}, \mathbf{types}(elements)) \\
\mathbf{invertTSE}_k(\{elements\}, ts \in \mathbf{S}) & = & (\mathsf{true}, \mathbf{invert}_k(\mathbf{types}(elements), \mathbf{Set}(ts)))
\end{array}
$$

$$
\begin{array}{rcl}
\mathbf{types}(t) & = & \{t\} \\
\mathbf{types}(t\ ,\ elements) & = & \{t\} \cup \mathbf{types}(elements)
\end{array}
$$

$$
\begin{array}{rcl}
\mathbf{disjoinEvents}_k(\{e\}) & = & \texttt{event}\ e \\
\mathbf{disjoinEvents}_k(\{e_1, \ldots, e_m\}) & = & \mathbf{disjoinEvents}_k(\{e_1\}) \ \| \ + \ \| \\
 & & \mathbf{disjoinEvents}_k(\{e_2, \ldots, e_m\})
\end{array}
$$

$$
\begin{array}{rcl}
\mathbf{constructLocal}(f, \mathsf{false}, \{t_1, \ldots, t_n\}) & = & \texttt{in} \ \| \ f \ \| \ \{ \ \| \ t_1 \ \| \ , \ \| \cdots \| \ , \ \| \ t_n \ \| \ \} \\
\mathbf{constructLocal}(f, \mathsf{true}, \emptyset) & = & \texttt{in} \ \| \ f \ \| \ \{\texttt{noerror}\} \\
\mathbf{constructLocal}(f, \mathsf{true}, \{t_1, \ldots, t_n\}) & = & \texttt{in} \ \| \ f \ \| \ \{\texttt{noerror}\} \ + \ \texttt{in} \ \| \\
 & & f \ \| \ \{ \ \| \ t_1 \ \| \ , \ \| \cdots \| \ , \ \| \ t_n \ \| \ \}
\end{array}
$$

$$
\begin{array}{rcl}
\mathbf{constructSub}(f, \mathsf{false}, \{t_1, \ldots, t_n\}) & = & \texttt{out} \ \| \ f \ \| \ \{ \ \| \ t_1 \ \| \ , \ \| \cdots \| \ , \ \| \ t_n \ \| \ \} \\
\mathbf{constructSub}(f, \mathsf{true}, \emptyset) & = & \texttt{out} \ \| \ f \ \| \ \{\texttt{noerror}\} \\
\mathbf{constructSub}(f, \mathsf{true}, \{t_1, \ldots, t_n\}) & = & \texttt{out} \ \| \ f \ \| \ \{\texttt{noerror}\} \ + \ \texttt{out} \ \| \\
 & & f \ \| \ \{ \ \| \ t_1 \ \| \ , \ \| \cdots \| \ , \ \| \ t_n \ \| \ \}
\end{array}
$$

$$
\begin{array}{rcl}
\mathbf{invert}(\{t\}, ts) & = & \begin{cases} \emptyset & \text{when } t \sqsubseteq ts \\ \{t\} & \text{otherwise} \end{cases} \\
\mathbf{invert}(\{t_1, \ldots, t_n\}, ts) & = & \mathbf{invert}(\{t_1\}, ts) \cup \mathbf{invert}(\{t_2, \ldots, t_n\}, ts)
\end{array}
$$

Figure 8.3: Function to Negate a Single Trigger

[DISTRIBUTION STATEMENT A] Approved for public release and unlimited distribution.

Function $\mathbf{negate}_k : \hat{\mathbf{T}}_k \rightarrow \mathbf{C}^k_+$ in Figure 8.3 converts an EMV2 trigger into a basic expression that expresses the trigger being unsatisfied. For event triggers, this is clearly the set of all possible events except for the event named in the trigger, plus the possibility of no event at all. Because $\mathbf{D}^k_{\mathsf{Event}}$ is known from the *syntax* of the model, it is trivial for an analysis of the model to compute $\mathbf{D}^k_{\mathsf{Event}} \setminus \{e\}$ and then build the syntactic disjunction of events using $\mathbf{disjoinEvents}_k : 2^{\mathbf{D}^k_{\mathsf{Event}}} \rightarrow \mathbf{C}_{\mathsf{k}}$.

Negation of a propagation trigger proceeds based on the observations above:

1. The set of possible propagations for the propagation point is obtained from $\mathbf{D}_{\triangleright\mathsf{PP}}$ for an in propagation point or $\mathbf{D}_{\mathsf{PP}\triangleright}$ for an out propagation point. Because these sets are syntactic, they must have references resolved using the **Set** function. This is purposely done in the syntactic space to demonstrate that this translation can be performed in an analysis tool. The optional type set from the trigger is then inverted.
2. $\mathbf{invertTSE}_k : \mathbf{S} \rightarrow \mathbf{S}_{\mathsf{NoError}} \rightarrow (\mathbb{B} \times 2^{\mathbf{S}_{\mathsf{Element}}})$ determines how an optional type set should be inverted. The Boolean value indicates whether the inverse should contain {`noerror`}; the mathematical set of triggers is the inverted trigger.
   - If the type set is missing (ϵ), then the trigger is testing against all the types that could possibly be propagated, and thus the inverse is just {`noerror`}.
   - If the specified type set is {`noerror`}, then the inverse includes all the declared propagated types, but *not* {`noerror`}.
   - If a set of types is specified, then the inverse includes {`noerror`} and all the propagated types not contained in the specified set.
3. $\mathbf{types} : \mathbf{S}_{\mathsf{Elements}} \rightarrow 2^{\mathbf{S}_{\mathsf{Element}}}$ converts a syntactic list of type set elements to a mathematical set of type set elements.
4. $\mathbf{invert} : 2^{\mathbf{S}_{\mathsf{Element}}} \rightarrow \mathbf{S} \rightarrow 2^{\mathbf{S}_{\mathsf{Element}}}$ tests each possible propagated type or containment in the trigger's type set. As described in Section 3.4.1, this test can be performed in the syntactic space. A mathematical set of the types that *are not contained* in the trigger's type set is constructed as a result.
5. $\mathbf{constructLocal}_k : \mathbf{D}^k_{\mathsf{In}} \rightarrow \mathbb{B} \rightarrow 2^{\mathbf{S}_{\mathsf{Element}}} \rightarrow \mathbf{C}_{\mathsf{k}}$ and $\mathbf{constructSub}_k : \mathbf{R}^k_{\mathsf{Out}} \rightarrow \mathbb{B} \rightarrow 2^{\mathbf{S}_{\mathsf{Element}}} \rightarrow \mathbf{C}_{\mathsf{k}}$ construct the basic expression of the negated trigger.

8.3.1 Negation Versus Silencing

It should be clear that negating a trigger, above, and silencing a trigger (Section 7.1) are distinct operations. To be clear,

- a silenced trigger is satisfied by environments in which the trigger *is silent*; that is, no event is occurring, or no propagation is occurring on the propagation point.
- a negated trigger is satisfied by environments in which the trigger *is not satisfied*. This is a *larger* set of environments than those described by silencing: it includes the silent environments but also includes environments in which the trigger has other values that do not satisfy the trigger.

In this context, the semantics of the exclusive or operator `xor`, presented in Section 7.2, are revisited. Recall that the exclusive or operator is defined as follows:

> If each port is referenced by itself in a separate alternative transition condition, i.e., `port1{BadValue} xor port2{BadValue}`, then the transition condition is satisfied if `port1` has an error propagation present and *`port2` does not have an error propagation present*, and vice versa, but is not satisfied when both ports have

[DISTRIBUTION STATEMENT A] Approved for public release and unlimited distribution.

> an error propagation present (exclusive **or** of alternatives). [27, E.8.2.(8)] [emphasis added, but in a different place than before]

Section 7.2 uses silencing to define the translation of `xor` based on the wording "does not have an error propagation present." This is similar wording to the requirement from EMV2 that when a propagation trigger is by itself "then all other incoming error propagation points must not have a propagation present" [27, E.8.2.(8)]. Thus, silencing is used to translate both situations.

Exclusive or, however, is typically defined using *negation*:

$$A \text{ xor } B = (A \wedge \neg B) \vee (\neg A \wedge B)$$

In the authors' opinion, neither the wording of the EMV2 standard nor the notes from its creation are clear enough to fully determine whether exclusive or should be translated via silencing or via negation. In particular, the silencing translation may make sense when A and B, above, are *solitary trigger expressions*. In this case, A `xor` B can be thought of merging the transition conditions of two alternate transition conditions. For example, the separate transitions

```
t1: s1 -[in1{A}]-> s2
t2: s1 -[in2{C}]-> s2
```

could be written as

```
t3: s1 -[in1{A} xor in2{C}]-> s2
```

and `t3` would have the same silencing semantics as the pair `t1` and `t2`. However, silencing begins to make less sense when A and B are allowed to be arbitrary expressions. For example, what is the rationale behind silencing the propagations `in1` and `in2` from the sub-condition `in1{X} and in2{Y}` of the transition condition of `t4`?

```
t4: s1 -[(in1{X} and in2{Y}) xor FailEvent]-> s2
```

Furthermore, does it ever really make sense to silence events, as would occur during the handling of event `FailEvent` above?

To translate `xor` via negation, the negation concepts from the previous section would need to be extended to cover full expressions, and not just triggers. In the absence of certainty, this extension is not carried out here, and it is possible that such an extension may introduce new issues regarding the interpretation of expressions.[1] This section concludes, however, with a concrete example demonstrating both translations.

Assume that

- $\mathbf{D}_{\triangleright\mathsf{PP}}(\mathsf{in1}) = \{\mathsf{A}, \mathsf{B}\}$
- $\mathbf{D}_{\triangleright\mathsf{PP}}(\mathsf{in2}) = \{\mathsf{C}, \mathsf{D}\}$
- Error types `A` and `B` are in separate hierarchies.
- Error types `C` and `D` are in separate hierarchies.

Interpreting `in in1{A} xor in in2{C}` via silencing (as described in Section 7.1), yields the basic expression

$$(\texttt{in in1\{A\}} \ \& \ \texttt{in in2\{noerror\}}) \ + \ (\texttt{in in1\{noerror\}} \ \& \ \texttt{in in2\{C\}})$$

[1] The authors conjecture that such a translation can be carried out using De Morgan's laws in such a way that condition expressions can always be reduced to basic expressions over propagations, including those with `{noerror}`, events, and `noevent`—even in cases where previously negated subexpressions are negated again.

[DISTRIBUTION STATEMENT A] Approved for public release and unlimited distribution.

$$
\begin{array}{rcl}
\lfloor \texttt{all} - kk \ (\ \mathit{Triggers}\ ) \rfloor^{k}_{\mathsf{Primitive}} & = & (\ \|\ \mathbf{buildAllBut}_k(\mathsf{choose}([\![kk]\!]_{\mathsf{N}}, trg), trg)\ \|\ ) \\
 & & \text{where } trg = \mathbf{triggers}(\mathit{Triggers}) \\
\mathbf{buildAllBut}_k(\{s\}, trg) & = & (\ \|\ \mathbf{negateSet}_k(s)\ \|\ )\ \&\ (\ \| \\
 & & \mathbf{conjoin}_k(trg \setminus s)\ \|\ ) \\
\mathbf{buildAllBut}_k(\{s_1, \ldots, s_m\}, trg) & = & \mathbf{buildAllBut}_k(\{s_1\}, trg)\ \|\ +\ \| \\
 & & \mathbf{buildAllBut}_k(\{s_2, \ldots, s_m\}, trg)
\end{array}
$$

Figure 8.4: The Translation of the "All But" Primitive

Interpreting `in in1{A} xor in in2{C}` via negation, however, would yield the basic expression

$$
\begin{array}{c}
(\texttt{in in1\{A\}}\ \&\ (\texttt{in in2\{noerror\}}\ +\ \texttt{in in2\{D\}}))\ + \\
((\texttt{in in1\{noerror\}}\ +\ \texttt{in in1\{B\}})\ \&\ \texttt{in in2\{C\}})
\end{array}
$$

8.4 The Primitive `all`

The simple case is $\texttt{all}(t_1, \ldots, t_n)$. The EMV2 specification gives the exact meaning [27, E.8.2.(8)]:

> `all (port1{BadValue}, port2{BadValue}, port3{BadValue})` is equivalent to the conjunction `port1{BadValue} and port2{BadValue} and port3{BadValue}`

The translation can be described using machinery already developed above:

$$
\lfloor \texttt{all}\ (\ \mathit{Triggers}\ ) \rfloor^{k}_{\mathsf{Primitive}} \quad = \quad \mathbf{conjoin}_k(\mathbf{triggers}_k(\mathit{Triggers}))
$$

For the general case of $\texttt{all} - kk\ (\ \mathit{Triggers}\ )$, the specification states [27, E.8.2.(8)] (emphasis added):

> if the alternative transition specifies `all { x (port1{BadValue}, port2{BadValue}, port3{BadValue})`, then the condition is satisfied if *all but x of them satisfies the condition*, for x an integer

Specifically, it is the case that exactly kk triggers are unsatisfied, and exactly $n - kk$ are satisfied. This too can be expressed using machinery described above via the function $\mathbf{buildAllBut}_k{:}2^{\hat{\mathbf{T}}_k} \times 2^{\hat{\mathbf{T}}_k} \rightarrow \mathbf{C}^k_+$. The translation is shown in Figure 8.4.

[DISTRIBUTION STATEMENT A] Approved for public release and unlimited distribution.

9 Component Behavior Automata

The preceding sections develop enough semantic machinery to enable a formal description of the error behavior of a component, which is a surprisingly complex entity. It has properties of many types of automata:

- A component's error behavior contains states and state transitions. This suggests that the behavior is a finite automaton. Furthermore, "the set of outgoing error behavior transitions from the same source error behavior state to different target states must be unambiguous for a given component" [27, E.8.2.(12)], so the behavior should be a *deterministic finite automaton* [12, §2.2].
- A component, however, can emit an error propagation as the result of a transition, which is similar to the behavior of a *Mealy machine* [18]. However, the behavior automata semantics differ slightly from that of a Mealy machine.
- Transitions can be *branched* [27, E.8.2.(5)], meaning that a transition leads to one of a set of states with a given probability. The behavior is thus similar to a *probabilistic automata* [23, 32].
- Most interestingly, the transitions are not directly driven by the input alphabet but rather, as already explored in Sections 6–8, are described using *expressions* over events and error propagations. This is similar to a *symbolic automata* [9, 33].

Because of the complexity of behavior automata, the description of them is built incrementally, with the bulk of this section being devoted to the semantics of transitions and propagations.

9.1 The Basic Automaton

A deterministic finite automaton (DFA) is defined in Hopcroft and Ullman [12] as a 5-tuple $(Q, \Sigma, \delta, q_0, F)$, where

- Q is a finite set of states. Symbol q is an element of Q.
- Σ is a finite input alphabet. Symbol a is an element of Σ.
- $\delta \in Q \times \Sigma \to Q$ is the transition function.
- $q_0 \in Q$ is the initial state.
- $F \subseteq Q$ is the set of final states.

In summary, the automaton starts in state q_0. When the automaton is in state q and receives input a, the transition function determines the new state of the automaton, if any: $q' = \delta(q, a)$. Because δ is a function, the automaton is *deterministic*: there is only a single value for each (q, a), and thus the exact transition can always be determined. The automaton terminates when the current state $q \in F$.

For an automaton describing the error behavior of a component, sets from the preceding sections define some of this information. Specifically, for a syntactic component k with $\square = [\![k]\!]_{\mathsf{K}}$, let $\mathcal{B}_\square$ be the behavior automaton for $\square$.[1] The exact form of $\mathcal{B}_\square$ evolves over the next few sections; initially, based on the structure of a DFA, it is $\mathcal{B}_\square = (Q, \Sigma, \delta, q_0)$, where

- $Q = \mathcal{Q}_\square$

[1] B for behavior.

[DISTRIBUTION STATEMENT A] Approved for public release and unlimited distribution.

- $\Sigma = \Gamma_\square$
- $q_0 = [\![q_0^k]\!]^\square_{\mathsf{State}}$
- The set of final states is irrelevant to the error behavior—the automaton is not being used to match strings—so set F is unnecessary.

Only the transitions and propagations have not yet been described. After a discussion of the use of environments as the input alphabet, this section describes the semantics of transitions, describes the semantics of propagations, and, finally, gives a full description of the behavior automaton for a specific component.

9.1.1 Environments as Input Symbols

This section clarifies the use of the set of environments $\Gamma_\square$ as the input alphabet. The challenge here is that an environment is a tuple (manipulated as a record); see Section 5 for more information. How can a record be used as an input character? This section shows that it is possible to construct a set of symbols $\Sigma_\square$ with a one-to-one correspondence to set $\Gamma_\square$. Then the set $\Sigma_\square$ is used as the input alphabet. But because every symbol in $\Sigma_\square$ corresponds to exactly one environment tuple in $\Gamma_\square$, the symbol can easily be interpreted as that environment when necessary and vice versa. Thus, there is no practical reason to distinguish between the members of $\Sigma_\square$ and $\Gamma_\square$.

For a given semantic component $\square = [\![k]\!]_{\mathsf{K}}$, consider the following:

1. Because the EMV2 model must be of finite length (as measured in characters), the following sets must also be finite:
   (a) the set of declared error events $\mathbf{D}^k_{\mathsf{Event}}$ and thus also the set $\mathcal{V}_\square$
   (b) the set of declared error types $\mathbf{D}_{\mathsf{Type}}$ and thus $\mathcal{E}$
   (c) the sets of in and out propagation points $\mathbf{D}^k_{\mathsf{In}}$ and $\mathbf{D}^k_{\mathsf{Out}}$
   (d) the set of declared subcomponents $\mathbf{D}^k_{\mathsf{Sub}}$
   (e) the set of propagation point references $\mathbf{R}^k_{\mathsf{Out}}$ (from 1d and 1c)
2. Based on 1b and the fact that $\mathsf{product}(p)$ must be true for all products $p \in \mathcal{P}$ that appear in the EMV2 model, a product cannot have more elements than the number of unique roots, $n_r \leq |\mathcal{E}|$, in $\mathcal{E}$. Thus, in practice, the only products that can appear in the EMV2 model for component instance $\square$ are those in $\mathcal{P}_\square = \bigcup_{i=2}^{n_r} \mathcal{P}_i$, a set with large but finite cardinality.
3. A type set is thus limited to drawing members from the set $\mathcal{T}_\square = \mathcal{E} \cup \mathcal{P}_\square$, which by 1b and 2 must also be a finite set.
4. By 3, the maximum cardinality of a type set s referenced in component instance $\square$ is thus $n_T = |\mathcal{T}_\square|$, and the cardinality of the set of expressible type sets $\mathcal{S}_\square = \mathcal{S} \cap 2^{\mathcal{T}_\square}$ must also be finite, although extremely large! (The set of sets of elements of $\mathcal{T}_\square$ is intersected with $\mathcal{S}$ to enforce the requirements of $\mathsf{typeset}$.)
5. By 4, for all $f \in \mathbf{D}^k_{\mathsf{In}}$ the set $[\![\mathbf{D}^k_{\triangleright\mathsf{PP}}(f)]\!]_{\mathsf{Set}}$ is finite. Similarly, for all $f \in \mathbf{D}^k_{\mathsf{Out}}$ the set $[\![\mathbf{D}^k_{\mathsf{PP}\triangleright}(f)]\!]_{\mathsf{Set}}$ is finite.
6. By 1a, 1c, 1e, and 5 there are a finite number of error events, in propagation points, and outgoing subcomponent propagation point references, and the cardinality of all the declared propagations is finite. Therefore, the cardinality of $\Gamma_\square$ must be finite.

Now, let $\Sigma_\square \subseteq \mathcal{A}$ be a set of symbols such that

- $|\Sigma_\square| = |\Gamma_\square|$

[DISTRIBUTION STATEMENT A] Approved for public release and unlimited distribution.

- Function $\theta : \Gamma_\square \to \Sigma_\square$ uniquely defines $\Sigma_\square$.

Clearly $\theta^{-1} : \Sigma_\square \to \Gamma_\square$ uniquely defines $\Gamma_\square$. The set of symbols $\Sigma_\square$ can now be said to be the input alphabet to the automaton $\mathcal{B}_\square$. The corresponding environment for an input symbol $a \in \Sigma_\square$ is $\gamma = \theta(a)$; the input symbol for an environment $\gamma \in \Gamma_\square$ is thus $a = \theta^{-1}(\gamma)$. There is little to be gained besides confusion by continuing to make this correspondence explicit, so hereon, as presented above, the set of input symbols to automaton $\mathcal{B}_\square$ is $\Gamma_\square$.

9.2 Transitions and Propagations—Prelude

The standard describes transitions [27, E.8.2] and output error propagation conditions [27, E.10.1]. The EMV2 model attached to a component specifies transitions and outgoing propagation declarations separately. Within the *declarative AADL model*,

- transitions are declared in
  - the `transitions` section of an `error behavior` declaration within the error model library
  - the `transitions` section of a component-specific `component error behavior` declaration
- propagations are declared in the `propagations` section of a component-specific `component error behavior` declaration

The instance model—and thus the syntactic component model used herein—keeps the transitions and propagations separate, although it does collect together all the transitions declared in the component's error behavior, including those from the named `error behavior` and from the component's `component error behavior`.[2] Similarly, it collects all the propagations from the component's `component error behavior`.

The point of giving this unusual level of detail on the specification of transitions and outgoing propagations for a component is to emphasize that the specification of transitions and the specification of outgoing propagations are completely *separate*. Therefore, it is necessary to describe the relationship between transitions and outgoing propagations in the behavior automaton. Generally,

- a transition declaration specifies a source state, a condition, and a target (destination) state
- an outgoing propagation declaration specifies a state, a condition, and a propagated error type

The issue at hand is the meaning of the states in the declarations. For transitions there are no surprises: "An error behavior transition specifies a transition from a source state to a target state if a transition condition is satisfied" [27, E.8.2.(4)]. More interesting is the interpretation of the state in an outgoing propagation, which is mentioned in three places:

1. An outgoing propagation declaration specifies the "conditions under which outgoing error propagations occur in terms of incoming error propagations and the *target error behavior state*" [27, E.10.(2)] [emphasis added].
2. "The error behavior state referenced in an outgoing propagation declaration is *the new (target) state of a transition*, if a transition occurs; otherwise it is the current state" [27, E.10.1.(3)] [emphasis added].

[2] Classifiers may extend the component error behavior of their ancestor, so the full behavior of an instantiated component may come from many places in the declarative model.

[DISTRIBUTION STATEMENT A] Approved for public release and unlimited distribution.

3. "An outgoing error propagation declaration can specify that an outgoing error propagation is triggered by an error behavior event. This can be the error behavior event independent of an error behavior state, or when the system is in a particular error behavior state. In the latter case, *the new (target) state of a transition*, if the event also affects a transition. [sic]" [27, E.10.1.(9)] [emphasis added]

It is thus clear that *transitions must be considered first*, and then outgoing propagations based on the results of any transitions that were actually performed. This is revisited in detail in Section 9.6.2. The following sections describe the semantics of transitions, the output alphabet, the semantics of outgoing propagations, and the exact relationship between transitions and propagations and, finally, summarize the behavior automata.

9.3 Transitions

As mentioned in the introduction, *transitions are deterministic*, in that only one transition may apply to any given combination of source state and input symbol/environment:

> The set of outgoing error behavior transitions from the same source error behavior state to different target states must be unambiguous for a given component, i.e., they must uniquely identify the target state for a given state, error behavior events, and incoming error propagations. [27, E.8.2.(12)]

This section describes how to detect the existence of ambiguity in transitions when conditions are considered.

The target state is *probabilistic*. That is, the unambiguous applicable transition may specify multiple potential target states, together with the probability that each is the actual target:

> A transition can be a branching transition with multiple target states. Once the transition is taken, one of the specified target states is selected according to a specified probability with fixed distribution. The probabilities of all branches must add up to one. One of the branches may specify `others`—taking on a probability value that is the difference between the probability value sum of the other branches and the value one. [27, E.8.2.(5)]

It is possible—as would be expected—to declare multiple transitions between the same source and target state with different conditions.[3] These are referred to as *alternate transition conditions*:

> The transition condition expression of an error behavior transition declaration can specify one or more alternative conditions, one of which must be satisfied in order for the transition to be triggered. Multiple error behavior transition declarations may name the same source and target state. In this case the transition condition expression of each transition declaration is considered to be an alternative transition condition. [27, E.8.2.(7)]

The syntactic domain $\mathbf{D}^{k}_{\mathsf{Trans}} \subseteq \mathbf{O}$ is the set of transition objects declared in component instance k.[4] Figure 9.1 shows the syntactic domains and abstract production rules for the source and target states of a transition. Here the syntactic domain $\mathbf{Real} \subset \mathbf{O}$ contains objects representing decimal numbers.[5] The semantic function $[\![\cdot]\!]_{\mathsf{R}} : \mathbf{Real} \to \mathbb{R}$ maps the object representation into a real number. Regarding branched transitions, it is assumed that

- any reference to `others` has been resolved into the correct decimal probability

[3] Indeed, this is similar to defining a DFA in which inputs a or b may cause a transition from state s to t. It is shown herein, however, that for a behavior automaton the single condition `a or b` is not the same as two separate alternative conditions `a` and `b`.

[4] Cf. `TransitionInstance` in the EMV2 instance meta model.

[5] Cf. the Java class `BigDecimal`.

[DISTRIBUTION STATEMENT A] Approved for public release and unlimited distribution.

$$\begin{array}{c}
\mathbf{Src}_k : \mathbf{Source_State}_k \\
\mathbf{Tgt}_k : \mathbf{Target}_k \\
\mathbf{Tgt}^k_{\mathsf{State}} : \mathbf{Target_State}_k \\
\mathbf{Tgt}^k_{\mathsf{Branch}} : \mathbf{Target_Branch}_k \\
\mathbf{Tgt}^k_{\mathsf{Prob}} : \mathbf{Target_Prob}_k \\
\mathbf{Real} : \textit{See Main Text}
\end{array}$$

$$\begin{array}{rcl}
\mathbf{Source_State}_k & ::= & \texttt{all} \mid \mathbf{D}^k_{\mathsf{State}} \\
\mathbf{Target}_k & ::= & \mathbf{Target_State}_k \\
 & \mid & \mathbf{Target_Branch}_k \\
\mathbf{Target_State}_k & ::= & \texttt{same state} \mid \mathbf{D}^k_{\mathsf{State}} \\
\mathbf{Target_Branch}_k & ::= & \mathbf{Target_Prob}_k \texttt{ , } \mathbf{Target_Branch}_k \\
 & \mid & \mathbf{Target_Prob}_k \\
\mathbf{Target_Prob}_k & ::= & \mathbf{Target_State}_k \texttt{ with } \mathbf{Real}
\end{array}$$

Figure 9.1: Syntactic Domains and Astract Production Rules for Transition Sources and Targets

- the probabilities of the branches of each transition have been verified to sum to 1

Naturally, there are three syntactic functions to get information from a transition:

- $\mathbf{Tr}^k_{\mathsf{Cond}} : \mathbf{D}^k_{\mathsf{Trans}} \to \hat{\mathbf{C}}_k$ is the condition of the transition.[6]
- $\mathbf{Tr}^k_{\mathsf{Src}} : \mathbf{D}^k_{\mathsf{Trans}} \to \mathbf{Src}_k$ is the source state of the transition.
- $\mathbf{Tr}^k_{\mathsf{Tgt}} : \mathbf{D}^k_{\mathsf{Trans}} \to \mathbf{Tgt}_k$ is the target state or branch of the transition.

The semantics of these three parts of a transition are described below, followed by the full meaning of a transition. As usual, it is assumed that k is the current syntactic component and that $\square = [\![k]\!]_{\mathsf{K}}$.

9.3.1 Conditions

The semantics of EMV2 conditions is described in Section 7. Here it is repeated that SAE AS5506/5 states,

> An alternative transition condition specifies all the error behavior events and error propagations that must be present in order for the condition to hold. Any error propagation points not specified must not have an error propagation present. [27, E.8.2.(8)]

The semantic function $\mathsf{satisfiedBy}_\square$ is thus used to evaluate the transition's condition to a set of environments that satisfy the condition. In this way, the *symbolic* nature of the automaton is eliminated [9, 33]. *Any* input symbol/environment that is a member of the set enables the transition; this is formalized below in Section 9.3.5.

Note that alternative transition conditions are *not* equivalent to a disjunction of the individual conditions. This is due to the silencing semantics of $\mathsf{satisfiedBy}_\square$. For example, consider the two alternative transition conditions `in in1{T1}` and `in in2{T2}`:

- The meaning of the first condition is $c_1 = \mathsf{satisfiedBy}_\square(\texttt{in in1\{T1\}})$, which will require that propagation point `in2` is silenced.
- Similarly, the meaning of the second condition is $c_2 = \mathsf{satisfiedBy}_\square(\texttt{in in2\{T2\}})$, which will require that propagation point `in1` is silenced.
- The meaning of the disjunction is $c_3 = \mathsf{satisfiedBy}_\square(\texttt{in in1\{T1\} or in in2\{T2\}})$, which silences neither `in1` nor `in2`.
- Thus, for environment $\gamma = \langle \texttt{in1} \mapsto \texttt{T1}, \texttt{in2} \mapsto \texttt{T2} \rangle$, it is the case that $\gamma \notin c_1$, $\gamma \notin c_2$, but that $\gamma \in c_3$.

[6] Tr for **Tr**ansition. This is to avoid confusion with the T sets used for triggers (e.g., $\mathbf{T}^k_{\mathsf{Event}}$).

9.3.2 Source State

The meaning of the source of a transition is straightforward. It is the set of states of which the current state of the behavior automaton must be a member. There are two options:

- A single specific state is identified, in which case the semantic function $[\![\cdot]\!]^{\square}_{\mathsf{State}}$ is used to translate the state to the semantic domain.
- The source state is `all`, which is defined in EMV2 [27, E.8.2.(4)]: "The keyword `all` may be used to express that the transition applies to all source states." So `all` must mean the set of all states in the component: $Q_{\square}$.

The semantic function $[\![\cdot]\!]^{\square}_{\mathsf{Src}} : \mathbf{Source\_State}_k \to 2^{Q_{\square}}$ is therefore

$$\begin{aligned} [\![q]\!]^{\square}_{\mathsf{Src}} &= \{[\![q]\!]^{\square}_{\mathsf{State}}\} \\ [\![\texttt{all}]\!]^{\square}_{\mathsf{Src}} &= Q_{\square} \end{aligned}$$

9.3.3 Transition Target

Branching transitions complicate the interpretation of transition targets because they require representing not only multiple destination states but also their probabilities. Traditionally this is accomplished by representing the target as a probability distribution $\mu : X \to [0, 1]$ where X is the set of states in the automaton [32]. The set of all probability distributions over X is $\mathsf{Distr}(X)$. The notation $\{x_1 \mapsto p_1, \ldots\}$, where elements with 0 probability are left out, is used to succinctly write μ. Finally, $x \mapsto p \in \mu \Leftrightarrow \mu(x) = p$.

Thus the semantic function ought to map a transition target to a distribution function over $Q_{\square}$. However, the interpretation of `same state` requires knowing *what the current state is*, so the meaning of a transition target must actually be a function parameterized by the current state: $[\![\cdot]\!]^{\square}_{\mathsf{Tgt}} : \mathbf{Tgt}_k \to Q_{\square} \to \mathsf{Distr}(Q_{\square})$. The function $[\![\cdot]\!]^{\square}_{\mathsf{Tgt}}$ simply delegates to the subrules:

$$\begin{aligned} [\![ts]\!]^{\square}_{\mathsf{Tgt}} &= [\![ts]\!]^{\square}_{\mathsf{Tgt\_S}} \\ [\![branch]\!]^{\square}_{\mathsf{Tgt}} &= [\![branch]\!]^{\square}_{\mathsf{Tgt\_B}} \end{aligned}$$

The semantic function $[\![\cdot]\!]^{\square}_{\mathsf{Tgt\_S}} : \mathbf{Tgt}^k_{\mathsf{State}} \to Q_{\square} \to \mathsf{Distr}(Q_{\square})$ maps target states into a function from a state to a singleton probability distribution using the helper $\mathsf{state}_{\square} : \mathbf{Tgt}^k_{\mathsf{State}} \times Q_{\square} \to Q_{\square}$:

$$\begin{aligned} [\![ts]\!]^{\square}_{\mathsf{Tgt\_S}} &= \lambda q_{\text{start}}.\, \{\mathsf{state}_{\square}(ts, q_{\text{start}}) \mapsto 1\} \\ \mathsf{state}_{\square}(\texttt{same state},\ q') &= q' \\ \mathsf{state}_{\square}(q,\ q') &= [\![q]\!]^{\square}_{\mathsf{State}} \end{aligned}$$

The semantic function $[\![\cdot]\!]^{\square}_{\mathsf{Tgt\_B}} : \mathbf{Tgt}^k_{\mathsf{Branch}} \to Q_{\square} \to \mathsf{Distr}(Q_{\square})$ maps branching targets into a function from a state to a probability distribution constructed from the union of its members:

$$\begin{aligned} [\![tp]\!]^{\square}_{\mathsf{Tgt\_B}} &= [\![tp]\!]^{\square}_{\mathsf{Tgt\_P}} \\ [\![tp\ ,\ tb]\!]^{\square}_{\mathsf{Tgt\_B}} &= \lambda q_{\text{start}}.\, ([\![tp]\!]^{\square}_{\mathsf{Tgt\_P}}(q_{\text{start}}) \cup [\![tb]\!]^{\square}_{\mathsf{Tgt\_B}}(q_{\text{start}})) \end{aligned}$$

Finally, the semantic function $[\![\cdot]\!]^{\square}_{\mathsf{Tgt\_P}} : \mathbf{Tgt}^k_{\mathsf{Prob}} \to Q_{\square} \to \mathsf{Distr}(Q_{\square})$ maps a probabilistic target into a function from a state to a singleton probability distribution:

$$[\![ts\ \texttt{with}\ p]\!]^{\square}_{\mathsf{Tgt\_P}} = \lambda q_{\text{start}}.\, \{\mathsf{state}_{\square}(ts, q_{\text{start}}) \mapsto [\![p]\!]_{\mathsf{R}}\}$$

[DISTRIBUTION STATEMENT A] Approved for public release and unlimited distribution.

9.3.4 Determinism

As stated previously, transitions are required to be deterministic: "For the current error behavior state there must be one unique outgoing transition" [27, E.10.(C10)].

When transitions are defined in terms of condition expressions, this is a difficult requirement to reason about. Reducing conditions to sets of environments, however, enables this restriction to be reasoned about in terms of these sets. In particular, the restriction is that if the set of source states of two transitions has a non-empty intersection, then the intersection of the satisfying environments for their conditions must be empty:

$$\forall \tau_1, \tau_2 \in \mathbf{D}^k_{\mathsf{Trans}}.\ s_1 \cap s_2 \neq \emptyset \Rightarrow \Gamma_1 \cap \Gamma_2 = \emptyset$$

where

$$\begin{aligned} s_i &= [\![\mathbf{Tr}^k_{\mathsf{Src}}(\tau_i)]\!]^{\square}_{\mathsf{Src}} \\ \Gamma_i &= \mathsf{satisfiedBy}_{\square}(\mathbf{Tr}^k_{\mathsf{Cond}}(\tau_i)) \end{aligned}$$

9.3.5 Transition Semantics

As mentioned above, the transition function for a DFA with states Q and input alphabet Σ is $\delta : Q \times \Sigma \to Q$. The behavior automata is no longer a DFA but a probabilistic automata whose target states are described by probability distributions. It is now the case that the transition function for a behavior automaton $\mathcal{B}_{\square}$ with states $\mathcal{Q}_{\square}$ and input alphabet $\Gamma_{\square}$ is $\delta_{\square} : \mathcal{Q}_{\square} \times \Gamma_{\square} \to \mathsf{Distr}(\mathcal{Q}_{\square})$. The semantics of a single transition $\tau \in \mathbf{D}^k_{\mathsf{Trans}}$ is *one or more transitions* in $\delta_{\square}$. The condition of τ evaluates to a set of environments; thus, τ defines a set of transitions, all of which start at same start state, and end at the same probability distribution, but which have different input environments.

The semantic function $[\![\cdot]\!]^{\square}_{\mathsf{Trans}} : \mathbf{D}^k_{\mathsf{Trans}} \to (\mathcal{Q}_{\square} \times \Gamma_{\square} \to \mathsf{Distr}(\mathcal{Q}_{\square}))$ yields a transition function by taking the cross product of the transition's start states, satisfying environments, and target probability distribution. Note that the semantic representation of the start state is passed to the *function* that is the semantic meaning of the transition target.

$$[\![\tau]\!]^{\square}_{\mathsf{Trans}} = \bigcup_{s \in srcStates} (\{s\} \times \Gamma \times \{tgt(s)\})$$

where

$$\begin{aligned} srcStates &= [\![\mathbf{Tr}^k_{\mathsf{Src}}(\tau)]\!]^{\square}_{\mathsf{Src}} \\ \Gamma &= \mathsf{satisfiedBy}_{\square}(\mathbf{Tr}^k_{\mathsf{Cond}}(\tau)) \\ tgt &= [\![\mathbf{Tr}^k_{\mathsf{Tgt}}(\tau)]\!]^{\square}_{\mathsf{Tgt}} \end{aligned}$$

Now the full transition function $\delta_{\square}$ is simply the union of all the transition functions produced by each transition:

$$\delta_{\square} = \bigcup_{\tau \in \mathbf{D}^k_{\mathsf{Trans}}} [\![\tau]\!]^{\square}_{\mathsf{Trans}}$$

9.3.5.1 Most Specific Transition

Finally, consider EMV2, which explains the reasoning behind the "silencing" semantics described in Section 7.1:

> Note: we chose to interpret listing a single error propagation point as all others being error free, because modelers often assume that they are dealing with one incoming error propagation at a time. [27, E.8.2.(9)]

That is, silencing is a design decision made to accommodate the way system designers think about errors. In particular, it ensures that the *most specific* transition is taken. This point is revisited in Section 9.5.1.

9.4 The Output Alphabet

A separate alphabet is used to describe the output of the behavior automata. For Mealy machines, this alphabet is traditionally named Δ [12], but it is named Π herein to more clearly associate it with propagations. For the behavior automata, this alphabet must be able to describe all the outgoing propagations of the component. Unlike the environments that serve as the input alphabet, it does not need to express an event instance or propagations related to subcomponents. The output alphabet is defined as a set of records in manner similar to the input alphabet; it can also be put into a one-to-one correspondence with a set of symbols in the same way as the input alphabet.

Set $\Pi_\square$ is the set of *propagation records* for component $\square = [\![k]\!]_{\mathsf{K}}$. Specific outputs are associated with the symbol π. The elements of set $\Pi_\square$ are structured as records with the index set $\mathcal{I}^{\triangleright}_\square$, so we have

$$(\Gamma_\square, \mathcal{I}^{\triangleright}_\square, E'_\square, \epsilon'_\square)$$

The propagation record contains one field for each out propagation point of the component, each indicating that point's currently propagated type. The *semantic function* $[\![\cdot]\!]^{\square}_{\mathsf{Field}\triangleright} : \mathbf{D}^k_{\mathsf{Out}} \to \mathcal{I}^{\triangleright}_\square$ maps references to propagation points to fields of the record. The index set $\mathcal{I}^{\triangleright}_\square$ and semantic function $[\![\cdot]\!]^{\square}_{\mathsf{Field}\triangleright}$ are concurrently defined:

- For each $f \in \mathbf{D}^k_{\mathsf{Out}}$ there is a unique symbol $i \in \mathcal{I}^{\triangleright}_\square$ such that $[\![f]\!]^{\square}_{\mathsf{Field}\triangleright} = i$. The value of this field is an error type, specifically a member of $[\![\mathbf{D}_{\mathsf{PP}\triangleright}(f)]\!]_{\mathsf{Set}}$.
- The function $[\![\cdot]\!]^{\square}_{\mathsf{Field}\triangleright}$ uniquely defines the set $\mathcal{I}^{\triangleright}_\square$.

The set of empty values is $E'_\square = \{\epsilon_{\mathsf{Type}}\}$. The empty value for each field is ϵ_{Type}: $\forall f \in \mathbf{D}^k_{\mathsf{Out}} . \epsilon'_\square([\![f]\!]^{\square}_{\mathsf{Field}\triangleright}) = \epsilon_{\mathsf{Type}}$.

Finally, the set of records is

$$\Pi_\square = \prod_{f \in \mathbf{D}^k_{\mathsf{Out}}} [\![f]\!]^{\square}_{\mathsf{Field}\triangleright} : ([\![\mathbf{D}_{\mathsf{PP}\triangleright}(f)]\!]_{\mathsf{Set}} \cup \{\epsilon_{\mathsf{Type}}\})$$

9.5 Outgoing Propagations

As mentioned above, outgoing propagations associated with state transitions make the behavior automata similar to a Mealy machine [18]. Traditionally, automata output is governed by an output function $\lambda : Q \times \Sigma \to \Delta$ that maps the current state and input symbol to an output symbol [12]. For a behavior automaton $\mathcal{B}_\square$, the output function is $\lambda_\square : \mathcal{Q}_\square \times \Gamma_\square \to \Pi_\square$. A novel aspect of the behavior automata is that multiple output propagations on distinct propagation points may be possible for a given state–environment pair. Abstractly, this reduces to a single output token because a single propagation record can describe this situation. But because the propagations are described by distinct output propagation declarations in the EMV2 model, the propagations are represented by multiple outputs in the output function $\lambda_\square$. *The behavior automata must, therefore, combine the multiple outputs into a single propagation record.*

The syntactic domain $\mathbf{D}^k_{\mathsf{OPC}} \subseteq \mathbf{O}$ is the set of output propagation objects declared in component instance k.[7] Figure 9.2 shows the syntactic domains and abstract production rules for the features of an output propagation conditions. There are three syntactic functions to get information from an output propagation condition:

- $\mathbf{OPC}^k_{\mathsf{Cond}} : \mathbf{D}^k_{\mathsf{OPC}} \to \mathbf{C}^k_{\mathsf{Opt}}$ is the *optional* condition of the outgoing propagation.

[7] Cf. `OutgoingPropagationConditionInstance` in the EMV2 instance meta model.

[DISTRIBUTION STATEMENT A] Approved for public release and unlimited distribution.

$$
\begin{array}{rl}
\mathbf{C}^k_{\mathsf{Opt}} : \mathbf{OptCondition}_k & \mathbf{OptCondition}_k ::= \diamond\mathbf{Condition}_k \mid \epsilon \\
\mathbf{Tgt}^k_{\mathsf{Prop}} : \mathbf{Target\_Prop}_K & \mathbf{Target\_Prop}_k ::= \mathbf{Target\_Point}_K\ \mathbf{Target\_Type}_K \\
\mathbf{Tgt}^k_{\mathsf{Point}} : \mathbf{Target\_Point}_K & \mathbf{Target\_Point}_k ::= \mathbf{D}^k_{\mathsf{Out}} \mid \texttt{all} \\
\mathbf{Tgt}^k_{\mathsf{Err}} : \mathbf{Target\_Type}_K & \mathbf{Target\_Type}_k ::= \mathbf{R}_{\mathsf{Type}} \mid \mathbf{TypeProduct} \mid \texttt{noerror}
\end{array}
$$

Figure 9.2: Syntactic Domains and Abstract Production Rules for Output Propagation Conditions

- $\mathbf{OPC}^k_{\mathsf{Src}} : \mathbf{D}^k_{\mathsf{OPC}} \to \mathbf{Src}_k$ is the source state of the outgoing propagation. These are the same as for transitions.
- $\mathbf{OPC}^k_{\mathsf{Tgt}} : \mathbf{D}^k_{\mathsf{OPC}} \to \mathbf{Tgt}^k_{\mathsf{Prop}}$ is the output propagation target, that is, the specification of propagation point and error type.

The semantics of these three parts of an outgoing propagation condition are described below, and then the full meaning of an outgoing propagation condition is described. As usual, it is assumed that k is the current syntactic component and that $\square = [\![k]\!]_{\mathsf{K}}$.

9.5.1 Conditions

Conditions in propagations differ from those in transitions in two ways: (1) they are optional, and (2) they are evaluated *without* silencing. When the condition is empty, ϵ in the abstract production rule, the condition is always satisfied: "An empty condition expression indicates that the outgoing error propagation occurs whenever the component is in the specified error behavior state" [27, E.10.1.(4)]. This means the empty condition evaluates to $\Gamma_\square$; it is satisfied by *any* environment.

Silencing as described in SAE AS5506/5 [27, E.8.2.(8)] applies to *alternate transition conditions*. The EMV2 specification does not discuss conditions with respect to outgoing propagation conditions per se. Section E.8.2 is, however, titled "Error Behavior States and Transitions," suggesting that generally it should not apply to outgoing propagation conditions, which are described in Section E.10.1. The decision is thus made herein to evaluate the conditions of output propagation conditions without silencing. In this way, by being agnostic about the value of unspecified propagation points, the condition is evaluated in *the most general* context possible. This serves the design principle that *as many errors should be reported as possible*.

The semantic function $[\![\cdot]\!]^\square_{\mathsf{Opt}} : \mathbf{C}^k_{\mathsf{Opt}} \to 2^{\Gamma_\square}$ is thus

$$
\begin{array}{rcl}
[\![\epsilon]\!]^\square_{\mathsf{Opt}} & = & \Gamma_\square \\
[\![c]\!]^\square_{\mathsf{Opt}} & = & [\![\lfloor c \rfloor^k_{\mathsf{Cond}}]\!]^\square_{\mathsf{Cond}}
\end{array}
$$

9.5.2 Source State

The abstract production rules for source states is the same as used for transitions. The EMV2 specification states: "The keyword `all` instead of the error behavior state indicates that it applies to all states, i.e., the outgoing propagation is solely determined by incoming propagations" [27, E.10.1.(4)]. The meaning of source states is thus also the same as for transitions and the same semantic function is used: $[\![\cdot]\!]^\square_{\mathsf{Src}}$.

9.5.3 Propagation Target

The propagation target specifies both the outgoing propagation point and the error type to be propagated. The propagation point must be an outgoing propagation point of the component, a member of $\mathbf{D}^k_{\mathsf{Out}}$, or the specification `all`. The EMV2 specification actually fails to

[DISTRIBUTION STATEMENT A] Approved for public release and unlimited distribution.

specify the meaning of `all` in this context, although it seems reasonable that it should mean *all members of* $\mathbf{D}^k_{\mathsf{Out}}$. Here it is especially important to consider the following: "The optional `target_error_type_instance` of the propagation target in an outgoing propagation declaration must be contained in the error type set specified with the defining error propagation point declaration and must not be contained in an outgoing error containment declaration" [27, E.10.(L36)]. In other words, the propagated type for propagation point f must be contained in $[\![\mathbf{D}_{\mathsf{PP}_\triangleright}(f)]\!]_{\mathsf{Set}}$. This is specified in more detail below.

The semantic function $[\![\cdot]\!]^{\square}_{\mathsf{Point}} : \mathbf{Tgt}^k_{\mathsf{Point}} \to 2^{\mathcal{I}^{\triangleright}_{\square}}$ is

$$\begin{aligned} [\![f]\!]^{\square}_{\mathsf{Point}} &= \{[\![f]\!]^{\square}_{\mathsf{Field}_\triangleright}\} \\ [\![\texttt{all}]\!]^{\square}_{\mathsf{Point}} &= \mathcal{I}^{\triangleright}_{\square} \end{aligned}$$

The error type must be a declared error type or type alias, or a product type. It cannot be a type set: the intent is that a specific error is being indicated. Alternatively, the target can specify that `NoError` is propagated. The EMV2 specification allows the propagated error type to be unspecified: "An outgoing propagation declaration can also explicitly specify the error type to be propagated. If this propagated error type is not specified, type transformation rules or default rules are used to determine the propagated error type" [27, E.10.1.(6)]. Herein, however, type transformation rules are not considered, so instead the error type is made mandatory.

The semantic function $[\![\cdot]\!]^{\square}_{\mathsf{Err}} : \mathbf{Tgt}^k_{\mathsf{Err}} \to \mathcal{T} \cup \{\epsilon_{\mathsf{Type}}\}$ is

$$\begin{aligned} [\![t \in \mathbf{R}_{\mathsf{Type}}]\!]^{\square}_{\mathsf{Err}} &= [\![t]\!]_{\mathsf{Ref}} \\ [\![p \in \mathbf{P}]\!]^{\square}_{\mathsf{Err}} &= [\![p]\!]_{\mathsf{Product}} \\ [\![\texttt{noerror}]\!]^{\square}_{\mathsf{Err}} &= \epsilon_{\mathsf{Type}} \end{aligned}$$

Now that the meaning of the propagated error type is defined, the constraint [27, E.10.(L36)] can be specified for *point err* $\in \mathbf{Tgt}^k_{\mathsf{Prop}}$. Let $\theta(\cdot)$ be the inverse of the function $[\![\cdot]\!]^{\square}_{\mathsf{Field}_\triangleright}$. The constraint is thus

$$\forall f \in [\![point]\!]^{\square}_{\mathsf{Point}}.\, [\![err]\!]^{\square}_{\mathsf{Err}} \in ([\![\mathbf{D}_{\mathsf{PP}_\triangleright}(\theta(f))]\!]_{\mathsf{Set}} \cup \epsilon_{\mathsf{Type}})$$

The overall meaning of a propagation target must be propagation points paired with error types. A propagation record is not yet introduced here—a function from fields to propagated types is used instead because it makes checking for nondeterminism easier (see below). The semantic function $[\![\cdot]\!]^{\square}_{\mathsf{Prop}} : \mathbf{Tgt}^k_{\mathsf{Prop}} \to (\mathcal{I}^{\triangleright}_{\square} \to (\mathcal{T} \cup \epsilon_{\mathsf{Type}}))$ generates a propagation mapping from a propagation target:

$$[\![point\ err]\!]^{\square}_{\mathsf{Prop}} = [\![point]\!]^{\square}_{\mathsf{Point}} \times [\![err]\!]^{\square}_{\mathsf{Err}}$$

Note that the propagation mapping either has a single element or $|\mathcal{I}^{\triangleright}_{\square}|$ elements, and the range of the mapping is always the singleton set $\{[\![err]\!]^{\square}_{\mathsf{Err}}\}$.

9.5.4 Determinism

The outgoing propagations of the behavior automaton must be deterministic: "The result of evaluating the outgoing propagation declarations for an outgoing error propagation must result in at most one propagated error type" [27, E10.(C14)].

In this context and in the language used in the EMV2 specification, "outgoing error propagation" means a single outgoing propagation point. So for every state and environment, the out propagation, if any, at each out propagation point must be unambiguous. Ambiguity would

arise if a pair of propagation points had the same start state, conditions that were both satisfied by the same environment, and propagation mappings that mapped the same propagation point to *different* values. This is the condition that *must not occur*. The opposite condition that *must always be* is

$$\forall o_1, o_2 \in \mathbf{D}^k_{\mathsf{OPC}}. \left(\begin{array}{l} (s_1 \cap s_2 \neq \emptyset) \ \wedge \ (\Gamma_1 \cap \Gamma_2 \neq \emptyset) \Rightarrow \\ \forall f \in \mathcal{I}^{\triangleright}_{\square}.(z_1(f) = z_2(f) \ \vee \ (\mathsf{undefined}\ z_1(f) \ \wedge \ \mathsf{undefined}\ z_2(f))) \end{array} \right)$$

where

$$\begin{array}{rcl} s_i & = & [\![\mathbf{OPC}^k_{\mathsf{Src}}(o_i)]\!]^{\square}_{\mathsf{Src}} \\ \Gamma_i & = & [\![\mathbf{OPC}^k_{\mathsf{Cond}}(o_i)]\!]^{\square}_{\mathsf{Opt}} \\ z_i & = & [\![\mathbf{OPC}^k_{\mathsf{Tgt}}(o_i)]\!]^{\square}_{\mathsf{Prop}} \end{array}$$

9.5.5 Propagation Semantics

Having established that the set of outgoing propagation conditions are deterministic, the output function $\lambda_{\square}$ is constructed from the individual outgoing propagation conditions. The meaning of each outgoing propagation condition is a function that maps a state and an environment to a propagation mapping: $[\![\cdot]\!]^{\square}_{\mathsf{OPC}} : \mathbf{D}^k_{\mathsf{OPC}} \rightarrow ((\mathcal{Q}_{\square} \times \Gamma_{\square}) \rightarrow (\mathcal{I}^{\triangleright}_{\square} \rightarrow (\mathcal{T} \cup \epsilon_{\mathsf{Type}})))$. A simple cross product is sufficient:

$$[\![o]\!]^{\square}_{\mathsf{OPC}} = [\![\mathbf{OPC}^k_{\mathsf{Src}}(o)]\!]^{\square}_{\mathsf{Src}} \times [\![\mathbf{OPC}^k_{\mathsf{Cond}}(o)]\!]^{\square}_{\mathsf{Opt}} \times \{[\![\mathbf{OPC}^k_{\mathsf{Tgt}}(o)]\!]^{\square}_{\mathsf{Prop}}\}$$

The full output function $\lambda_{\square}$ is constructed by merging together, via set union, functions generated by all the outgoing propagation conditions in the component. The result remains a function: every $\lambda_{\square}(q, \gamma)(f)$ has a single value because the outgoing propagations have already been constrained to be deterministic; see above. The propagation function generated for each $\lambda_{\square}(q, \gamma)$ is finally converted to a propagation record in $\Pi_{\square}$ using the record operator (see Section 2.4).

$$\lambda_{\square}(q, \gamma) = \mathsf{record} \left(\bigcup_{o \in \mathbf{D}^k_{\mathsf{OPC}}} [\![o]\!]^{\square}_{\mathsf{OPC}}(q, \gamma) \right)$$

9.6 Behavior Automata—Conclusion

This section provides a complete definition of a behavior automaton and its implementation for a specific component. A behavior automaton is defined by the structure $B = (Q, \Gamma, \Pi, \delta, \lambda, q_0)$:

- $Q \subseteq \mathcal{A}$ is the set of behavior states; see Section 4.2.2.
- Γ is the set environments/input symbols; see Section 5.3.
- Π is the set of propagation records/output symbols; see Section 9.4.
- $\delta : Q \times \Gamma \rightarrow \mathsf{Distr}(Q)$ is the *probabilistic* transition function; see Section 9.3.5. $\delta(q, \gamma) = \mu$, where $q' \mapsto p \in \mu$ indicates a transition to state q' with a probability of p. The transition function may be *partial*.
- $\lambda : Q \times \Gamma \rightarrow \Pi$ is the output function; see Section 9.5.5. The output function may be *partial*.
- $q_0 \in Q$ is the initial state; see Section 4.2.2.

A behavior automaton has the same structure as a Mealy machine, but the operation of the behavior automata is slightly different: as discussed in Section 9.2, propagations are considered *after* transitions, based on the *target state* of the transition.

[DISTRIBUTION STATEMENT A] Approved for public release and unlimited distribution.

9.6.1 For Component k

For a specific component $\square = [\![k]\!]_{\mathsf{K}}$, the behavior automaton is

$$\mathcal{B}_{\square} = (\mathcal{Q}_{\square}, \Gamma_{\square}, \Pi_{\square}, \delta_{\square}, \lambda_{\square}, [\![q_0^k]\!]^{\square}_{\mathsf{State}})$$

9.6.2 Operation

The operation of a behavior automata B is as follows. At all times the automata is "in" one, and only one, of the states in Q. That is, there is a current state $q_c \in Q$ that describes the active state of the automaton. Initially $q_c = q_0$. When the automata receives an input symbol $\gamma \in \Gamma$, it *may* change its active state q_c and *must* output a symbol $\pi \in \Pi$. Specifically, when the automaton is in state q_c and receives input γ, it responds as follows, in the following order:

1. The current state q_c is updated to q_c' based on the transition function δ. First δ is extended to be *total* (i.e., to have a value for all inputs) by inserting "self-transitions" with the probability of 1 when $\delta(q, \gamma)$ is undefined:

   $$\delta'(q, \gamma) = \begin{cases} \{q \mapsto 1\} & \mathsf{undefined}\ \delta(q, \gamma) \\ \delta(q_c, \gamma) & \text{otherwise} \end{cases}$$

   The new state $q_c' = q$ with the probability of p when $q \mapsto p \in \delta'(q_c, \gamma)$.

2. The output symbol π is determined using the output function λ and *the new state* q_c'. Once again, this differentiates behavior automata from Mealy machines, which use the original state q_c to determine the output symbol. The function λ is extended to be total by inserting the empty propagation record $\langle\rangle$ when $\lambda(q_c', \gamma)$ is undefined:

   $$\lambda'(q, \gamma) = \begin{cases} \langle\rangle & \mathsf{undefined}\ \lambda(q, \gamma) \\ \lambda(q, \gamma) & \text{otherwise} \end{cases}$$

   The behavior automaton outputs symbol $\lambda'(q_c', \gamma)$.

The behavior automaton is then ready to receive and react to a new environment or input symbol.

[DISTRIBUTION STATEMENT A] Approved for public release and unlimited distribution.

10 Generating a Fault Tree from AADL and EMV2 Models

This section addresses the problem of generating a dynamic fault tree (DFT) [8, 34] from an AADL model that includes EMV2 annex subclauses (AADL+EMV2). The technique described herein for generating fault trees is based on using component fault trees, with some adaptions particular to AADL.

10.1 Problem Statement

Given

- an AADL instance model
- a particular *system operation mode* of the system, that is, a one-to-one mapping of component instances to one of their operational modes
- a particular *initial behavior state* of the *entire system* relative to the system operational mode, that is, a one-to-one mapping of component instances to EMV2 error behavior states (such a thing is not currently described in the EMV2 specification or its meta model)
- a particular *top event*, that is, *state transition*, *state activation*, or *propagation* within the system—not necessarily from the top-level component

the problem is how to generate a dynamic fault tree that enables answering the following two questions:

1. What are the shortest event sequences that lead to the occurrence of the top event? Colloquially, what is the fastest way to crash the system?
2. What is the probability of the top event occurring? Colloquially, what is the probability of crashing the system?

Collectively these inputs are known as the *fault tree query*.

Keep in mind that *a fault tree is not a simulation of the modeled system*; such simulation is a separate problem addressed by automata and other techniques. The fault tree answers the above questions only about the specific operational and behavioral states of the system.

10.1.1 Implications: Operational Modes

Note that the problem statement constrains the fault tree to consider event sequences within the given system operation mode only. The possibility of the system changing modes is not even considered by the constructed fault tree. A system with multiple system operation modes would need to have a fault tree generated in each system operation mode of interest. Because system operation modes are used in AADL to model system reconfiguration (e.g., due to component failures), this means that when considering common example systems, the fault tree generated by the approach described in this work may be different from those usually presented in the literature.

For example, a "k of N" system design (e.g., a system with five redundant sensors that can function as long as three sensors are operational) is usually presented as resulting in a fault tree utilizing a k-of-N gate.[1] For the example just mentioned, a 3-of-5 gate would be used because failure occurs when three or more of the sensors have failed. In the approach described

[1] A k-of-N gate can be reduced to an OR gate fed by AND gates based on expanding all the combinations in which k of the inputs are true.

[DISTRIBUTION STATEMENT A] Approved for public release and unlimited distribution.

herein, the AADL model would contain 32 system operation modes: one for each configuration of the system based on which sensors are currently operational. A forest of fault trees would be obtained, one for each system operation mode; the relationships among the different system operation modes, and, thus, the relationships among their fault trees, are not explored by this work. Specifically, though, no 3-of-5 gate equivalent to the one traditionally presented would be evident in any of the resulting fault trees.

10.1.1.1 Handling Modes and Reconfiguration

By ignoring mode changes, the approach described herein is also not capable of producing fault trees that would traditionally contain spare gates of the hot, warm, or cold variety. To do so requires knowledge of *design intent* that is not evident in a standard AADL model, even one extended with EMV2 annex clauses. Of course, the different configurations (system operation modes) of such a system can be analyzed separately, but no information about the reconfiguration itself would be available.

AADL modes and mode changes present issues with timing and inter-component synchronization that do not yet have a formal semantics. Such an understanding is vital to ensuring that the fault tree generated for such a system considers events in the proper order.

10.2 Component Fault Trees for AADL Components

Following the basic principles of CFTs, one component fault tree is generated per AADL component. Recall, however, that the syntactic model of AADL is derived from the AADL system instance model (i.e., an instantiated AADL system). Although a CFT as originally described by Kaiser and colleagues [15] is intended to be associated with a generic component description (e.g., an AADL declarative component classifier) and reused for each instance of such a component in the system, the reality of analyzing AADL models does not support this. Instead, a new CFT is generated for each component in the AADL model, regardless of whether a CFT for a component with the same classifier type has already been created. There is no inherent difficulty with this, but it does undercut the reusability claim made in support of CFTs—in fact, the classifier of a component plays no actual role in the process at all! The fact that each AADL component has its own CFT, however, does enable a different sort of reuse when it comes time to connect the CFTs together; see Section 10.3.2.

10.2.1 First Impressions: AADL and EMV2 Models and Component Fault Trees

Existing approaches to deriving fault trees from AADL and EMV2 models employ a backward search technique on the AADL instance model [7, 10, 20]. Generally speaking, an output propagation is followed backwards to find the conditions that cause the propagation. The causes of those conditions are further traced, and so on, until an error event or unconnected in port is reached. While conceptually simple, this process can become complex due to

- the notion of state in the EMV2 behavior model
- the interactions among states and type propagations
- the fact that causality often passes through many components within the model

CFTs *return the focus of analysis to the components*, thus bringing the potential to simplify the process of deriving a fault tree from an AADL model. In particular, by design, they allow the fault tree to be generated on a per–component-instance basis:

- Behavior (error and core AADL) that comes from the component is represented by gates *within* the CFT. This information is derived solely from attributes associated with the component.

[DISTRIBUTION STATEMENT A] Approved for public release and unlimited distribution.

- Behavior (error and core AADL) that comes from a subcomponent is described by the CFT for that subcomponent; within the CFT of the containing component, it is represented by a single node for that subcomponent's CFT.
- Additionally, as is shown below, CFTs can further abstract a component's behavior on a *per-state* basis.

10.3 AADL to CFTs: An Overview

First, the gates needed to convert AADL to a fault tree are limited to AND, OR, and PAND.[2] The Boolean literals TRUE and FALSE, however, are also used, something that may be novel within this work. Also, as the final fault tree is derived from a CFT, the fault trees created by the process described herein are actually cause–effect graphs.

The AADL instance model must first be projected into the system operation mode provided as part of the fault tree query. This is a standard AADL instance model operation. It limits the model to containing exactly those components, connections, and other features that exist in the specific system operation mode.

The following subsections describe the general structure of CFTs generated from AADL. The full specification, in terms of the formalism presented in Section 2.2.2, is in the following sections.

10.3.1 Instance CFT

As mentioned already, one CFT is associated with each AADL component (instance). To differentiate this CFT from roles introduced below, this CFT is known as an *instance CFT*. The purpose of this CFT is to abstract the failure behavior of the component as described by the core AADL and EMV2 structures in the component. The ports of the instance CFT are therefore the interface to this abstraction and reflect the information that is produced or received by the component:

- There is one *output port* for each behavior state in the component. The fault tree rooted at this port describes the condition under which the behavior state becomes the current state of the component. (For brevity below, this is equivalent to writing "The port indicates when the behavior state becomes the current state.")
- There is one *output port* for each type declared for each out propagation of the component. The port indicates when the type is propagated by the propagation point. For example, if the component declares the propagation `measurement : out propagation {Missing, Delayed};`, then there would be two out ports added to the instance CFT: one for the possibility that error type `Missing` is propagated by `measurement` and one for the possibility that `Delayed` is propagated.
- Similarly, there is one *input port* for each type declared for each in propagation of the component. The port indicates when the type propagation is received by the propagation point.
- Less obviously, there is also one *input port* for each behavior state in the component. This port is used to manage the initial behavior state of the entire system (see Section 10.1). Its full use is described in Section 10.3.4.

Because the exact behavior of the component depends on its current error behavior state, the instance CFT contains one nested *instance state CFT* for each state in the component.

[2] Here the gate names are written in sans-serif typeface because they refer to specific gate identifiers that may be used within the CFT. See Section 11.1.

[DISTRIBUTION STATEMENT A] Approved for public release and unlimited distribution.

10.3.1.1 Instance State CFT

An instance state CFT abstracts the behavior of a single error behavior state of a component. As such, it has some of the same ports as the instance CFT:

- There is a single *output port* indicating the state is the current state.
- There is one *output port* for each type declared for each out propagation of the component.
- There is one *input port* for each type declared for each in propagation of the component.
- There is an *input port* indicating that the state should become the current state.

Because the CFT represents behavior internal to the component, transitions are also part of the interface:

- There is one *output port* for each state that indicates the condition under which the current state transitions to the state. That is, the port associated with state B on the instance state CFT associated with state A of component K indicates when K transitions from state A to state B.

The formal details of this are described in Section 11.2.2.

10.3.1.2 Gates and Edges

A minimal amount of "logic" in the form of fault tree gates is added to the instance CFT. One of the states of the component must be current, and the component's behavior is then determined by the corresponding instance state CFT. Thus, the outputs of the different instance state CFTs can be ORed together. Generally speaking,

- each out propagation port is connected to an OR gate whose inputs are the corresponding output ports on the nested instance state CFTs (recall that a port can only receive a single incoming edge)
- each in propagation port is a direct input to the corresponding input propagation ports on the nested instance state CFTs (there is no problem with a port having multiple outgoing edges)
- the current state port of each instance state CFT connects to the corresponding behavior state output port of the instance CFT
- the initial state ports of the instance CFT connect to an OR gate connecting to the activation port of the corresponding instance state CFT
- the transition output port of each instance state CFT is connected to the OR gate connected to the activation port of the corresponding state CFT. That is, the B transition port of each instance state CFT is connected to the OR gate connected to the activation port of instance state CFT associated with state B.

The formal details of this are described in Section 11.4.

10.3.2 Subcomponents

The instance state CFTs represent an abstraction layer not explicitly represented in the AADL model. This use of CFTs seems novel to our approach. The (traditional) subcomponent CFTs of the CFT, however, have been displaced. Here, a subcomponent of an AADL component is represented by the instance CFT associated with it. There is an interesting problem:

[DISTRIBUTION STATEMENT A] Approved for public release and unlimited distribution.

- On the one hand, a subcomponent exists independently from—that is, its existence does not depend on—the error behavior states of its containing component. Its behavior is also unaffected by the behavior of the containing component; importantly, the current state of the subcomponent is unaffected by the state transitions of the containing component.
- On the other hand, the observable behavior of the subcomponent may be used to determine the behavior of the containing component. This *may* depend on the current state of the containing component.

Because the subcomponent may be used by each instance state CFT, and the connections to each may be different based on the different transitions and propagations in each state, it is tempting to simply duplicate the subcomponent across each of the instance state CFTs. But this runs afoul of the first bullet point above because duplicating the subcomponent would create the problem of synchronizing the subcomponent's current state in *each* instance state CFT.

The solution is to *share* a single instance CFT of each subcomponent among the different instance state CFTs of the containing component's instance CFT. The formal details of this are described in Section 11.3.3.

10.3.2.1 Gates and Edges

The subgraphs—gates and edges—connecting the ports of an instance state CFT to its (shared) subcomponents mainly derive from the conditions of transitions and outgoing propagations associated with the state:

- The in propagation ports of the instance state CFT and the out propagation ports of the subcomponent CFTs are the basic "trigger expressions" of the conditions.
- The logic gates OR and AND construct the remainder of the condition expression, which is then rooted at the appropriate transition or out propagation port (although see point below).

It is important that the output ports of an instance state CFT only become "active" when the state associated with the CFT is active. Therefore, every out port of an instance state CFT is guarded by a PAND (i.e., priority and) gate:

- The A input of the gate receives an edge from the activation port of the instance state CFT. That is, *the state must become the current state before its output is turned on.*
- The B input of the gate receives the root of the actual fault tree that would be connected to the port.

The formal details of this are described in Section 11.4.2.

10.3.3 Inter-Component Edges

Edges between instance CFTs are guided by EMV2 propagation paths, which are similar to AADL semantic connections: they describe a propagation starting from the most nested source component (i.e., a thread), up though its containing components (i.e., thread groups, processes, systems), across to another sibling component (i.e., a process or system), and then back down nested components to the ultimate destination component. Processing a propagation path adds

- edges from instance CFT out ports to out ports of their containing instance state CFTs during the journey "up"

[DISTRIBUTION STATEMENT A] Approved for public release and unlimited distribution.

- edges from instance CFT out ports to instance CFT in ports during the journey "across"
- edges from instance state CFT in ports to subcomponent instance CFT in ports during the journey "down"

Keep in mind that the following edges exist *regardless* of any propagation paths that may be added:

- edges from instance state CFT propagation output ports to instance state propagation output ports
- edges from instance CFT propagation input ports to instance state CFT propagation input ports

The formal details of this are described in Section 11.5.

10.3.4 The Fault Tree and the Initial System Behavior State

Generally speaking, the fault tree (really CEG) is produced as described in Section 2.2.1. The root port for the traversal is the out port associated with the component state, state transition, or out propagation provided by the fault tree query. It needs to be pointed out, however, that the collection of CFTs is *incomplete* at this point: paths leading to instance CFT behavior state input ports fail to connect to a gate output or basic event. This is resolved during the traversal based on *the initial system behavior state* provided as part of the fault tree query. When the child of a behavior state port is required, the initial system behavior state is consulted:

- If the port is associated with a component state that is part of the initial system behavior state, then the TRUE literal gate is used.
- If the port is not part of the initial system behavior state, then the FALSE literal gate is used.

The fault tree obtained from the traversal is then simplified via the application of semantic transformation rules to eliminate the use of TRUE and FALSE gates: for example, TRUE AND A = A. It is possible to remove all uses of TRUE and FALSE except for the cases where they become the root, and sole, node of the fault tree. This can happen, for example, when the queried transition/propagation/state

- is immediately available from the initial system behavior state (TRUE)
- is impossible from the initial system behavior state (FALSE)

The formal details of this process, and the exact optimizations performed, are described in Sections 12.2 and 12.3, respectively.

10.4 Additional Observations About This Approach

As mentioned above, existing approaches to generating fault trees from component architecture models, including AADL, directly produce a fault tree using a backwards search from an output port of an architectural component and interpreting the semantics of the component model during the backwards traversal. In a sense, this is a *global* analysis of the system instance and fails to leverage the modularization provided by the components themselves. In contrast, the CFT-based approach described herein does utilize the modularization provided by both components and error behavior states: subsections of the fault tree are generated per-state and per-component, utilizing only information declared in that component. It is a *local* and *composable* analysis. This makes the analysis easier to define, implement, and reason

[DISTRIBUTION STATEMENT A] Approved for public release and unlimited distribution.

about. The backwards search is performed starting from a root port in the CFT (see above) instead of from a feature in the component model.

10.4.1 Trade-Offs

There are trade-offs to using this approach:

- A CFT contains all potentially interesting events that may occur within a component; that is, the CFT is not focused on the events necessary for the specific fault tree being generated. This can be seen as performing unnecessary work. However, see the discussion below regarding reusability.
- The generated fault tree explores *all* the potential paths in the system state, even those that do not apply to the initial behavior state. These paths are ultimately disabled by the fact that the literal FALSE values are connected to the initial state ports of instance CFTs. Furthermore, the existing analyses in the literature *do visit* these paths; they just do not incorporate them into the generated fault tree.

Said more succinctly, this approach pursues ease of analysis at the expense of generating a final fault tree that has "dead paths." The fault tree that results from the above analysis steps can be simplified to "pull up" the TRUE and FALSE literal values. In particular, the FALSE literals cause unreachable "dead paths" to be pruned from the fault tree.

10.4.2 Reusability

CFTs are introduced as a technique to generate a partial fault tree that can be associated with a component *type*. The CFT serves as an abstraction of the component's behaviors: each out port is a predicate over the component's in ports. A component type can be reused in many different system designs, and its CFT is carried with it. The component's CFT needs to be generated only once but can be reused in many different contexts.

It is tempting to apply this concept directly to AADL by generating CFTs based on declarative AADL component classifiers. As mentioned already in Section 10.2, this is not feasible, making the full reuse scenario intended for CFTs inapplicable to AADL.

A lesser amount of reusability, however, is possible in the approach presented here due to the CFT being parameterized by the initial behavior state. Only one set of instance CFTs needs to be created per system operation mode of the system. As pointed out above, the generated CFTs *are not* focused to any particular output or event of the system. Furthermore, the initial state ports are left "disconnected." Thus, once a system instance model is projected into a specific system operation mode, and analyzed to produce CFTs, any number of fault trees using different initial behavior states and root ports can be produced without producing a new set of CFTs.

10.5 Limitations

The expressiveness of AADL and EMV2 is beyond what can be expressed within a fault tree, either static or dynamic. The production of a fault tree from AADL and EMV2 models is thus subject to several limitations:

- As already discussed, the full semantics of AADL modes and their effects on error behavior have not been explored. Thus, a fault tree is always from a particular system operation mode. This is true as well of the existing work that generates fault trees from AADL, but this work is the first to directly acknowledge the limitation.
- Probabilistic branched transitions are not handled by the fault tree generation process. They are simply ignored. They require the ability to express probabilistic choice in the

[DISTRIBUTION STATEMENT A] Approved for public release and unlimited distribution.

middle of a fault tree. This is true as well of the existing work that generates fault trees from AADL, but this work is the first to directly acknowledge the limitation.

- Operations requiring logical negation cannot be expressed in the generated fault tree. This is generally known as *non-coherence* in the literature; see Sharvia and Papadopoulos for a history of the problem and a summary of recent approaches to address it [29]. Generally, techniques exist to analyze non-coherent static fault trees, but there are no widely discussed techniques to analyze non-coherent dynamic fault trees.[3] It is unclear what tool support, if any, exists for the analysis of non-coherent fault trees, either static or dynamic.

  From the point of view of generating a fault tree from AADL, any feature that requires negation cannot be considered:

  - The requirement that condition expressions consisting of a single propagation trigger be evaluated as if all other propagations are not occurring (see Section 7.1) cannot be enforced. This requirement is neither enforced nor discussed by the existing approaches to generate fault trees from AADL.
  - The `xor` operator cannot be translated to a fault tree.
  - The primitive `orless` cannot be translated to a fault tree.
  - The primitive "all but" `all`$-n$ cannot be translated to a fault tree.

[3] But see Schilling [31]. The authors became aware of this work too late for it to have an influence on the work presented herein.

[DISTRIBUTION STATEMENT A] Approved for public release and unlimited distribution.

11 The Instance CFT and Instance State CFT

This section elaborates on the mathematical structure of instance and instance state CFTs as introduced in Section 10.3.1. It describes the translation of an AADL model component k into a component CFT structure in detail.

11.1 CFT Defined

Recall that a CFT is described by a 4-tuple (N, G, S, E) in Section 2.2.2. Here, the representation of a CFT is extended to be a 5-tuple (N, G, O, S, E) to better capture the association between a gate and its logical formula:

- Internal nodes, input ports, output ports, and gates are represented by distinct (non-intersecting) sets of symbols: $N \subseteq \mathcal{A}$, $G \subseteq \mathcal{A}$, and $N \cap G = \emptyset$. Furthermore, the symbols are *globally unique* among all the CFTs produced for the components in $\mathbf{K}$.
  - Whenever a *fresh* symbol is needed, it is unique among all symbols used in all CFTs created for components in $\mathbf{K}$.
  - N is partitioned into the disjoint subsets N_{Intern} of internal events, N_{In} of input ports, and N_{Out} of output ports.
  - Set $G_{\mathsf{In}} = \bigcup_{g \in G} \{g.\mathsf{in_i}\}$ and set $G_{\mathsf{Out}} = \bigcup_{g \in G} \{g.\mathsf{out}\}$.
- Function $O : G \rightarrow \mathsf{Op}$ is a total function mapping a gate node to the operator that it represents, drawn from the set Op. The members of G are instances of the following gates: TRUE, FALSE, AND, OR, and PAND. That is,

  $$\mathsf{Op} = \{\mathsf{TRUE}, \mathsf{FALSE}, \mathsf{AND}, \mathsf{OR}, \mathsf{PAND}\}$$

  The formula associated with each gate and the constraints on the number of children of each gate are shown in Figure 11.1. The formulas are given as expressions in Merle's algebra. Note that AND and OR gates are allowed to have a single child. This allowance makes algorithmic construction of the fault trees easier by eliminating the need for testing for corner cases. These cases are removed by the fault tree simplification step described in Section 12.3. In particular, $\mathsf{AND}(x) = x$ and $\mathsf{OR}(x) = x$.
- The specific representation for subcomponents depends on whether the CFT is an instance CFT or an instance state CFT; in either case, $S \subset \mathcal{A}$.
  - In an instance CFT, subcomponents are directly represented by the behavior state symbols.
  - In an instance state CFT, subcomponents are directly represented by semantic components.
  - Set $S_{\mathsf{In}} = \bigcup_{s \in S} \{s.\mathsf{in_i}\}$ and set $S_{\mathsf{Out}} = \bigcup_{s \in S} \{s.\mathsf{out}_i\}$.
- $E \subseteq \mathsf{Src} \times \mathsf{Dst}$, where
  - $\mathsf{Src} = N_{\mathsf{Intern}} \cup N_{\mathsf{In}} \cup G_{\mathsf{Out}} \cup S_{\mathsf{Out}}$
  - $\mathsf{Dst} = N_{\mathsf{Out}} \cup G_{\mathsf{In}} \cup S_{\mathsf{In}}$

  For $(s, d) \in E$, s is the source of the edge and d is the destination of the edge. That is, edges point towards the output ports of the CFT.

[DISTRIBUTION STATEMENT A] Approved for public release and unlimited distribution.

Gate	Number of Inputs	Formula
TRUE	0	$\mathsf{out} = \top$
FALSE	0	$\mathsf{out} = \bot$
AND	$n \geq 1$	$\mathsf{out} = \mathsf{in}_1 \cdot \cdots \cdot \mathsf{in}_n$
OR	$n \geq 1$	$\mathsf{out} = \mathsf{in}_1 + \cdots + \mathsf{in}_n$
PAND	2	$\mathsf{out} = \mathsf{in}_2 \cdot (\mathsf{in}_1 \lhd \mathsf{in}_2)$

Figure 11.1: The Gates Used in the Constructed CFTs

11.2 The Instance CFT Interface

A component k with $\square = [\![k]\!]_{\mathsf{K}}$ has a component fault tree $CFT_\square = (N_\square, G_\square, O_\square, S_\square, E_\square)$. The specific sets $N_\square = N^\square_{\mathsf{Intern}} \cup N^\square_{\mathsf{In}} \cup N^\square_{\mathsf{Out}}$ and $S_\square$ for semantic component $\square$ are now defined. The general pattern is to define *abstraction functions*: *total* functions that map semantic aspects of the component to symbols in the CFT. This way, the actual symbols need not be specifically identified. As much as is possible, the existing sets of semantic symbols from the EMV2 denotation are reused.

11.2.1 Events

The set of error events defined in the component is already identified with the set $\mathcal{V}_\square$; see Section 4.2.1.

$$N^\square_{\mathsf{Intern}} = \mathcal{V}_\square$$

11.2.2 Ports

There are two distinct subsets of input ports to the CFT: one for ports that manage the initial state of the component, and one for ports that represent the in propagations of the component. The scheme for propagations is to have a set of ports for each propagation point: one port for each declared propagated type. Because no two elements of a type set may have a containment relationship with each other, these ports all represent distinct occurrences. Because *negative* behavior is not being modeled in the generated fault tree, no ports are generated to represent the {`noerror`} case of a propagation.

- For the initial state of the component, the function $\mathsf{Init}_\square : \mathcal{Q}_\square \to \mathcal{A}$ uniquely maps behavior states of the component to symbols.[1]
- The ports representing in propagations depend on the declared propagation points *and* the set of types they are declared to propagate. There is one port for each potentially propagated type. The function $\mathsf{In}_\square : \mathcal{A} \times \mathcal{T} \to \mathcal{A}$ uniquely maps each propagation symbol–type pair to a symbol. For in propagations, the symbols from $\mathcal{I}_\square$ are used to represent the propagation point. For $f \in \mathbf{D}^k_{\mathsf{In}}$, the types are drawn from $[\![\mathbf{D}_{\triangleright\mathsf{PP}}(f)]\!]_{\mathsf{Set}}$. Thus, the abstraction has a value for each pair in $\bigcup_{f \in \mathbf{D}^k_{\mathsf{In}}} (\{[\![f]\!]^\square_{\mathsf{Field}}\} \times [\![\mathbf{D}_{\triangleright\mathsf{PP}}(f)]\!]_{\mathsf{Set}})$.

Thus,

$$N^\square_{\mathsf{In}} = \mathsf{ran}\ \mathsf{Init}_\square \cup \mathsf{ran}\ \mathsf{In}_\square$$

There are two distinct subsets of output ports to the CFT: one for ports that report the current state of the component and one for ports that represent the out propagations of the component.

1. For the current state of the component, the function $\mathsf{Current}_\square : \mathcal{Q}_\square \to \mathcal{A}$ uniquely maps behavior states of the component to symbols.

[1] Unlike with events, the set of states $\mathcal{Q}_\square$ cannot be directly used here because there are two sets of ports based on states within the component. Further, the symbols in $\mathcal{Q}_\square$ are used to identify the subcomponents of the CFT.

[DISTRIBUTION STATEMENT A] Approved for public release and unlimited distribution.

2. The ports representing out propagations are handled similarly to the in propagations, but propagation points are mapped to symbols using $[\![\cdot]\!]^{\square}_{\mathsf{Field}\triangleright}$ instead of $[\![\cdot]\!]^{\square}_{\mathsf{Field}}$. The function $\mathsf{Out}_{\square} : \mathcal{A} \times \mathcal{T} \to \mathcal{A}$ uniquely maps each pair in $\bigcup_{f \in \mathbf{D}^{k}_{\mathsf{Out}}} (\{[\![f]\!]^{\square}_{\mathsf{Field}\triangleright}\} \times [\![\mathbf{D}_{\mathsf{PP}\triangleright}(f)]\!]_{\mathsf{Set}})$ to a symbol.

Thus,

$$N^{\square}_{\mathsf{Out}} = \mathsf{ran}\ \mathsf{Current}_{\square} \cup \mathsf{ran}\ \mathsf{Out}_{\square}$$

11.2.3 Subcomponents

There is one subcomponent CFT for each behavior state of the component; thus, it is simply the case that

$$S_{\square} = \mathcal{Q}_{\square}$$

Each subcomponent $s \in S_{\square}$ (i.e., $q \in Q_{\square}$) *maps directly* to the instance state CFT $CFT^{\square}_{q} = (N^{\square}_{q}, G^{\square}_{q}, O^{\square}_{q}, S^{\square}_{q}, E^{\square}_{q})$, described below. Unlike the original presentation of component fault trees [15], note that subcomponent port $q.\mathsf{p}$ *maps directly* to the equivalent port in $N^{\square}_{q}$:

$$S^{\square}_{\mathsf{in}} = \bigcup_{q \in \mathcal{Q}_{\square}} N^{\square}_{q\ \mathsf{In}} \qquad S^{\square}_{\mathsf{Out}} = \bigcup_{q \in \mathcal{Q}_{\square}} N^{\square}_{q\ \mathsf{Out}}$$

That is, the ports of the instance state CFT are *directly connected* to nodes in the instance CFT, cutting out the "middle man" of the subcomponent. This is possible because there is no reuse of referenced CFTs by subcomponents within the overall CFT structure.

11.3 The Instance State CFT Interface

The specific sets $N^{\square}_{q} = N^{\square}_{q\ \mathsf{Intern}} \cup N^{\square}_{q\ \mathsf{In}} \cup N^{\square}_{q\ \mathsf{Out}}$ and $S^{\square}_{q}$ for the instance state CFT representing state q of semantic component $\square$ are now defined.

11.3.1 Events

As described above, the error events of $\square$ are described as internal events of the instance CFT. It is thus the case that

$$N^{\square}_{q\ \mathsf{Intern}} = \emptyset$$

The events of $CFT_{\square}$, however, are needed by the fault tree structures generated within $CFT^{\square}_{q}$ to represent the expressions from transition and propagation declarations. The instance state CFT, therefore, has one input port for each internal event of the parent instance CFT. Function $\mathsf{Event}^{\square}_{q} : \mathcal{V}_{\square} \to \mathcal{A}$ *uniquely maps* each symbol representing a declared event to a symbol representing an input port event in the instance state CFT.

11.3.2 Ports

The instance state CFT $CFT^{\square}_{q}$ has the same propagation-based input and output ports as does the instance CFT. Thus, the analogous abstraction functions $\mathsf{In}^{\square}_{q} : \mathcal{A} \times \mathcal{T} \to \mathcal{A}$ and $\mathsf{Out}^{\square}_{q} : \mathcal{A} \times \mathcal{T} \to \mathcal{A}$ are used to identify the ports in the instance state CFT.

In addition, $CFT^{\square}_{q}$ has

- one input port indicating "state activation" (i.e., that the represented state should become the current state of the component). The abstraction function $\mathsf{Activate}_{\square} : \mathcal{Q}_{\square} \to \mathcal{A}$ identifies the port. (This function is shared by all the instance state CFTs for $\square$.)
- one output port indicating that the represented state is the current state of the component. The abstraction function $\mathsf{Active}_{\square} : \mathcal{Q}_{\square} \to \mathcal{A}$ identifies the port. (This function is shared by all the instance state CFTs for $\square$.)

[DISTRIBUTION STATEMENT A] Approved for public release and unlimited distribution.

- one output port for each *other* state of the component that indicates that the current state should transition to that state. There is no value to a self-transition port, and in fact, loops in fault trees should be avoided. The abstraction function $\mathsf{Trans}_q^{\square} : (\mathcal{Q}_{\square} \setminus \{q\}) \to \mathcal{A}$ uniquely maps destination states to transition ports.

Thus, the sets of input and output ports for $CFT_q^{\square}$ are

$$N_{q\ \mathsf{In}}^{\square} = \{\mathsf{Activate}_{\square}(q)\} \cup \mathsf{ran}\ \mathsf{Event}_q^{\square} \cup \mathsf{ran}\ \mathsf{In}_{\mathsf{q}}^{\square}$$
$$N_{q\ \mathsf{Out}}^{\square} = \{\mathsf{Active}_{\square}(q)\} \cup \mathsf{ran}\ \mathsf{Out}_{\mathsf{q}}^{\square} \cup \mathsf{ran}\ \mathsf{Trans}_{\mathsf{q}}^{\square}$$

11.3.3 Subcomponents

As described in Section 10.3.2, the instance CFT of each subcomponent is *shared* by all the instance state CFTs. To this end,

$$S_q^{\square} = \mathsf{Sub}(\square)$$

Similar to instance state CFTs themselves, each subcomponent $s \in S_{\square}$, equivalently, $\square' \in \mathsf{Sub}(\square)$, *maps directly* to the instance CFT $CFT_{\square'}$. In particular, subcomponent port $\square'.\mathsf{p}$ maps directly to the equivalent port in $N^{\square'}$. That is,

$$S_{q\ \mathsf{in}}^{\square} = \bigcup_{\square' \in \mathsf{Sub}(\square)} N_{\mathsf{In}}^{\square'} \qquad S_{q\ \mathsf{Out}}^{\square} = \bigcup_{\square' \in \mathsf{Sub}(\square)} N_{\mathsf{Out}}^{\square'}$$

Namely, the ports of the subcomponent's instance CFT are directly connected to nodes in the instance state CFT. As described below, the input ports of each subcomponent instance CFT are directly connected to OR gates to allow the possibility of multiple incoming connections. This reuse and simultaneous connection of each subcomponent to multiple state representations is possible only because exactly one of the instance state CFTs is "active" at any given time.

11.4 Gates and Edges

The set of gates $G^{\square}$ and edges $E^{\square}$ are now concurrently defined. For ease of presentation, some notational statements are introduced that *imply* edges between gates and ports. For a CFT (N, G, O, S, E), the following properties hold:

- when $d \in \mathsf{Dst}$, and $s \in \mathsf{Src}$, then "$d = s$" if and only if $(s, d) \in E$
- when $d \in \mathsf{Dst}$, $\mathsf{X} \in \mathsf{Op}$, and $s_i \in \mathsf{Src}$, then "$d = \mathsf{X}(s_1, \ldots s_n)$" if and only if there exists a fresh gate $g \in G$ such that
  - g is an X gate: $O(g) = \mathsf{X}$
  - the output of g is connected to d: $(g.\mathsf{out}, d) \in E$
  - the sources s_i are connected to the input ports of g: $(s_i, g.\mathsf{in}_1) \in E$
- notation is abused by allowing the gate label X to be used with an iterative subscript: for example, $\mathsf{AND}_{x \in S}(f(x))$ implies an AND gate whose input ports are the destinations of edges that start at the nodes determined by applying function f to each element of set S
- notation is further abused by allowing $\mathsf{X}(\ldots)$ to appear in a context expecting a member of Src, in which case the actual source value is $g.\mathsf{out}$, where g is the node associated with the gate
- when $g \in G$, and $s \in \mathsf{Src}$, then "INPUT(g, s)" if and only if there exists an $i \geq 1$ such that "$g.\mathsf{in}_i = s$"

[DISTRIBUTION STATEMENT A] Approved for public release and unlimited distribution.

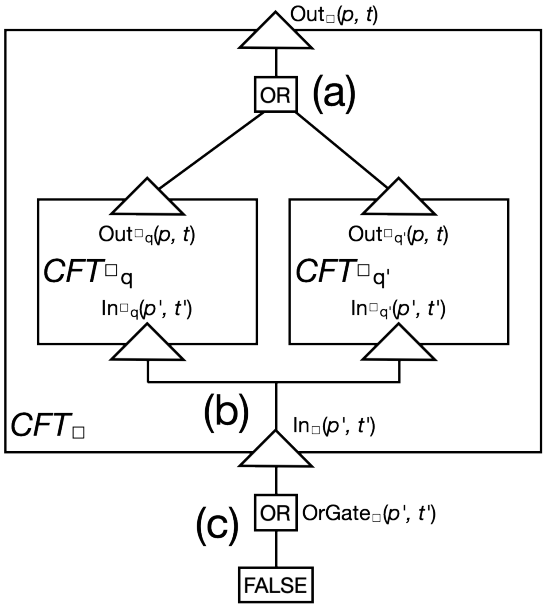

Figure 11.2: Example Showing the Propagation Port Connections Between an Instance CFT and Its Instance State CFTs

11.4.1 The Instance CFT

Section 10.3.1.2 generally describes the edges internal to an instance CFT. The exact set of gates and edges for $CFT_\square = (N_\square, G_\square, O_\square, \mathcal{Q}_\square, E_\square)$ is described by the following statements:

- For all $(p,t) \in \mathsf{dom\ Out}_\square$, $\mathsf{Out}_\square(p,t) = \mathsf{OR}_{q \in \mathcal{Q}_\square}(\mathsf{Out}_q^\square(p,t))$. See Figure 11.2(a).
- For all $(p,t) \in \mathsf{dom\ In}_\square$, for all $q \in \mathcal{Q}_\square$, $\mathsf{In}_q^\square(p,t) = \mathsf{In}_\square(p,t)$. See Figure 11.2(b).
- For all $q \in Q_\square$, $\mathsf{Current}_\square(q) = \mathsf{Active}_\square(q)$. See Figure 11.3(a).
- For all $q' \in \mathcal{Q}_\square$, $\mathsf{Activate}_\square(q') = \mathsf{OR}\left(\mathsf{Init}_\square(q'), \mathsf{OR}_{q \in \mathcal{Q}_\square \setminus \{q'\}}(\mathsf{Trans}_q^\square(q'))\right)$. See Figure 11.4.

In addition, the internal events of the instance CFT need to be connected to the event ports of the instance state CFTs:

- For all $e \in \mathcal{V}_\square$, for all $q \in Q_\square$, $\mathsf{Event}_q^\square(e) = e$. See Figure 11.3(b).

Finally, the in propagation ports of the instance CFT need to be primed to receive edges from multiple instance state CFT input ports via the addition of an OR gate. The first input of the gate is connected to FALSE to account for the case where no actual connections are made to the port. The identity of the gate itself is abstracted via the function $\mathsf{OrGate}_\square : \mathcal{A} \times \mathcal{T} \to \mathcal{A}$, which maps each member of $\mathsf{dom\ In}_\square$ to a fresh gate symbol.

- For all $(p,t) \in \mathsf{dom\ In}_\square$,
  - $(\mathsf{OrGate}_\square(p,t).\mathsf{out}, \mathsf{In}_\square(p,t)) \in E_\square$
  - $O(\mathsf{OrGate}_\square(p,t)) = \mathsf{OR}$
  - $\mathsf{OrGate}_\square(p,t).\mathsf{in}_1 = \mathsf{FALSE}$

  See Figure 11.2(c).

[DISTRIBUTION STATEMENT A] Approved for public release and unlimited distribution.

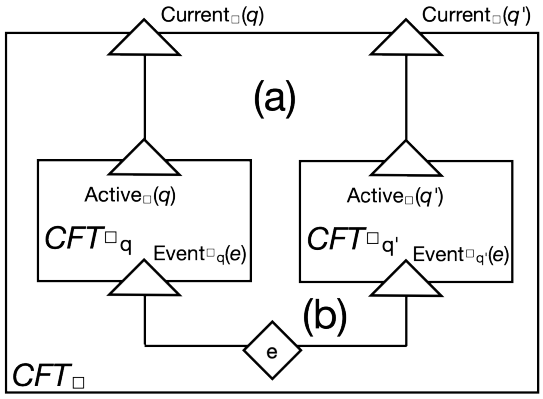

Figure 11.3: Example Showing the Current State Edges and Event Edges in an Instance CFT

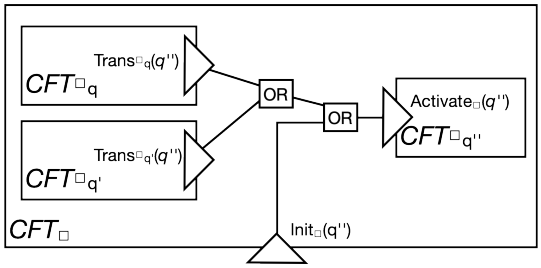

Figure 11.4: Example Showing the Edges That Model the Transition to State q''

In Section 11.5, more inputs to $\mathsf{OrGate}_{\square}(p, t)$ that describe propagation paths are asserted to exist.

11.4.2 The Instance State CFT

The internal structure of instance state CFT $CFT_q^{\square} = (N_q^{\square}, G_q^{\square}, O_q^{\square}, \mathsf{Sub}(\square), E_q^{\square})$ is mostly determined by the transition and propagation declarations of component k, where $\square = [\![k]\!]_{\mathsf{K}}$. There is a single direct connection between the state activation input port and the current state output port:

- $\mathsf{Active}_{\square}(q) = \mathsf{Activate}_{\square}(q)$. See Figure 11.5(a).

Like the in propagation ports of the instance CFT, the out propagation ports of the instance

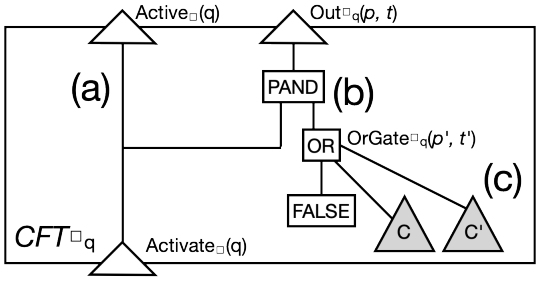

Figure 11.5: Example Showing the CFT Edges Connected to Activation Ports and Out Propagation Ports Within the Instance State CFT $CFT_q^{\square}$

[DISTRIBUTION STATEMENT A] Approved for public release and unlimited distribution.

state CFT need to be primed to receive edges from multiple sources, in this case from outgoing propagation conditions (see Section 11.4.5) and from propagation paths (see Section 11.5). They are connected to an OR gate guarded by a PAND gate that makes sure the current state is the active state. The first input of the OR gate is connected to FALSE to account for the case where no actual connections are made to the port. The identity of the gate itself is abstracted via the function $\mathsf{OrGate}_q^{\square} : \mathcal{A} \times \mathcal{T} \to \mathcal{A}$, which maps each member of $\mathsf{dom}\ \mathsf{Out}_q^{\square}$ to a fresh gate symbol.

- For all $(p, t) \in \mathsf{dom}\ \mathsf{Out}_q^{\square}$,
  - $\mathsf{Out}_q^{\square}(p, t) = \mathsf{PAND}(\mathsf{Activate}_{\square}(q), \mathsf{OrGate}_q^{\square}(p, t).\mathsf{out})$
  - $O(\mathsf{OrGate}_q^{\square}(p, t)) = \mathsf{OR}$
  - $\mathsf{OrGate}_q^{\square}(p, t).\mathsf{in}_1 = \mathsf{FALSE}$

  See Figure 11.5(b).

In Section 11.4.5, more inputs to $\mathsf{OrGate}_q^{\square}(p, t)$ that describe outgoing propagation conditions for the port are asserted to exist, and in Section 11.5, inputs are asserted to exist to account for propagation paths; see Figure 11.5(c).

11.4.3 Fault Tree Semantics of Expressions

To describe the gates and edges produced by transitions and propagations, the gates and edges produced by a condition expression must be described. This section develops a denotational semantics that converts a condition expression into a statement about gates and edges. As discussed in Section 10.5, analysis of fault trees that contain negation is difficult, and tool support for analysis of dynamic fault trees that contain negation is non-existent. Thus, as pointed out above, no ports are created for the propagation of {`noerror`}. Similarly, the semantics here do not handle `noevent` or triggers based on {`noerror`}. This limitation specifically means that the following behaviors are not supported by the generated fault trees:

- expressions that use `xor`, `orless`, and `all - `(“all but”)
- silencing of propagations when the condition expression of a transition is a solitary propagation trigger

To the best of our knowledge, none of the existing techniques for deriving fault trees from AADL properly handle these cases either.

The general form of the semantic interpretation function is to operate on a condition $\mathbf{C}_k$ or one of its subexpressions and return an edge source node from **Src**. As usual, the presentation is bottom up towards $\mathbf{C}_k$. An event trigger evaluates to the appropriate event port of the instance state CFT; as stated above, `noevent` is not interpretable.

$$[\![\texttt{event}\ e]\!]^{(\square,q)}_{\mathsf{FT\_Trigger_E}} = \mathsf{Event}_q^{\square}([\![e]\!]^{\square}_{\mathsf{Event}})$$

A propagation trigger evaluates to the output port of an OR gate. The gate is necessary for two reasons:

1. The type set of the trigger may contain more than one type.
2. The port itself may declare the potential to propagate more than one type.

Each type from the port declaration must be tested to determine if it is contained in the type set of the trigger. As described in Section 3.4.1, this test can be made on the syntactic form of the model, that is, within an analysis tool. Each port $\mathsf{In}_q^{\square}(i, t)$ associated with the trigger propagation f whose type t is contained (via ϕ) in the specified type set d is an edge source to an input port of the OR gate. As previously mentioned, propagation triggers based on

{noerror} are not handled. As in Section 6.3, the semantic function $[\![\cdot]\!]_{\mathsf{TS}}$ is used to obtain the containment-testing predicate.

$$\begin{array}{rcl} [\![\texttt{in } f \ ts]\!]^{(\square,q)}_{\mathsf{FT\_Trigger\_P}} & = & \mathsf{OR}_{t \in contained}\left(\mathsf{In}^{\square}_{q}(i,t)\right) \\ & \text{where} & i = [\![f]\!]^{\square}_{\mathsf{Field}} \\ & & d = [\![\mathbf{D}_{\triangleright \mathsf{PP}}(f)]\!]_{\mathsf{Set}} \\ & & \phi = [\![ts]\!]_{\mathsf{TS}} \\ & & contained = \{t \in d \mid \phi(d)(t)\} \end{array}$$

The case for triggers from subcomponent out propagations is similar but complicated by the fact that the port belongs to subcomponent $\square' = [\![s]\!]_{\mathsf{K}}$, where s is the syntactic component that f belongs to. The ports $\mathsf{Out}_{\square'}(i,t)$ come from the subcomponent CFT for $\square'$, which is really the instance CFT $CFT_{\square'}$ itself.

$$\begin{array}{rcl} [\![\texttt{out } f \ ts]\!]^{(\square,q)}_{\mathsf{FT\_Trigger\_P}} & = & \mathsf{OR}_{t \in contained}\left(\mathsf{Out}_{\square'}(i,t)\right) \\ & \text{where} & s = \mathbf{PP}_{\mathsf{K}}(f) \\ & & \square' = [\![s]\!]_{\mathsf{K}} \\ & & i = [\![f]\!]^{\square'}_{\mathsf{Field}\triangleright} \\ & & d = [\![\mathbf{D}_{\mathsf{PP}\triangleright}(f)]\!]_{\mathsf{Set}} \\ & & \phi = [\![ts]\!]_{\mathsf{TS}} \\ & & contained = \{t \in d \mid \phi(d)(t)\} \end{array}$$

Triggers, as a whole, delegate based on the category:

$$\begin{array}{rcl} [\![t \in \mathbf{T}^{k}_{\mathsf{Event}}]\!]^{(\square,q)}_{\mathsf{FT\_Trigger}} & = & [\![t]\!]^{(\square,q)}_{\mathsf{FT\_Trigger\_E}} \\ [\![t \in \mathbf{T}^{k}_{\mathsf{PP}}]\!]^{(\square,q)}_{\mathsf{FT\_Trigger}} & = & [\![t]\!]^{(\square,q)}_{\mathsf{FT\_Trigger\_P}} \end{array}$$

The remaining cases of condition expressions are straightforward:

- Parenthesized expressions just pass through the source node.

$$\begin{array}{rcl} [\![(\ c\ )]\!]^{(\square,q)}_{\mathsf{FT\&}} & = & [\![c]\!]^{(\square,q)}_{\mathsf{FT\_Cond}} \\ [\![t \in \mathbf{T}_{\mathsf{k}}]\!]^{(\square,q)}_{\mathsf{FT\&}} & = & [\![t]\!]^{(\square,q)}_{\mathsf{FT\_Trigger}} \end{array}$$

- Conjunctions assert the existence of an AND gate.

$$\begin{array}{rcl} [\![c_1 \ \&\ c_2]\!]^{(\square,q)}_{\mathsf{FT+}} & = & \mathsf{AND}([\![c_1]\!]^{(\square,q)}_{\mathsf{FT+}}, [\![c_2]\!]^{(\square,q)}_{\mathsf{FT+}}) \\ [\![c]\!]^{(\square,q)}_{\mathsf{FT+}} & = & [\![c]\!]^{(\square,q)}_{\mathsf{FT\&}} \end{array}$$

- Disjunctions assert the existence of an OR gate.

$$\begin{array}{rcl} [\![c_1 \ +\ c_2]\!]^{(\square,q)}_{\mathsf{FT\_Cond}} & = & \mathsf{OR}([\![c_1]\!]^{(\square,q)}_{\mathsf{FT\_Cond}}, [\![c_2]\!]^{(\square,q)}_{\mathsf{FT\_Cond}}) \\ [\![c]\!]^{(\square,q)}_{\mathsf{FT\_Cond}} & = & [\![c]\!]^{(\square,q)}_{\mathsf{FT+}} \end{array}$$

11.4.4 Transitions

A fault tree semantics of the source and target of a transition $\tau \in \mathbf{D}^{k}_{\mathsf{Trans}}$ is also necessary. They both evaluate to predicates that test whether a particular state is a source or a target, respectively. This process is made simple because *branched transition targets are not handled by the translation*—it is unclear how to introduce probabilistic branching into the interior of a fault tree. For both sources and targets, a specific state evaluates to a predicate that tests for that state. The source `all` evaluates to a predicate that always returns `true`. The target

[DISTRIBUTION STATEMENT A] Approved for public release and unlimited distribution.

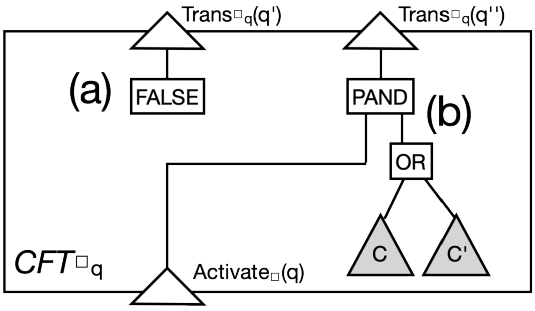

Figure 11.6: Example Showing the CFT Edges Connected to the Transition Ports Within the Instance State CFT $CFT_q^{\square}$

`same state`, however, evaluates to a predicate that always returns false: as stated previously, loops in fault trees are undesirable, so there is no reason to describe staying in the same state (in fact, this scenario is preemptively prevented by subtracting the current state from $\mathcal{Q}_{\square}$ below).

$$\begin{array}{rcl} [\![q_s]\!]^{(\square,q)}_{\mathsf{FT\_Src}} & = & \lambda q'.\, q' = q_s \\ [\![\texttt{all}]\!]^{(\square,q)}_{\mathsf{FT\_Src}} & = & \lambda q'.\, \mathsf{true} \end{array} \qquad \begin{array}{rcl} [\![ts \in \mathbf{Tgt}^k_{\mathsf{State}}]\!]^{(\square,q)}_{\mathsf{FT\_Tgt}} & = & [\![ts]\!]^{(\square,q)}_{\mathsf{FT\_Tgt\_S}} \\ [\![q_t]\!]^{(\square,q)}_{\mathsf{FT\_Tgt\_S}} & = & \lambda q'.\, q' = q_t \\ [\![\texttt{same state}]\!]^{(\square,q)}_{\mathsf{FT\_Tgt\_S}} & = & \lambda q'.\, \mathsf{false} \end{array}$$

Finally, a tree is generated rooted at each transition output port $\mathsf{Trans}^{\square}_q$; there is one such port for each $q_t \in Q_{\square} \setminus \{q\}$. Each port is influenced by all the transitions that start at q and transition to q_t—the set *influencers* below. The condition of each such transition is converted to a fault tree assertion. Because negation is not expressible, silencing of expressions that contain only a single propagation target *is not performed*: specifically, $\mathsf{satisfiedBy}_{\square}$ is *not* used. There are two cases for the final tree:

1. The set *influencers* is empty: The result is simply a FALSE gate to prevent the transition from ever occurring. See Figure 11.6(a).
2. The set *influencers* is not empty: The result is a PAND gate whose first input is the activation port of the instance state CFT and whose second input is a disjunction of all the influencing transition conditions. See Figure 11.6(b).

Thus, the gates and edges for transition ports are given by the following statements:

$$\forall q_t \in \mathcal{Q}_{\square}\ \{q\}.\quad \mathsf{Trans}^{\square}_q(q_t) = \begin{cases} \mathsf{FALSE} & \textit{influencers} = \emptyset \\ \mathsf{PAND}(\mathsf{Activate}_{\square}(q), \textit{conditions}) & \text{otherwise} \end{cases}$$

where

$$\textit{influencers} = \left\{ \tau \in \mathbf{D}^k_{\mathsf{Trans}} \mid [\![\mathbf{Tr}^k_{\mathsf{Src}}(\tau)]\!]^{(\square,q)}_{\mathsf{FT\_Src}}(q) \ \wedge\ [\![\mathbf{Tr}^k_{\mathsf{Tgt}}(\tau)]\!]^{(\square,q)}_{\mathsf{FT\_Tgt}}(q_t) \right\}$$

and

$$\textit{conditions} = \mathsf{OR}_{\tau \in \textit{influencers}} \left([\![\lfloor \mathbf{Tr}^k_{\mathsf{Cond}}(\tau) \rfloor^k_{\mathsf{Cond}}]\!]^{(\square,q)}_{\mathsf{FT\_Cond}} \right)$$

11.4.5 Propagations

Propagations reuse the semantic functions $[\![\cdot]\!]^{(\square,q)}_{\mathsf{FT\_Src}}$ from above and $[\![\cdot]\!]^{\square}_{\mathsf{Prop}}$ from Section 9.5.3, which interprets propagation targets. To review, $[\![\cdot]\!]^{\square}_{\mathsf{Prop}}$ converts a propagation target into a set of index-symbol–type pairs and matches up perfectly with the domain of $\mathsf{Out}^{\square}_q$. Because

[DISTRIBUTION STATEMENT A] Approved for public release and unlimited distribution.

$\mathsf{dom}\ \mathsf{Out}_q^{\square}$ does not contain any pairs where the type is ϵ_{Type} (for {`noerror`}), it is irrelevant that the results of $[\![\cdot]\!]^{\square}_{\mathsf{Prop}}$ might contain such elements.

The port for each propagation–type pair $(f,t) \in \mathsf{dom}\ \mathsf{Out}_q^{\square}$ is influenced by all the propagations that have q as the source state and that have (f,t') as a propagation target, where $t' \sqsubseteq t$—the set *influencers* below. The condition of each such propagation is converted to assertions about gates and edges in the fault tree: the root of that tree is connected to the OR gate for the port identified by $\mathsf{OrGate}_q^{\square}(f,t)$. Thus, the gates and edges for propagation ports are given by the statements:

$$\begin{array}{l}\forall (f,t) \in \mathsf{dom}\ \mathsf{Out}_q^{\square}. \\ \quad \forall o \in \mathit{influencers}. \\ \qquad \mathrm{INPUT}(\mathsf{OrGate}_q^{\square}(f,t),\ [\![\lfloor \mathbf{OPC}^k_{\mathsf{Cond}}(o) \rfloor^k_{\mathsf{Cond}}]\!]^{(\square,q)}_{\mathsf{FT\_Cond}})\end{array}$$

where

$$\mathit{influencers} = \left\{ o \in \mathbf{D}^k_{\mathsf{OPC}} \mid [\![\mathbf{OPC}^k_{\mathsf{Src}}(o)]\!]^{(\square,q)}_{\mathsf{FT\_Src}}(q) \ \wedge\ (f,t') \in [\![\mathbf{OPC}^k_{\mathsf{Tgt}}(o)]\!]^{\square}_{\mathsf{Prop}} \ \wedge\ t' \sqsubseteq t \right\}$$

See Figure 11.5(c).

As discussed in Section 9.2, outgoing propagations are considered based on the *new state* after transitions are considered. That is, when state q *becomes* the current state, each propagation whose source state is q is considered. Because the propagations in the fault tree depend on the *activation* of the state (via $\mathsf{PAND}(\mathsf{Activate}_{\square}(q), \ldots)$), this is precisely the semantics that are implemented in the fault tree.

11.5 Edges from Propagation Paths

After the initial construction of component faults trees for the components in $\mathbf{K}$ as described above, edges between the propagation ports of an instance CFT and its nested instance state CFTs exist:

- edges from instance state CFT propagation output ports to instance CFT propagation output ports
- edges from instance CFT propagation input ports to instance state CFT propagation input ports

See Figure 11.2. What is missing are

- edges between the propagation ports of instance state CFTs and their *subcomponent* instance CFTs
- edges between *sibling* instance CFTs

These edges are added by processing the *propagation paths* of the EMV2 model. Propagation paths derive from

- the semantic connections[2] between features in the AADL instance model
- the declared propagation paths in the EMV2 annex clauses

Explicitly declared propagation paths allow the modeler to express paths to/from user-declared propagation points that cannot be expressed by connections in the AADL core language. The syntactic domain $\mathbf{D}^k_{\mathsf{Path}} \subseteq \mathbf{O}_{\mathsf{Path}} \subset \mathbf{O}$ is the set of propagation path objects in component k.[3] A path has two attributes: a list of source propagations and a list of destination propagations. These are obtained using the functions $\mathbf{Path}_{\mathsf{Src}} : \mathbf{O}_{\mathsf{Path}} \to \mathsf{list}(\mathbf{O}_{\mathsf{PP}})$ and

[2] Cf. `ConnectionInstance` in the AADL meta model.
[3] Cf. Class `PropagationPathInstance` in the EMV2 instance meta model.

[DISTRIBUTION STATEMENT A] Approved for public release and unlimited distribution.

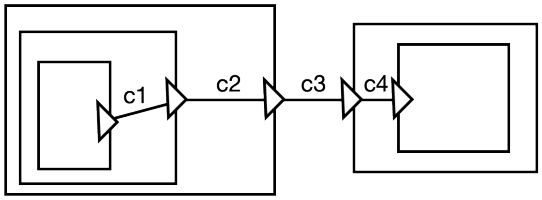

Figure 11.7: A Propagation Path Through and Across Nested Components

$\mathbf{Path}_{\mathsf{Dst}} : \mathbf{O}_{\mathsf{Path}} \rightarrow \mathsf{list}(\mathbf{O}_{\mathsf{PP}})$, respectively. The source and destination lists have the following meaning (see Figure 11.7):

- The source list describes the propagation "upwards" via out propagations from a leaf component to an out propagation of an ancestor component; see path segments `c1` and `c2` in Figure 11.7. The propagations must be out propagations.
- Similarly, the destination list describes the propagation "downwards" via in propagations from a component to an in propagation of a descendant leaf component; see path segment `c4` in Figure 11.7. The propagations must be in propagations.
- If the source list is empty, the propagation begins at an in propagation of the root component $\mathbf{k}_{\mathsf{Top}}$.
- If the destination list is empty, the propagation ends at an out propagation of the root component $\mathbf{k}_{\mathsf{Top}}$.
- The source and destination lists cannot both be empty.
- When the source and destination lists are both non-empty, then there is an implied connection between the final source propagation and the first destination propagation; see path segment `c3` in Figure 11.7.

Considering that a propagation path exists across multiple components, it is fair to wonder which propagation paths are in $\mathbf{D}^k_{\mathsf{Path}}$ for any given component k. A path belongs to the component k lowest in the tree that is sufficient to contain the entire path:

- When the source or destination list is empty, this is $\mathbf{k}_{\mathsf{Top}}$.
- Otherwise, it is the component k such that $\mathbf{PP}_{\mathsf{K}}(\mathsf{last}(s)) \in \mathbf{D}^k_{\mathsf{Sub}} \wedge \mathbf{PP}_{\mathsf{K}}(\mathsf{first}(d)) \in \mathbf{D}^k_{\mathsf{Sub}}$.

The set of all propagation paths in the entire model is $\mathbf{D}_{\mathsf{Path}} = \bigcup_{k \in \mathbf{K}} \mathbf{D}^k_{\mathsf{Path}}$. Finally, note that the propagation path is described solely in terms of the propagation points and *not* the type being propagated. The edges introduced into the CFTs, therefore, must take type containment into account.

Each propagation path $w \in \mathbf{D}_{\mathsf{Path}}$ implies edges across the CFTs. As suggested by the above interpretation of the lists of source and destination propagations, edges are implied

- by sequential pairs of propagations in the list of source propagations
- by sequential pairs of propagations in the list of destination propagations
- between the last source propagation and first destination propagation

These are described in the following sections.

11.5.1 Edges from the Source List

Each sublist $[s, d] \in \mathbf{Path}_{\mathsf{Src}}(w)$ implies a set of edges from an instance CFT's output ports to the output ports of all the instance state CFTs that contain it. The set of propagation–type

[DISTRIBUTION STATEMENT A] Approved for public release and unlimited distribution.

pairs identifying the destination output ports *is the same* for each instance state CFT because the system configuration is identical in each state. Specifically,

- the ports of $CFT_{\square}$, where semantic component $\square = [\![\mathbf{PP}_{\mathsf{K}}(s)]\!]_{\mathsf{K}}$, are the sources of the edges
- the set $srcs = \{(f,t) \in \mathsf{dom}\ \mathsf{Out}_{\square} \mid f = s\}$ identifies the propagation–type pairs that describe the source ports of $CFT_{\square}$
- the ports of instance state CFTs of component $\square' = [\![\mathbf{PP}_{\mathsf{K}}(d)]\!]_{\mathsf{K}}$ are the nominal destinations of the edges (In actuality, the destinations of the edges are the input ports of the OR gate connected to the port as abstracted by $\mathsf{OrGate}_q^{\square'}$ for each state $q \in \mathcal{Q}_{\square'}$.)
- the set $dsts_t = \{(f',t') \in \mathsf{dom}\ \mathsf{Out}_{\square'} \mid f' = d \wedge t \sqsubseteq t'\}$ identifies the propagation–type pairs that describe the ports of $CFT^q_{\square'}$ that are the destinations for the source port described by $(f,t) \in srcs$

Therefore, the following statements regarding edges are implied:

$$
\begin{array}{l}
\forall\ w \in \mathbf{D}_{\mathsf{Path}}. \\
\quad \forall\ [s,d] \in \mathbf{Path}_{\mathsf{Src}}(w). \\
\quad\quad \forall\ (f,t) \in srcs. \\
\quad\quad\quad \forall\ q \in \mathcal{Q}_{\square'}. \\
\quad\quad\quad\quad \forall\ (f',t') \in dsts_t. \\
\quad\quad\quad\quad\quad \mathrm{INPUT}(\mathsf{OrGate}_q^{\square'}(f',t'),\ \mathsf{Out}_{\square}(f,t))
\end{array}
$$

11.5.2 Edges from the Destination List

Each sublist $[s,d] \in \mathbf{Path}_{\mathsf{Dst}}(w)$ implies a set of edges from the in propagation ports of all the instance state CFTs in a component to the in propagation ports of a specific subcomponent instance CFT. The set of source input ports *is the same* for each instance state CFT. Specifically,

- the ports of the instance state CFTs of component $\square = [\![\mathbf{PP}_{\mathsf{K}}(s)]\!]_{\mathsf{K}}$ are the sources of the edges
- the set $srcs = \{(f,t) \in \mathsf{dom}\ \mathsf{In}_{\square} \mid f = s\}$ identifies the propagation–type pairs that describe the source ports. The actual source ports are given by $\mathsf{In}_q^{\square}$ for each $q \in \mathcal{Q}_{\square}$.
- the ports of $CFT_{\square'}$, where semantic component $\square' = [\![\mathbf{PP}_{\mathsf{K}}(d)]\!]_{\mathsf{K}}$, are the nominal destinations of the edges (In actuality, the destinations of the edges are the input ports of the OR gate connected to the port as abstracted by $\mathsf{OrGate}_{\square'}$.)
- the set $dsts_t = \{(f',t') \in \mathsf{dom}\ \mathsf{In}_{\square'} \mid f' = d \wedge t \sqsubseteq t'\}$ identifies the propagation–type pairs that describe the ports of $CFT_{\square'}$ that are the destinations for the source port described by $(f,t) \in srcs$

Therefore, the following statements regarding edges are implied:

$$
\begin{array}{l}
\forall\ w \in \mathbf{D}_{\mathsf{Path}}. \\
\quad \forall\ [s,d] \in \mathbf{Path}_{\mathsf{Dst}}(w). \\
\quad\quad \forall\ (f,t) \in srcs. \\
\quad\quad\quad \forall\ q \in \mathcal{Q}_{\square}. \\
\quad\quad\quad\quad \forall\ (f',t') \in dsts_t. \\
\quad\quad\quad\quad\quad \mathrm{INPUT}(\mathsf{OrGate}_{\square'}(f',t'),\ \mathsf{In}_q^{\square}(f,t))
\end{array}
$$

[DISTRIBUTION STATEMENT A] Approved for public release and unlimited distribution.

11.5.3 Edges Between Siblings

When both lists of edges are non-empty, there are edges between the instance CFTs of two sibling components. This is a simpler operation because no states are involved, and there is only one source–destination pair:

- The set of all paths with non-empty source and destination lists is $across = \{w \in \mathbf{D}_{\mathsf{Path}} \mid |\mathbf{Path}_{\mathsf{Src}}(w)| \neq 0 \wedge |\mathbf{Path}_{\mathsf{Dst}}(w)| \neq 0\}$.
- The source propagation is $s = \mathsf{last}(\mathbf{Path}_{\mathsf{Src}}(w))$.
- The destination propagation is $d = \mathsf{first}(\mathbf{Path}_{\mathsf{Dst}}(w))$.
- The ports of $CFT_{\square}$, where semantic component $\square = [\![\mathbf{PP}_{\mathsf{K}}(s)]\!]_{\mathsf{K}}$, are the sources of the edges.
- The set $srcs = \{(f,t) \in \mathsf{dom}\ \mathsf{Out}_{\square} \mid f = s\}$ identifies the propagation–type pairs that describe the source ports of $CFT_{\square}$.
- The ports of $CFT_{\square'}$, where semantic component $\square' = [\![\mathbf{PP}_{\mathsf{K}}(d)]\!]_{\mathsf{K}}$, are the nominal destinations of the edges. In actuality, the destinations of the edges are the input ports of the OR gate connected to the port as abstracted by $\mathsf{OrGate}_{\square'}$.
- The set $dsts_t = \{(f',t') \in \mathsf{dom}\ \mathsf{In}_{\square'} \mid f' = d \wedge t \sqsubseteq t'\}$ identifies the propagation–type pairs that describe the ports of $CFT_{\square'}$ that are the destinations for the source port described by $(f,t) \in srcs$.

Therefore, the following statements regarding edges are implied:

$$
\begin{array}{l}
\forall\ w \in across. \\
\quad \forall\ (f,t) \in srcs. \\
\quad\quad \forall\ (f',t') \in dsts_t. \\
\quad\quad\quad \mathrm{INPUT}(\mathsf{OrGate}_{\square'}(f',t'),\ \mathsf{In}_{\square}(f,t))
\end{array}
$$

[DISTRIBUTION STATEMENT A] Approved for public release and unlimited distribution.

12 Generating and Optimizing the Fault Tree

Assuming a syntactic AADL instance model $(\mathbf{K}, \mathbf{sub}, \mathbf{k}_{\mathsf{Top}})$ with corresponding semantic model $(\mathcal{K}, \gg, \square_{\mathsf{Top}})$, the previous section describes how to obtain set $CFT_{\mathcal{K}} = \{CFT_{\square} \mid \square \in \mathcal{K}\}$. This section describes how to obtain a fault tree—really a cause–effect graph—from the resulting hierarchy of CFTs:

- The definitions of *fault tree* and *fault tree query* from Section 10.1 are revisited and formalized.
- The process for extracting a fault tree from the CFTs is formalized.
- The simplifying transformations applied to the extracted fault tree are described.

When X is a family of functions parameterized by a semantic component, such as $N_{\mathsf{In}}(\cdot)$, then the notation $X_{\mathcal{K}} = \bigcup_{\square \in \mathcal{K}} \left(X^{\square} \cup \bigcup_{q \in \mathcal{Q}_{\square}} X_q^{\square} \right)$. That is, it unifies the sets from the instance and instance state CFTs.

12.1 Fault Tree Queries

Section 10.1 defines the problem of generating a fault tree from an AADL model thusly:

> Given
>
> - an AADL instance model
> - a particular *system operation mode* of the system, that is, a one-to-one mapping of component instances to one of their operational modes
> - a particular *initial behavior state* of the *entire system* relative to the system operational mode, that is, a one-to-one mapping of component instances to EMV2 error behavior states (such a thing is not currently described in the EMV2 specification)
> - a particular *top event*—that is, *state transition*, *state activation*, or *propagation* within the system—not necessarily from the top-level component
>
> the problem is how to generate a dynamic fault tree that enables answering the following two questions:
>
> 1. What are the shortest event sequences that lead to the occurrence of the top event? Colloquially, what is the fastest way to crash the system?
> 2. What is the probability of the top event occurring? Colloquially, what is the probability of crashing the system?
>
> Collectively these inputs are known as the *fault tree query*.

The components of a fault tree query are now formalized:

- The first two aspects of the query are already formalized as the semantic AADL model: recall *it is assumed that the syntactic AADL model is already projected into a specific system operation mode*; see Section 4.
- A *system behavior state* is a function β whose domain is $\mathcal{K}$.[1] The function is *total* (i.e., has a value for each domain element), and $\beta(\square) \in \mathcal{Q}_{\square}$ for each $\square \in \mathcal{K}$. In plain English,

[1] β for behavior.

[DISTRIBUTION STATEMENT A] Approved for public release and unlimited distribution.

a system behavior state maps each component to a specific error behavior state.

- A *top event* ρ is any output port of an AADL instance CFT or AADL instance state CFT: $\rho \in N^{\square}_{\mathsf{Out}}$ or $\rho \in N^{\square}_{q\ \mathsf{Out}}$ for $\square \in \mathcal{K}$ and $q \in \mathcal{Q}_{\square}$.[2]

Now, a fault tree query for a semantic AADL model $(\mathcal{K}, \gg, \square_{\mathsf{Top}})$ with component fault trees $CFT_{\mathcal{K}}$ is a pair (β, ρ). Based on the definitions of $N^{\square}_{\mathsf{Out}}$ and $N^{\square}_{q\ \mathsf{Out}}$, a rich set of queries is enabled:

- $\rho = \mathsf{Current}_{\square}(q)$ — How can, or with what probability does, component $\square$ reach state q? (This is the same as $\rho \in \mathsf{Activate}_{\square}(q)$.)
- $\rho = \mathsf{Out}_{\square}(f, t)$ — How can, or with what probability does, component $\square$ propagate error type t over propagation point f?
- $\rho = \mathsf{Out}^{\square}_{q}(f, t)$ — How can, or with what probability does, component $\square$ propagate error type t over propagation point f when in state q?
- $\rho = \mathsf{Trans}^{\square}_{q}(q')$ — How can, or with what probability does, component $\square$ transition from state q to state q'?

12.2 Fault Tree Generation

The general description of fault tree generation from CFTs from Section 10.3.4 is elaborated here. First, the manner of inserting the system behavior state into the CFTs is described. Then, the cause–effect graph is formally defined. Finally, the fault tree for a fault tree query (β, ρ) is defined.

12.2.1 Setting the Initial System State

The instance CFTs leave the initial state ports described by $\mathsf{Init}_{\square}$ without inputs. (All ports in $\mathsf{ran}\ \mathsf{In}^{\square}$ do have inputs because they are primed with an OR gate that has a FALSE input.) This must be remedied before a fault tree is generated from them. The initial system state β from the fault tree query is used to produce the initial state component fault trees $CFT'_{\mathcal{K}}$. For each $\square \in \mathcal{K}$ where $CFT_{\square} = (N_{\square}, G_{\square}, O_{\square}, S_{\square}, E_{\square})$, let $CFT^{\square}_{\beta} = (N_{\square}, G'_{\square}, O'_{\square}, S_{\square}, E'_{\square})$:

- $G'_{\square} = G_{\square} \cup \{t, f\}$, where t, f are fresh symbols
- $O'_{\square} = O_{\square} \cup \{t \mapsto \mathsf{TRUE}, f \mapsto \mathsf{FALSE}\}$
- $E'_{\square} = E_{\square} \cup \{(t.\mathsf{out}, \mathsf{Init}_{\square}(q_0))\} \cup \{(f.\mathsf{out}, \mathsf{Init}_{\square}(q)) \mid q \in \mathcal{Q}_{\square} \setminus \{q_0\}\}$, where $\beta(\square) = q_0$

Let $CFT^{\mathcal{K}}_{\beta} = \{CFT^{\square}_{\beta} \mid \square \in \mathcal{K}\}$ be the full set of initial state CFTs for the AADL model.

12.2.2 Cause–Effect Graph

While Kaiser and colleagues do not formally describe the cause–effect graph (CEG) extracted from a hierarchy of CFTs [15], here it is considered a special form of a component fault tree in which nodes representing ports and subcomponents have been eliminated. A CEG (V, G, O, E, r) for a semantic AADL model $(\mathcal{K}, \gg, \square_{\mathsf{Top}})$ with component fault trees $CFT_{\mathcal{K}}$ is a rooted directed graph with nodes $V \cup G$, edges E, and root node $r \in V \cup G_{\mathsf{Out}}$:

- $V \subseteq N^{\mathcal{K}}_{\mathsf{Intern}}$, where $N^{\mathcal{K}}_{\mathsf{Intern}} = \mathcal{V}_{\mathcal{K}}$, is the set of basic event nodes in the graph.[3]
- $G \subseteq G_{\mathcal{K}}$ is the set of gate nodes in the graph.
- $O \subseteq O_{\mathcal{K}}$ is the total gate node–operator mapping.

[2] ρ for root.
[3] Again, V for eVent.

- $E \subseteq (V \cup G_{\mathsf{Out}}) \times G_{\mathsf{In}}$ is the set of edges in the graph: for $(s, d) \in E$, s is the source of the edge and d is the destination of the edge. Here the edges are not necessarily a subset of the edges of those in the CFTs due to port nodes being elided.

An important property of a CEG is that every node in $V \cup G$ is reachable from the root r. This is formalized using the definition of reachable from Section 2.4.1. A modified version of the edges set must be used because reachable expects edges to point away from the root, and the input and output ports of the gate nodes must be elided:

1. Let $E' \in G \times (V \cup G)$ be set of edges: $E' = \{(g', g) \mid (g.\mathsf{out}, g'.\mathsf{in}_i) \in E\} \cup \{(g', v) \mid (v, g'.\mathsf{in}_i) \in E\}$.
2. $\forall n \in (V \cup G).\ \mathsf{reachable}(r, n)$ when the set of edges is E'.

12.2.3 Generating the Fault Tree

Given a set of component fault trees $CFT_{\mathcal{K}}$, the fact that subcomponents (the sets $S_{\square}$) are directly replaced with their associated CFT means that $CFT_{\mathcal{K}}$ essentially implies a giant graph structure with nodes $N_{\mathcal{K}} \cup G^{\mathcal{K}}_{\mathsf{Out}} \cup G^{\mathcal{K}}_{\mathsf{In}}$ and edges $E_{\mathcal{K}}$. Recall that subcomponent replacement means that the edges to/from subcomponents are really edges to/from nodes in $N^{\mathcal{K}}_{\mathsf{In}}$ and $N^{\mathcal{K}}_{\mathsf{Out}}$, respectively. A CEG can be constructed via a depth-first traversal over this graph.

12.2.3.1 Depth-First Traversal

Given a set of component fault trees $CFT_{\mathcal{K}}$ and a node $n \in N_{\mathcal{K}} \cup G_{\mathcal{K}}$, then $DFT_{\mathcal{K}}(n)$ produces the graph rooted at n inductively defined by the process below.[4] Based on the definition of CEG above, the purpose is to collect reachable nodes and to elide port nodes.[5] Induction is over the lengths of the paths implied by the edges in $E_{\mathcal{K}}$. The base cases (for a path length of 0) are when n is an event node or inputless gate:

- When n is an event node, $n \in N^{\mathcal{K}}_{\mathsf{Intern}}$, $DFT_{\mathcal{K}}(n) = (\{n\}, \emptyset, \emptyset, \emptyset, n)$.
- When g is a Boolean literal, $g.\mathsf{out} \in G^{\mathcal{K}}_{\mathsf{Out}} \wedge O_{\mathcal{K}}(g) = o \wedge o \in \{\mathsf{TRUE}, \mathsf{FALSE}\}$, $DFT_{\mathcal{K}}(g.\mathsf{out}) = (\emptyset, \{g\}, \{g \mapsto o\}, \emptyset, g.\mathsf{out})$.

The remaining inductive cases are when $n \in N^{\mathcal{K}}_{\mathsf{In}}$, $n \in N^{\mathcal{K}}_{\mathsf{Out}}$, and $g.\mathsf{out} \in G^{\mathcal{K}}_{\mathsf{Out}} \wedge O_{\mathcal{K}}(g) \in \{\mathsf{OR}, \mathsf{AND}, \mathsf{PAND}\}$.

Port nodes, either input or output, are elided from the final CEG:

- When n is an input port, $n \in N^{\mathcal{K}}_{\mathsf{In}}$, $DFT_{\mathcal{K}}(n) = DFT_{\mathcal{K}}(n')$ where $(n', n) \in E_{\mathcal{K}}$. (Remember, nodes may only have a single incoming edge.)
- When n is an output port, $n \in N^{\mathcal{K}}_{\mathsf{Out}}$, $DFT_{\mathcal{K}}(n) = DFT_{\mathcal{K}}(n')$ where $(n', n) \in E_{\mathcal{K}}$. (Remember, nodes may only have a single incoming edge.)

Gate output ports must be traced through their implicit edges to the gate's input ports. The CEGs defined by the subtrees below each input port must be merged with the gate itself to form a new CEG description:

- When $g.\mathsf{out} \in G^{\mathcal{K}}_{\mathsf{Out}} \wedge O_{\mathcal{K}}(g) \in \{\mathsf{OR}, \mathsf{AND}, \mathsf{PAND}\}$, $DFT_{\mathcal{K}}(g.\mathsf{out}) = (V', G', O', E', g.\mathsf{out})$, where
  - $(V_i, G_i, O_i, E_i, r_i) = DFT_{\mathcal{K}}(n_i)$ for $(n_i, g.\mathsf{in}_i) \in E_{\mathcal{K}}$.
  - $V' = \bigcup_i V_i$.

[4] It is convenient that DFT could stand for either "depth first search" or "dynamic fault tree."

[5] Keep in mind that this is a *definition* of the desired CEG. It is not presented as the best algorithm for computing it. In particular, the fact that nodes might be visited multiple times by the recursive/inductive definition is not a problem because of set union.

[DISTRIBUTION STATEMENT A] Approved for public release and unlimited distribution.

- $G' = \{g\} \cup \bigcup_i G_i$.
- $O' = \{g \mapsto O_{\mathcal{K}}(g)\} \cup \bigcup_i O_i$.
- $E' = \bigcup_i \{(r_i, g.\mathsf{in}_i)\} \cup \bigcup_i E_i$.

12.2.3.2 Answering the Query (β, ρ)

Finally, the full process of producing the fault tree—really the CEG—$FT^{\mathcal{K}}_{(\beta,\rho)}$ in response to the query (β, ρ) about the AADL model $(\mathcal{K}, \gg, \square_{\mathsf{Top}})$ with component fault trees $CFT_{\mathcal{K}}$ is as follows:

1. Create the initial state CFTs $CFT^{\mathcal{K}}_{\beta}$.
2. Create $FT^{\mathcal{K}}_{(\beta,\rho)} = DFT^{\mathcal{K}}_{\beta}(\rho)$.

12.3 Fault Tree Simplification

The fault tree $FT^{\mathcal{K}}_{(\beta,\rho)}$ benefits from simplification. Generally, such a step is tantamount to performing analysis of the fault tree and is not historically of interest because fault trees were developed by hand. In this case, however, the fault is developed automatically from an existing model and contains, therefore, what might be termed "extraneous structure," "required paths," and "dead paths":

- Extraneous OR gates exist due to OR gates with FALSE inputs being attached to *all* input ports representing in propagations and state activations during construction of the CFTs; see Sections 11.4.1 and 11.4.2. This is done to make it possible to connect multiple incoming edges to the port. These OR gates are extraneous when the port has zero or one actual inputs.
- Required paths and dead paths originate from the TRUE and FALSE gates, respectively, that are added when setting the initial system state. These mark events that must always or never occur, respectively.

From a human-understanding point of view, it is definitely beneficial to remove these extra gates and paths, as it improves the readability of the fault tree. The main motivation for simplification, however, is that fault trees in the literature are not presented as featuring gates that represent the Boolean constants. In fact, doing so may be a novelty of this work. In particular, they do not feature in any of the common fault tree analysis techniques. Although true ($\top$) and false ($\bot$) do appear in Merle's algebra, they are there for completeness of the logical reasoning and as intermediate values and are not presented as being part of the original fault tree.

With this in mind, the simplification of the fault tree is limited in scope to removing the TRUE and FALSE gates. For each gate with inputs (i.e., PAND, AND, and OR), there are transformations that handle the output of a TRUE or FALSE gate being used as an input. The goal is either to remove the input or to replace the gate (and its inputs) with a TRUE or FALSE gate and then to push the problem up to the next level in the fault tree. Ultimately, either all uses of TRUE and FALSE gates are removed, or a single TRUE or FALSE gate remains as the only item in the fault tree. Regarding the problem of interpreting TRUE and FALSE gates in traditional fault tree analyses, in this case, the gates can be interpreted thusly:

- A TRUE gate is an event that occurs with 100% probability.
- A FALSE gate is an event that occurs with 0% probability.

The specific transformations/simplifications are described below assuming a CEG (V, G, O, E, r). Figure 12.1 lists the referenced theorems from Merle.

$$
\begin{array}{lll}
(a) & \begin{array}{ll} (3.1) & a + b = b + a \\ (3.2) & a \cdot b = b \cdot a \\ (3.8) & a + \bot = a \\ (3.9) & a \cdot \top = a \end{array} & \begin{array}{ll} (3.10) & a \cdot \bot = \bot \\ (3.12) & a + \top = \top \\ (3.49) & \bot \trianglelefteq a = \bot \\ (3.50) & a \trianglelefteq \bot = a \end{array} \\
(b) & \multicolumn{2}{c}{\mathsf{PAND}(A, B) = B \cdot (A \trianglelefteq B)}
\end{array}
$$

Figure 12.1: (a) Cited Theorems from Merle's Algebra [19, §3]; (b) Definition of PAND *[19, §4.1]*

12.3.1 Transforming OR Gates

Consider an OR gate with m inputs:

- $g \in G$
- $O(g) = \mathsf{OR}$
- $\{(n_1, g.\mathsf{in}_1), \ldots, (n_m, g.\mathsf{in}_m)\} \subseteq E$

The following transformations apply:

- If *any* of the inputs are TRUE—$\exists 1 \leq i \leq m.\, (n_i = g'.\mathsf{out} \wedge O(g') = \mathsf{TRUE})$—then the gate can be replaced by the TRUE gate n_i. This is justified by the repeated application of theorems (3.1) and (3.12).
- When any of the inputs are FALSE, they can be removed. This is justified by repeated application of theorems (3.1) and (3.8). There are two cases here:
  - When at most one of the inputs is not FALSE—$\exists 1 \leq j \leq m. \forall 1 \leq i \leq m. (i \neq j) \Rightarrow (n_i = g_i'.\mathsf{out} \wedge O(g_i') = \mathsf{FALSE})$—then the gate can be replaced with n_j. An interesting case here is when *all* the inputs are FALSE, in which case any one of the inputs may be chosen as the replacement with the effect of "evaluating" the disjunction to FALSE.
  - When at least two inputs are not FALSE—$X = \{1 \leq i \leq m \mid \nexists g' \in G.\, (n_i = g'.\mathsf{out} \wedge O(g') = \mathsf{FALSE})\} \wedge |X| \geq 2$—the FALSE inputs can be dropped.

12.3.2 Transforming AND Gates

Consider an AND gate with m inputs:

- $g \in G$
- $O(g) = \mathsf{AND}$
- $\{(n_1, g.\mathsf{in}_1), \ldots, (n_m, g.\mathsf{in}_m)\} \subseteq E$

The following transformations apply:

- If *any* of the inputs are FALSE—$\exists 1 \leq i \leq m.\, (n_i = g'.\mathsf{out} \wedge O(g') = \mathsf{FALSE})$—then the gate can be replaced by the FALSE gate n_i. This is justified by the repeated application of theorems (3.2) and (3.10).
- When any of the inputs are TRUE, they can be removed. This is justified by repeated application of theorems (3.2) and (3.9). There are two cases here:
  - When at most one of the inputs is not TRUE—$\exists 1 \leq j \leq m. \forall 1 \leq i \leq m. (i \neq j) \Rightarrow (n_i = g_i'.\mathsf{out} \wedge O(g_i') = \mathsf{TRUE})$—then the gate can be replaced with n_j. An interesting case here is when *all* the inputs are TRUE, in which case, any one

[DISTRIBUTION STATEMENT A] Approved for public release and unlimited distribution.

of the inputs may be chosen as the replacement with the effect of "evaluating" the conjunction to TRUE.

- When at least two inputs are not TRUE—$X = \{1 \leq i \leq m \mid \nexists g' \in G.(n_i = g'.\mathsf{out} \wedge O(g') = \mathsf{TRUE})\} \wedge |X| \geq 2$—the TRUE inputs can be dropped.

12.3.3 Transforming PAND Gates

Simplification of PAND gates relies on theorems whose proofs are given in Section 12.5. Consider a PAND gate:

- $g \in G$
- $O(g) = \mathsf{PAND}$
- $\{(a, g.\mathsf{in}_1), (b, g.\mathsf{in}_2)\} \subseteq E$

Transformation is trickier here because the first and second inputs have different meanings. The four cases need to be considered explicitly:

- When a is FALSE—$a = g'.\mathsf{out} \wedge O(g') = \mathsf{FALSE}$—the gate can be replaced by the FALSE gate.
- When a is TRUE—$a = g'.\mathsf{out} \wedge O(g') = \mathsf{TRUE}$—the gate can be replaced by b.
- When b is FALSE—$b = g'.\mathsf{out} \wedge O(g') = \mathsf{FALSE}$—the gate can be replaced by the FALSE gate.
- When b is TRUE and a is not TRUE—$(b = g_b.\mathsf{out} \wedge O(g_b) = \mathsf{TRUE}) \wedge (\nexists g_a \in G.(a = g_a.\mathsf{out} \wedge O(g_a) = \mathsf{TRUE}))$—the gate can be replaced by a fresh FALSE gate. The case where both a and b are TRUE is handled by the "a is TRUE" case above.

  It must be emphasized here that "not TRUE" *does not mean* FALSE. Recall the gates are defined in terms of Merle's algebra. So this condition specifically means that the event represented by a must become true at some date after 0 (i.e., $d(a) > 0$), which does allow for $d(a) = +\infty \Rightarrow a = \bot$. What this condition cannot tolerate is a being universally true from date 0: $d(a) = 0$ (i.e., a is the output of a TRUE gate).

12.3.4 Transformation/Simplification Defined

Given a CEG (V, G, O, E, r) and a node $n \in V \cup G_{\mathsf{Out}}$, then $SIMPLIFY(n)$ produces the simplified version of the CEG rooted at n. So $SIMPLIFY(r)$ is the simplified version of the entire CEG. *SIMPLIFY* is defined by induction over the lengths of the paths implied by the edges in E. The base cases (for a path length of 0) are when n is an event node or the output of an inputless gate:

- When n is an event node, $n = v \in V$—$SIMPLIFY(v) = (\{v\}, \emptyset, \emptyset, \emptyset, v)$.
- When g is a Boolean literal—$g.\mathsf{out} \in G_{\mathsf{Out}} \wedge O(g) = o \wedge o \in \{\mathsf{TRUE}, \mathsf{FALSE}\}$—$SIMPLIFY(g.\mathsf{out}) = (\emptyset, \{g\}, \{g \mapsto o\}, \emptyset, g.\mathsf{out})$.

For the inductive cases, the node must an m-input gate, where $m > 1$:

- $n = g.\mathsf{out} \in G_{\mathsf{Out}}$
- $o = O(g) \in \{\mathsf{AND}, \mathsf{OR}, \mathsf{PAND}\}$
- $\{(n_1, g.\mathsf{in}_1), \ldots, (n_m, g.\mathsf{in}_m)\} \in E$
- $(V_i, G_i, O_i, E_i, r_i) = SIMPLIFY(n_i)$

When none of the *simplified* inputs r_i are TRUE or FALSE, the gate is left as is but the subtrees are simplified:

- When $\nexists 1 \leq i \leq m.\,(r_i = g'.\mathsf{out} \wedge O(g') \in \{\mathsf{TRUE}, \mathsf{FALSE}\})$,
  $SIMPLIFY(g.\mathsf{out}) = (\bigcup_i V_i,\, \{g\} \cup \bigcup_i G_i,\, \{g \mapsto o\} \cup \bigcup_i O_i,\, \bigcup_i \{(r_i, g.\mathsf{in}_i)\} \cup \bigcup_i E_i,\, g.\mathsf{out})$.

There are three cases for simplifying an OR gate (i.e., $o = \mathsf{OR}$):

- When $\exists 1 \leq i \leq m.\,(r_i = g'.\mathsf{out} \wedge O(g') = \mathsf{TRUE})$,
  $SIMPLIFY(g.\mathsf{out}) = (\emptyset,\, \{g'\}, \{g' \mapsto \mathsf{TRUE}\}, \emptyset, g'.\mathsf{out})$.
- When $\exists 1 \leq j \leq m. \forall 1 \leq i \leq m.\,(i \neq j) \Rightarrow (r_i = g'_i.\mathsf{out} \wedge O(g'_i) = \mathsf{FALSE})$,
  $SIMPLIFY(g.\mathsf{out}) = (V_j, G_j, O_j, E_j, r_j)$.
- Let $X = \{1 \leq i \leq m \mid \nexists g' \in G.\,(r_i = g'.\mathsf{out} \wedge O(g') = \mathsf{FALSE})\} = \{i'_j\}$. When $|X| \geq 2$,
  $SIMPLIFY(g.\mathsf{out}) = (\bigcup_{i' \in X} V_{i'},\, \{g\} \cup \bigcup_{i' \in X} G_{i'},\, \{g \mapsto o\} \cup \bigcup_{i' \in X} O_{i'},\, \bigcup_j \{(r_{i'_j},\, g.\mathsf{in}_j)\} \cup \bigcup_{i' \in X} E_{i'},\, g.\mathsf{out})$.

  Note that X is a set of indices, and transformation reindexes the input ports to the gate, which is justified by repeated application of (3.1).

There are three similar cases for simplifying an AND gate (i.e., $o = \mathsf{AND}$):

- When $\exists 1 \leq i \leq m.\,(n_i = g'.\mathsf{out} \wedge O(g') = \mathsf{FALSE})$,
  $SIMPLIFY(g.\mathsf{out}) = (\emptyset,\, \{g'\}, \{g' \mapsto \mathsf{FALSE}\}, \emptyset, g'.\mathsf{out})$.
- When $\exists 1 \leq j \leq m. \forall 1 \leq i \leq m.\,(i \neq j) \Rightarrow (n_i = g'_i.\mathsf{out} \wedge O(g'_i) = \mathsf{TRUE})$,
  $SIMPLIFY(g.\mathsf{out}) = (V_j, G_j, O_j, E_j, r_j)$.
- Let $X = \{1 \leq i \leq m \mid \nexists g' \in G.\,(n_i = g'.\mathsf{out} \wedge O(g') = \mathsf{TRUE})\} = \{i'_j\}$. When $|X| \geq 2$,
  $SIMPLIFY(g.\mathsf{out}) = (\bigcup_{i' \in X} V_{i'},\, \{g\} \cup \bigcup_{i' \in X} G_{i'},\, \{g \mapsto o\} \cup \bigcup_{i' \in X} O_{i'},\, \bigcup_j \{(r_{i'_j},\, g.\mathsf{in}_j)\} \cup \bigcup_{i' \in X} E_{i'},\, g.\mathsf{out})$.

  Note that X is a set of indices, and transformation reindexes the input ports to the gate, which is justified by repeated application of (3.2).

Finally, there are four cases for simplifying a PAND gate (i.e., $o = \mathsf{PAND}$). In this case, $m = 2$, $a = r_1$, and $b = r_2$.

- When $a = g'.\mathsf{out} \wedge O(g') = \mathsf{FALSE}$, $SIMPLIFY(g.\mathsf{out}) = (\emptyset,\, \{g'\}, \{g' \mapsto \mathsf{FALSE}\}, \emptyset, g'.\mathsf{out})$.
- When $a = g'.\mathsf{out} \wedge O(g') = \mathsf{TRUE}$, $SIMPLIFY(g.\mathsf{out}) = (V_2, G_2, O_2, E_2, r_2)$.
- When $b = g'.\mathsf{out} \wedge O(g') = \mathsf{FALSE}$, $SIMPLIFY(g.\mathsf{out}) = (\emptyset,\, \{g'\}, \{g' \mapsto \mathsf{FALSE}\}, \emptyset, g'.\mathsf{out})$.
- When $(b = g_b.\mathsf{out} \wedge O(g_b) = \mathsf{TRUE}) \;\wedge\; (\nexists g_a \in G.(a = g_a.\mathsf{out} \wedge O(g_a) = \mathsf{TRUE}))$,
  $SIMPLIFY(g.\mathsf{out}) = (\emptyset,\, \{g'\}, \{g' \mapsto \mathsf{FALSE}\}, \emptyset, g'.\mathsf{out})$, where g' is a fresh symbol.

12.3.4.1 Simplifying the Query Answer

To be clear, given the fault tree $FT^{\mathcal{K}}_{(\beta,\rho)} = (V, G, O, E, r)$ that is the answer to the query (β, ρ) about the AADL model $(\mathcal{K}, \gg, \square_{\mathsf{Top}})$ with component fault trees $CFT_{\mathcal{K}}$, the simplified query response is $SIMPLIFY(r) = \widehat{FT}^{\mathcal{K}}_{(\beta,\rho)}$.

12.4 Fault Tree Analysis

This work is *not* about analyzing the fault tree $\widehat{FT}^{\mathcal{K}}_{(\beta,\rho)}$. EMV2 does, however, allow using AADL property associations to describe probability distributions for error event declarations, and even for error propagations and error types themselves, although the exact intended meanings of the latter two cases are unclear. Thus, one last group of syntactic functions may be described:

[DISTRIBUTION STATEMENT A] Approved for public release and unlimited distribution.

- $\mathbf{D}^k_{\%} : \mathbf{D}^k_{\mathsf{Event}} \to [0,1]$ is the absolute probability that the event occurs when a *fixed* probability distribution is used.
- $\mathbf{D}^k_{\lambda} : \mathbf{D}^k_{\mathsf{Event}} \to \mathbb{R}$ is the error occurrence rate when an *exponential* probability distribution is used.

These functions can be used to decorate the fault tree with any necessary error occurrence information. The exact style of distributions that should be used in the model depends on the tools and methods being used to analyze the fault tree. The Storm model checker [3], for example, supports fixed probability distributions only on subgraphs that are static; dynamic subgraphs must use exponential distributions.

Generally speaking, analysis requires the fault tree to be converted into yet another representation. These translations and their descriptions are beyond the scope of this work but include

- binary decision diagrams (BDDs)
- Markov chains
- Petri nets
- algebraic expressions

12.5 Proofs of PAND Theorems

This section provides proofs for the PAND-related transformations used in Section 12.3. See the section on Merle's algebra, Section 2.1.2, and Merle's work itself [19, §3]. The cited theorems from Merle are shown in Figure 12.1.

Transformation 1 $\mathsf{PAND}(\bot, B) \longrightarrow \bot$

1. $\mathsf{PAND}(\bot, B) = B \cdot (\bot \trianglelefteq B)$ — *Definition of* PAND
2. $B \cdot (\bot \trianglelefteq B) = B \cdot \bot$ — *(3.49)*
3. $B \cdot \bot = \bot$ — *(3.10)*

Q.E.D.

Transformation 2 $\mathsf{PAND}(\top, B) \longrightarrow B$

1. $\mathsf{PAND}(\top, B) = B \cdot (\top \trianglelefteq B)$ — *Definition of* PAND
2. $d(\top) = 0$ — *definition of* $\top$
3. $0 \leq d(B)$ — *definition of* $\leq$
4. $\top \trianglelefteq B = \top$ — *(2), (3), and definition of* $\trianglelefteq$
5. $B \cdot (\top \trianglelefteq B) = B \cdot \top$ — *(4)*
6. $B \cdot \top = B$ — *(3.9)*

Q.E.D.

Transformation 3 $\mathsf{PAND}(A, \bot) \longrightarrow \bot$

1. $\mathsf{PAND}(A, \bot) = \bot \cdot (A \trianglelefteq \bot)$ — *Definition of* PAND
2. $\bot \cdot (A \trianglelefteq \bot) = \bot \cdot A$ — *(3.50)*
3. $\bot \cdot A = A \cdot \bot$ — *(3.2)*
4. $A \cdot \bot = \bot$ — *(3.10)*

Q.E.D.

Transformation 4 $\mathsf{PAND}(A, \top) \longrightarrow \bot$ *when* $A \neq \top$

1. $\mathsf{PAND}(A, \top) = \top \cdot (A \trianglelefteq \top)$ — *Definition of* ***PAND***
2. $\top \cdot (A \trianglelefteq \top) = (A \trianglelefteq \top) \cdot \top$ — *(3.2)*
3. $(A \trianglelefteq \top) \cdot \top = A \trianglelefteq \top$ — *(3.9)*
4. *Case analysis of* $A \trianglelefteq \top$*, where* $d(a) = d(A)$ *and* $d(b) = d(\top) = 0$
   - *Case* $d(a) < d(b)$*:* $d(A) < 0$ *is not possible*
   - *Case* $d(a) = d(b)$*: Occurs when* $d(A) = 0$*, in which case* $A = \top$ *and* $d(A \trianglelefteq \top) = d(A) = 0 = d(\top)$
   - *Case* $d(a) > d(b)$*: Occurs when* $A \neq \top$*, in which case* $d(A \trianglelefteq \top) = +\infty \Leftrightarrow A \trianglelefteq \top = \bot$
5. *Thus* $\mathsf{PAND}(A, \top) = \begin{cases} \top & \text{when } A = \top \\ \bot & \text{otherwise} \end{cases}$

Q.E.D.

[DISTRIBUTION STATEMENT A] Approved for public release and unlimited distribution.

13 Conclusion

The work presented herein advances the state of the art in understanding the error behavior of AADL models that include EMV2 annex subclauses. Specifically, the preceding sections

- provide a denotational semantics for a *subset* of the features of EMV2
- formally describe how to convert *each component* in AADL and EMV2 models into a behavior automata
- formally describe how to convert AADL and EMV2 models into a hierarchy of CFTs
- provide a formal definition, relative to specific AADL and EMV2 models and their CFT hierarchy, of the question answered by a fault tree
- formally describe how to obtain a fault tree—really a cause–effect graph—from a hierarchy of CFTs

The denotational semantics covers the commonly used features of EMV2. The usefulness of the coverage is demonstrated by using the semantic framework to construct CFTs from components in AADL and EMV2 models. Conversely, the CFT analysis provides a case study of *how to apply* the semantics to a real-world model-based analysis. This work is the first to formally describe the complete process for translating an AADL model—or any model-based architecture description—to a fault tree. Having a formal semantics of the AADL model and of EMV2 facilitated this presentation. Comments throughout, however, indicate features of EMV2 not captured by the semantics and automata and limitations to the scenarios that can be captured by extracted fault trees. This presentation, therefore, concludes with a discussion of how future work might address these issues.

13.1 Component Semantics and Behavior

The denotational semantics omits some features of AADL and EMV2. Notably, AADL modes are ignored by requiring the system model to be projected into a particular system operation mode. This fixes the mode (i.e., configuration) for each component. The primary difficulty with modes is that *mode transitions* are complicated, requiring the coordination of multiple components and even the consideration of timing constraints. This requires a better understanding of how the behaviors of multiple components interact—a subject that is beyond the scope of the current work. As stated in Section 1.2, this work contributes the description of a behavior automata for a *single component* as a first step towards understanding the formal error behavior of *a collection* of components.

13.1.1 Omitted Features of EMV2

Omitted features of EMV2 include typed state machines, recover and repair events, type mappings and transformations, composite error behavior, and error detection. EMV2 error behaviors can be typed [27, E.8]:

> The error behavior state machine can be defined as a typed token state transition system similar to a Colored Petri net, resulting in a more compact representation. This is accomplished by associating error types and type sets with error behavior events, states and error propagations to specify acceptable types of tokens. The current state, when typed, is represented by a type instance.

This is not a commonly used or understood feature of EMV2, so it was not included in the current semantics. EMV2 actually supports three kinds of events [27, E.8.1]:

[DISTRIBUTION STATEMENT A] Approved for public release and unlimited distribution.

> The Error Model language distinguishes between three kinds of error behavior events: error events, recover events, and repair events. An error event instance represents fault activation within a component and will result in a state transition to an error state that represents the resulting failure mode and in an outgoing error propagation. Recover events represent recovery from a nonworking state to a working state. This is used to model recovery from transient errors. Repair events represent longer duration repair action, whose completion results in a transition back to a working state.

This distinction is ignored in the semantics because

- the distinction between recover and repair in the specification itself is unclear
- making a distinction between kinds of events does not seem to affect the rest of the semantics (i.e., no other features depend on a distinction being made)
- a key assumption in fault tree modeling is that *errors are unrecoverable* (i.e., cannot be undone), so explicitly supporting the notion of recovery is not currently necessary

Error type mappings and transformations, composite error behavior, and error detection are omitted because they relate to intercomponent behavioral issues, which, again, are beyond the scope of this work. In particular,

- a significant scenario in which type mappings apply is when a propagation may be affected by the outgoing propagations of the components (e.g., bus or virtual bus) that the connection "carrying" the propagation is bound to
- composite error behavior directly relates the error state of a component to the error states of its subcomponents. It is unclear, furthermore, how this is intended to interact with the component's declared component behavior state.
- error detection allows a component to declare under what conditions—condition expressions again—the component detects particular error occurrences. Furthermore, these detections result in the AADL component sending data over a core AADL `event port` or `event data port` (not to be confused with EMV2 behavior events). These appear to be meant to trigger component mode transitions, but the exact intended use of detections is unclear.

13.1.2 Towards a Collection of Automata

Section 1.2 introduces the idea of modeling the behavior of a system by a hierarchical collection of automata; see Figure 1.1. The behavior automaton of an individual component (e.g., C in the figure) interacts with the automata of

- the component's subcomponents (e.g., S)
- the component's sibling components (e.g., D)
- the component's containing component (e.g., B)

Furthermore, each event (e.g., E or F) is represented by an automaton. An occurrence of the event is represented by the event's automaton outputting a particular symbol. This corresponds to what the EMV2 standard refers to as an "occurrence instance of an event" [27, E.6.(1)] or "error event instance" [27, E.8.1.(1)]. The following observations about the collection of automata can be made:

- The behavior automaton of a single component (e.g., C), as described in this work, is only *part* of the ultimate behavior of C. The full behavior of the component is determined by its behavior automaton together with the full behavior of each subcomponent

[DISTRIBUTION STATEMENT A] Approved for public release and unlimited distribution.

and the automata of the component's events. Of course, this is hierarchical: the full behavior of each subcomponent depends on the subcomponent's behavior automaton, the full behaviors of the subcomponent's subcomponents, and the automata of the subcomponent's events.

Thus, the component's behavior automaton, event automata, and subcomponent automata must be composed into an automaton representing the component's complete behavior.

- Implementing an event as an automaton provides flexibility in the behavior of the event. The EMV2 standard says little about an event except that it can be associated with a probability distribution. But this does not address, for example, whether an event can have multiple occurrences. As an automaton, an event could occur multiple times by outputting its occurrence symbol more than once, or it might transfer to a terminal state after emitting the symbol exactly once.
- Each *output symbol* from a behavior automaton represents zero or more error type propagations; see Section 9.4. The output from subcomponent automata, event automata, and any sibling automata must be integrated and converted into an environment *input symbol* of the component's behavior automata. Similarly, the output of the behavior automaton must be converted to an output symbol of the component's full behavior automaton.

These desiderata suggest that DEVS [6] could be a good fit, or at least an inspiration, for aggregating the different automata. In particular, it supports hierarchical decomposition and aggregation of automata and provides a mechanism for translating between alphabets.

13.2 Fault Trees

Future work on fault tree generation falls into two categories:

1. how to express more of the EMV2 semantics in the fault tree
2. how to capture more details of the system architecture in the fault tree

13.2.1 Expressing More EMV2 Semantics

As discussed in Section 10.5, the full expressiveness of EMV2 is beyond what can be expressed in typical fault tree formulations. The implied logical *negation* required by some of the primitive operators and by silencing leads to non-coherent fault trees. The logical foundation for fault trees used by this work, Merle's algebra, does not incorporate negation. An alternative foundational logic by Shilling [31] does include negation and could provide the means to expand the translation of EMV2 condition expressions to fault trees. But it does not provide a behavior model of the dynamic gates as comprehensive as the one provided by Merle. Additionally, as already mentioned, it is unclear if any tool support exists for evaluating non-coherent dynamic fault trees.

13.2.2 Capturing More Architecture Details

AADL models contain detailed information about the architecture of a system and the relationships of the components in the system—indeed, this is a significant purpose of AADL. This information can be exploited to add additional structure to generated fault trees:

- Component relationships can be analyzed to introduce instances of the functional dependency gate FDEP.
- Basic events can be replicated to analyze scenarios in which EMV2 events may occur more than once.

[DISTRIBUTION STATEMENT A] Approved for public release and unlimited distribution.

- Component relationships can be analyzed to introduce SPARE gates.

13.2.2.1 Functional Dependency

From Dugan, Bavuso, and Boyd,

> A functional dependency gate has: trigger-input (either a basic event or the output of another a gate in the tree), a non-dependent output (reflecting the status of the trigger event, one or more dependent basic events. The dependent basic events are functionally dependent on the trigger event. When the trigger event occurs, the dependent basic events are forced to occur. [8]

There is controversy around the FDEP gate. Many have observed that it is similar to an OR gate (e.g., Vesely and colleagues [34]), and Merle is able to prove this using his algebra [19, §4.2]. Some argue that the FDEP gate is misleading and poorly defined, claim that it confuses the true natures of the failures associated with the trigger and dependent events, and, therefore, discourage its use (e.g., Xiang and colleagues [35]). It is the opinion of the authors herein that

- it is best to express functional dependency using an explicit FDEP gate and not a logically equivalent OR gate. An FDEP expresses the *intent* that one component influences the accessibility or usability of other components. Using an OR gate downplays and obfuscates this intent. It is, of course, appropriate to treat the FDEP gate as an OR gate during *subsequent analysis* of the fault tree.
- while the arguments of Xiang and colleagues [35] have merit, it is undeniable that scenarios exist wherein the failure of one component makes it *appear to other components in the system as if* the dependent components have failed. As far as those other components are concerned, the dependent components are inaccessible; therefore, the system must behave as if the dependent components have indeed failed.

Existing work on deriving fault trees from architecture models does attempt to introduce FDEP gates. Both Baklouti [2] and Ghadhab and colleagues [11] use a pattern-based approach to generating fault trees from a system architecture: specific arrangements of components in the architecture are replaced by specific subtrees in the generated fault tree. Some of these fault tree templates include FDEP gates. It is clear, however, that functional dependency is intimately related to the *communication structure of components*. This information is captured by AADL models as semantic connections and end-to-end flows. A directed graph could be constructed describing the flow of information between components. Generally speaking, it is then the case that a subset of components whose communication with other components must always flow through a single component C can be said to be functionally dependent on C. That is, it should be possible to define functional dependency for AADL models based on the semantics of AADL. This would only be a first step, however, because it is not clear how the dependent components should be perceived as failed. Which of their EMV2 error events should be triggered by the dependency? This may require additional intent to be expressed as AADL property associations.

13.2.2.2 Repeated Events

Nothing in the EMV2 specification says an error event cannot have multiple occurrences, and although nothing clearly states that it is permitted to have multiple occurrences, it is certainly implied. This creates a problem when converting AADL and EMV2 models to a fault tree. Consider the model in Listing 13.1: the `Checker` component tolerates two `LateDelivery` errors from the `Sensor`. On the third, the checker propagates an error of its own. Clearly, for the `Sensor` component to propagate `LateDelivery` more than once, *the `Stutter` event must occur more than once*. This is fine within the context of EMV2, but

[DISTRIBUTION STATEMENT A] Approved for public release and unlimited distribution.

when an instance of `Main.i` is used to generate a fault tree, the process described in Section 11 only introduces a single internal node representing the event `Stutter`. *This fault tree does not correctly represent the intention of the `Checker` component*:

- The fault tree represents each state transition from `None` to `One`, `One` to `Two`, and `Two` to `Three` as occurring when the event `Stutter` occurs.
- Once a basic event occurs, however, it is true forever.
- Therefore, the fault tree models that all three transitions occur simultaneously and that the `Checker` component propagates `TooManyStutters` when *only a single `Stutter` event has occurred.*

(The discussion continues after the AADL listing.)

```
package StateProblem
public
  annex EMV2 {**
    error types
      TooManyStutters: type;
    end types;

    error behavior SensorBehavior
    events
      Stutter: error event;
    states
      Normal: initial state;
    end behavior;

    error behavior MonitorBehavior
    states
      None: initial state;
      One: state;
      Two: state;
      Three: state;
    end behavior;
  **};

  device Sensor
  features
    output: out data port;
  annex EMV2 {**
    use behavior StateProblem::SensorBehavior;

    error propagations
      output: out propagation {ErrorLibrary::LateDelivery};
    end propagations;

    component error behavior
    propagations
      p2: Normal -[Stutter]-> output {ErrorLibrary::LateDelivery};
    end component;
  **};
  end Sensor;

  device Checker
```


[DISTRIBUTION STATEMENT A] Approved for public release and unlimited distribution.

```
    features
      input: in data port;
      output: out data port;
    annex EMV2 {**
      use behavior StateProblem::MonitorBehavior;

      error propagations
        input: in propagation {ErrorLibrary::LateDelivery};
        output: out propagation {StateProblem::TooManyStutters};
      end propagations;

      component error behavior
      transitions
        t1: None -[input {ErrorLibrary::LateDelivery}]-> One;
        t2: One -[input {ErrorLibrary::LateDelivery}]-> Two;
        t3: Two -[input {ErrorLibrary::LateDelivery}]-> Three;
      propagations
        p1: Three -[]-> output {StateProblem::TooManyStutters};
      end component;
    **};
  end Checker;

  system Main
  end Main;

  system implementation Main.i
  subcomponents
    sensor: device Sensor;
    checker: device Checker;
  connections
    c1: port sensor.output -> checker.input;
  end Main.i;
end StateProblem;
```

Listing 13.1: An Example Exploiting Repeated Events

The Desired Fault Tree

Because the *intention* of the `Checker` component is to transition on separate occurrences of the `Stutter` event, the fault tree should be generated differently:

- In the CFT for the `Sensor` component,
  - there should be (at least) three basic events that represent distinct occurrences of the `Stutter` event
  - there should be one out propagation port for each occurrence (i.e., for each error type event), triggered by that basic event. This is based on the specific propagation expression `Stutter`.
- Likewise, the CFT for the `Checker` component should have
  - an identical number of in propagation ports, corresponding to the multiple potential propagations of `Sensor`
  - state transitions driven by the different in propagations ports

[DISTRIBUTION STATEMENT A] Approved for public release and unlimited distribution.

This way each occurrence of `Stutter` corresponds to a distinct propagation of `LateDelivery` and a unique state transition.

- Finally, a dynamic SEQ gate (i.e., sequence gate [8]) should *impose an order* on the events representing the multiple occurrences of `Stutter`.[1] This is necessary to enforce, for example, that the second event in the sequence actually causes the transition from state `One` to state `Two`.

Towards a Solution

It is not obvious how to determine automatically that an EMV2 error event should be replicated as multiple basic events in the fault tree or even if it is universally possible to determine automatically. In the example shown here, it seems likely that the necessity can be inferred from the event being referenced in multiple states of the component `Checker` (i.e., it appears in the condition expressions for transitions starting in different states). But this is not straightforward because the references are *indirect* (i.e., via the propagation trigger `input{LateDelivery}`). It is likely that, in the general case, the fault-tree generation process would require guidance from the modeler regarding this problem. This could come in the form of additional input to the process (e.g., "allow error event `Stutter` to occur three times," "allow error event `Overflow` to occur once," and "allow error event `Dirty` to occur twice"). But even this input could be provided in multiple ways:

- It could be directly provided by the user of the model analysis tool when the AADL-to-fault tree analysis is executed.
- It could be included in the model explicitly, via AADL property associations, to permanently describe the intent of the modeler. In this way, it would be like a promise or program annotation [5], but at the system-design level.

There is no inherent problem with producing a SEQ gate during the fault tree generation process. Such gates do not have an output, but rather just serve as constraints on the evaluation of the dynamic fault tree. So they might be thought of simply "belonging" to the CFT that contains the events being ordered and ultimately "belonging" to the extracted CEG. However, as mentioned in Section 2.1.2.2, Merle's algebra that is used as the logical framework for reasoning about dynamic fault trees in this work does not model SEQ gates correctly. Of course, this does not affect evaluating the fault trees using other methods or tools, but it does result in a logical "hole" in this work. This problem would need to be addressed in the algebra, or a different logical framework would have to be used, such as Schilling [31].

13.2.3 Capturing Redundancy in System Design

The SPARE gate—or its "hot," "cold," and "warm" variants—is a prominent feature of dynamic fault trees. It handles the case where an alternate component can become active based on the failure of another component (e.g., a backup CPU comes online when the primary fails). The fault tree generation process described in this work does not introduce these gates. In fact, the semantic AADL model used throughout does not even support this concept. Activation of new components and removal of failed components are system reconfiguration problems, requiring a proper semantic description of AADL system operation modes and their transitions. Furthermore, because this process necessarily involves the coordinated behavior of multiple components (i.e., the failed component, the potential replacement components, a possible monitor component), it, like the other problems discussed above, requires additional design intent from the modeler. Which components are replaceable? What may they be replaced by? What indicates that a component should be replaced?

[1] The SEQ gate is defined in Dugan and colleagues: "The sequence-enforcing gate forces events to occur in a particular order. . . . The sequence enforcing gate can be contrasted with the priority-AND gate in that the priority-AND gate detects whether events occur in a particular order (the events can occur in any order), whereas the sequence enforcing gate allows the events to occur only in a specified order" [8].

[DISTRIBUTION STATEMENT A] Approved for public release and unlimited distribution.

Existing work does attempt to answer these questions. The process described in Ghadhab and associates is purely driven by architectural patterns where the relationship between components is assumed [11]. Specific input patterns result in specific patterns in the constructed fault tree. This work lacks a formal model for the input architecture description and uses rather generic block diagrams to represent the architecture. The work described in Baklouti and colleagues is more closely related to the work described herein: SysML models describe the system architecture [2]. Furthermore, SysML is extended with a profile to describe redundancy in the modeled system. Importantly, this profile is able to describe which sets of components are considered redundant to each other, the order in which components are replaced, which component decides if a replacement is necessary, and the nature of the redundancy (e.g., active or standby). Here, active redundancy corresponds to a "k of N" scenario and standby to replacement via hot/warm/cold spares. The different scenarios described via use of the redundancy profile reduce to specific patterns to generate in the fault tree.

A similar approach could be applied to AADL and EMV2. Instead of a "redundancy profile," an AADL "redundancy property set" could be defined containing properties that capture the redundancy information. Furthermore, Baklouti and colleagues [2] do not exploit the full power of information provided by such additional information when it is considered to be a promise about the model [5]. The redundancy information could

- inform the generation of the fault tree, specifically controlling the use of SPARE gates
- be used to derive AADL and EMV2 model fragments that ensure the desired behavior
- alternately be used to check that the existing AADL and EMV model conforms to the desired behavior

13.3 Final Remarks

The EMV2 semantics described herein is sufficient to describe AADL models containing EMV2 annex clauses as they are being produced today. The primary missing feature of the AADL model is system operation modes. In this regard, this work is no different than the many other works on AADL that also ignore modes. Describing the error behavior of interacting AADL components is future work. The semantics presented in this document are sufficient to enable a detailed formal description of how to generate CFTs from AADL components and then how to, in turn, generate a fault tree from those. The translation to the fault tree is hampered by the limited expressiveness of fault trees compared to EMV2. Finally, expanding the fault tree to exploit more detail about the system architecture is also future work. It is certainly the case that to do so effectively requires relying on explicitly provided modeler intent that can be expressed through the use of AADL property annotations.

[DISTRIBUTION STATEMENT A] Approved for public release and unlimited distribution.

References

URLs are valid as of the publication date of this document.

[1] Alfred V. Aho, John E. Hopcroft, and Jeffrey D. Ullman. *Data Structures and Algorithms*. Addison-Wesley Publishing Company, 1983.

[2] Anis Baklouti, Nga Nguyen, Faïda Mhenni, Jean-Yves Choley, and Abdelfattah Mlika. Dynamic Fault Tree Generation for Safety-Critical Systems Within a Systems Engineering Approach. *IEEE Systems Journal*, 14(1):1512–1522, March 2020.

[3] Daniel Basgöze, Matthias Volk, Joost-Pieter Katoen, Shahid Khan, and Mariëlle Stoelinga. BDDs Strike Back. In Jyotirmoy V. Deshmukh, Klaus Havelund, and Ivan Perez, editors, *NASA Formal Methods*, Lecture Notes in Computer Science, pages 713–732. Springer International Publishing, 2022.

[4] Hichem Boudali, Pepijn Crouzen, and Mariëlle Stoelinga. A Compositional Semantics for Dynamic Fault Trees in Terms of Interactive Markov Chains. In Kedar S. Namjoshi, Tomohiro Yoneda, Teruo Higashino, and Yoshio Okamura, editors, *Automated Technology for Verification and Analysis*, pages 441–456. Springer, 2007.

[5] Edwin C. Chan, John T. Boyland, and William L. Scherlis. Promises: Limited Specifications for Analysis and Manipulation. In *Proceedings of the 20th International Conference on Software Engineering*, pages 167–176. IEEE Computer Society, April 1998.

[6] A.I. Concepcion and B.P. Zeigler. DEVS Formalism: A Framework for Hierarchical Model Development. *IEEE Transactions on Software Engineering*, 14(2):228–241, February 1988.

[7] Josh Dehlinger and Joanne Bechta Dugan. Analyzing Dynamic Fault Trees Derived from Model-Based Systems Architectures. *Nuclear Engineering and Technology*, 40(5):365–374, 2008.

[8] J.B. Dugan, S.J. Bavuso, and M.A. Boyd. Dynamic Fault-Tree Models for Fault-Tolerant Computer Systems. *IEEE Transactions on Reliability*, 41(3):363–377, September 1992.

[9] Loris D'Antoni and Margus Veanes. The Power of Symbolic Automata and Transducers. In Rupak Majumdar and Viktor Kunčak, editors, *Computer Aided Verification*, Lecture Notes in Computer Science, pages 47–67. Springer International Publishing, 2017.

[10] Peter Feiler and Julien Delange. Automated Fault Tree Analysis from AADL Models. *ACM SIGAda Ada Letters*, 36(2):39–46, May 2017.

[11] Majdi Ghadhab, Sebastian Junges, Joost-Pieter Katoen, Matthias Kuntz, and Matthias Volk. Safety Analysis for Vehicle Guidance Systems with Dynamic Fault Trees. *Reliability Engineering & System Safety*, 186:37–50, June 2019.

[12] John E. Hopcroft and Jeff D. Ullman. *Introduction to Automata Theory, Languages, and Computation*. Addison-Wesley Publishing Company, 1979.

[13] Anjali Joshi, Steve Vestal, and Pam Binns. Automatic Generation of Static Fault Trees from AADL Models. June 2007. `https://conservancy.umn.edu/bitstream/handle/11299/217313/Joshi-CameraReady-WADS07.pdf`.

[14] Sebastian Junges, Dennis Guck, Joost-Pieter Katoen, and Mariëlle Stoelinga. Uncovering Dynamic Fault Trees. In *2016 46th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, pages 299–310. IEEE, June 2016. ISSN: 2158-3927.

[DISTRIBUTION STATEMENT A] Approved for public release and unlimited distribution.

[15] Bernhard Kaiser, Peter Liggesmeyer, and Oliver Mäckel. A New Component Concept for Fault Trees. In *Proceedings of the 8th Australian Workshop on Safety Critical Systems and Software, Volume 33*, pages 37–46. Australian Computer Society, Inc., October 2003.

[16] Bernhard Kaiser, Daniel Schneider, Rasmus Adler, Dominik Domis, Felix Möhrle, Axel Berres, Marc Zeller, Kai Höfig, and Martin Rothfelder. Advances in Component Fault Trees. In *Safety and Reliability – Safe Societies in a Changing World*, pages 815–823. CRC Press, 2018.

[17] Kenneth Slonneger and Barry L. Kurtz. *Formal Syntax and Semantics of Programming Languages: A Laboratory Based Approach*. Addison-Wesley Publishing Company, 1995.

[18] George H. Mealy. A Method for Synthesizing Sequential Circuits. *Bell System Technical Journal*, 34(5):1045–1079, 1955. Available at `https://onlinelibrary.wiley.com/doi/pdf/10.1002/j.1538-7305.1955.tb03788.x`.

[19] Guillaume Merle. *Algebraic Modelling of Dynamic Fault Trees, Contribution to Qualitative and Quantitative Analysis*. PhD Thesis, École normale supérieure de Cachan - ENS Cachan, July 2010. `https://tel.archives-ouvertes.fr/tel-00502012`.

[20] Zhibao Mian, Leonardo Bottaci, Yiannis Papadopoulos, and Nidhal Mahmud. Model Transformation for Analyzing Dependability of AADL Model by Using HiP-HOPS. *Journal of Systems and Software*, 151:258–282, May 2019.

[21] S. Montani, L. Portinale, A. Bobbio, M. Varesio, and Codetta-Raiteri. A Tool for Automatically Translating Dynamic Fault Trees into Dynamic Bayesian Networks. In *RAMS '06. Annual Reliability and Maintainability Symposium, 2006*, pages 434–441. IEEE, January 2006. ISSN: 0149-144X.

[22] G.J. Pai and J.B. Dugan. Automatic Synthesis of Dynamic Fault Trees from UML System Models. In *13th International Symposium on Software Reliability Engineering, 2002. Proceedings*, pages 243–254. IEEE, November 2002. ISSN: 1071-9458.

[23] Michael O. Rabin. Probabilistic Automata. *Information and Control*, 6(3):230–245, September 1963.

[24] SAE International. Guidelines and Methods for Conducting the Safety Assessment Process on Civil Airborne Systems and Equipment. Technical Report Standards Document ARP4761, SAE International, 1996.

[25] SAE International. SAE Architecture Analysis and Design Language (AADL) Annex Volume 1: Annex A: Graphical AADL Notation, Annex C: AADL Meta-Model and Interchange Formats, Annex D: Language Compliance and Application Program Interface, Annex E: Error Model Annex. Technical Report Standards Document AS5506/1, SAE International, 2011.

[26] SAE International. Architecture Analysis & Design Language (AADL). Technical Report Standards Document AS5506 Rev. C, SAE International, 2017.

[27] SAE International. SAE Architecture Analysis and Design Language (AADL) Annex Volume 5: Annex A: ARINC653 Annex, Annex C: Code Generation Annex, Annex E: Error Model Annex. Technical Report Standards Document AS5506/5, SAE International, 2020.

[28] David Schmidt. *Denotational Semantics: A Methodology for Language Development*. 1997. `https://people.cs.ksu.edu/~schmidt/text/densem.html`.

[29] Septavera Sharvia and Yiannis Papadopoulos. Non-coherent Modelling in Compositional Fault Tree Analysis. In *IFAC Proceedings*, volume 41 of *17th IFAC World Congress*, pages 4138–4143. Elsevier, January 2008.

[DISTRIBUTION STATEMENT A] Approved for public release and unlimited distribution.

[30] Simon J. Schilling. *Beitrag zur dynamischen Fehlerbaumanalyse ohne Modulbildung und zustandsbasierte Erweiterungen*. PhD Thesis, Bergische Universität Wuppertal, 2009. https://elekpub.bib.uni-wuppertal.de/urn/urn:nbn:de:hbz:468-20100070.

[31] Simon J. Schilling. *Contribution to Temporal Fault Tree Analysis without Modularization and Transformation into the State Space*. PhD Thesis, Bergische Universität Wuppertal, 2009. English translation of [30]. https://arxiv.org/abs/1505.04511.

[32] Mariëlle Stoelinga. An Introduction to Probabilistic Automata. *Bulletin of the EATCS*, 78(2):176–198, October 2002.

[33] Margus Veanes, Peli de Halleux, and Nikolai Tillmann. Rex: Symbolic Regular Expression Explorer. In *Verification and Validation 2010 Third International Conference on Software Testing*, pages 498–507. IEEE, April 2010. ISSN: 2159-4848.

[34] William Vesely, Joanne Dugan, Joseph Fragola, Joseph Minarick III, and Jan Railsback. *Fault Tree Handbook with Aerospace Applications, Version 1.1*. National Aeronautics and Space Administration, 2002.

[35] Jianwen Xiang, Kazuo Yanoo, Yoshiharu Maeno, and Kumiko Tadano. Automatic Synthesis of Static Fault Trees from System Models. In *2011 Fifth International Conference on Secure Software Integration and Reliability Improvement*, pages 127–136. IEEE, June 2011.

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-188

Public reporting burden for this collection of information is estimated to average 1 hour per response, including the time for reviewing instructions, searching existing data sources, gathering and maintaining the data needed, and completing and reviewing the collection of information. Send comments regarding this burden estimate or any other aspect of this collection of information, including suggestions for reducing this burden, to Washington Headquarters Services, Directorate for information Operations and Reports, 1215 Jefferson Davis Highway, Suite 1204, Arlington, VA 22202-4302, and to the Office of Management and Budget, Paperwork Reduction Project (0704-0188), Washington, DC 20503.

1. AGENCY USE ONLY	2. REPORT DATE	3. REPORT TYPE AND DATES COVERED
(Leave Blank)	September 2025	Final

4. TITLE AND SUBTITLE	5. FUNDING NUMBERS
A Semantics of AADL EMV2 and Its Application to Model-Based Fault Tree Generation	FA870225DB003

6. AUTHORS
Aaron Greenhouse, Jérôme Hugues, Sam Procter, Lutz Wrage, Joseph R. Seibel

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)	8. PERFORMING ORGANIZATION REPORT NUMBER
Software Engineering Institute Carnegie Mellon University Pittsburgh, PA 15213	CMU/SEI-2025-TR-002

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)	10. SPONSORING/MONITORING AGENCY REPORT NUMBER
SEI Administrative Agent AFLCMC/AZS 5 Elgin Street Hanscom AFB, MA 01731-2100	N/A

11. SUPPLEMENTARY NOTES

12A. DISTRIBUTION/AVAILABILITY STATEMENT	12B. DISTRIBUTION CODE
Unclassified/Unlimited, DTIC, NTIS	

13. ABSTRACT (MAXIMUM 200 WORDS)

The Architecture Analysis & Design Language (AADL) is an SAE International standard for the design and analysis of both the software and hardware architecture of performance-critical real-time systems. The Error Model Annex, Version 2 (EMV2), extends AADL with concepts to perform safety analyses, such as error types, error propagations, and the impact of errors propagated on components. EMV2 builds on AADL concepts of components and ports to define error propagations and error state machines. These definitions rely on a precise definition of the effect of an error being triggered in this system. The definitions of these concepts rely on powerful abstractions, yet they are mostly defined in plain text. This report first proposes a formal semantics for EMV2. Then, it shows how to leverage this semantics to generate fault trees from an AADL model enriched with EMV2 information. Defining a formal semantics improves the understanding of the EMV2 model, and the precision of model transformation from EMV2 to analysis techniques.

14. SUBJECT TERMS	15. NUMBER OF PAGES
	106

16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT	18. SECURITY CLASSIFICATION OF THIS PAGE	19. SECURITY CLASSIFICATION OF ABSTRACT	20. LIMITATION OF ABSTRACT
Unclassified	Unclassified	Unclassified	UL

NSN 7540-01-280-5500

Standard Form 298 (Rev. 2-89) Prescribed by ANSI Std. Z39-18
298-102

[DISTRIBUTION STATEMENT A] Approved for public release and unlimited distribution.