Safety Annex Users Guide

Version 0.8

Danielle Stewart, University of Minnesota

Jing (Janet) Liu, Collins Aerospace

Darren Cofer, Collins Aerospace  
Mike Whalen, University of Minnesota

Mats Heimdahl, University of Minnesota

Version History

|  |  |  |  |
| --- | --- | --- | --- |
| Version | Date | Author | Information |
| 0.1 | 9/1/2017 | Danielle Stewart | Initial version of the Safety Annex Users Guide. |
|  |  |  |  |
| 0.2 | 3/22/2018 | Danielle Stewart | Updates to tool and grammar |
|  |  |  |  |
| 0.3 | 4/2/2018 | Danielle Stewart | Updates to installation instructions |
| 0.4 | 9/21/2018 | Danielle Stewart | Compositional generation of artifacts, SOTERIA installation instructions, OSATE development environment installation instructions. |
|  |  |  |  |

|  |  |  |  |
| --- | --- | --- | --- |
| 0.5 | 12/18/2018 | Danielle Stewart  Janet Liu | Removed SOTERIA install instructions and updated OSATE user and development environment installation instructions. |

0.6 6/17/2019 Danielle Stewart Added section 5.4.1.6 Propagate Type

Statement describing asymmetric faults.

0.7 7/9/2019 Danielle Stewart Added running example to grammar

description.

# Table of Contents

[1 Table of Contents 3](#_Toc18580590)

[2 Table of Figures 6](#_Toc18580591)

[3 Introduction 7](#_Toc18580592)

[3.1 Github Repositories 7](#_Toc18580593)

[4 Brief Overview of AADL, AGREE, and the Safety Annex 8](#_Toc18580594)

[4.1 Using the Safety Annex AADL Plugin 11](#_Toc18580595)

[5 Safety Annex Language 18](#_Toc18580596)

[5.1 Syntax Overview 18](#_Toc18580597)

[5.2 Lexical Elements and Types 19](#_Toc18580598)

[5.3 Subclauses 20](#_Toc18580599)

[5.4 Spec Statement 22](#_Toc18580600)

[5.4.1 Fault Statement 22](#_Toc18580601)

[5.4.1.1 Input Statement 23](#_Toc18580602)

[5.4.1.2 Output Statement 24](#_Toc18580603)

[5.4.1.3 Probability Statement 25](#_Toc18580604)

[5.4.1.4 Duration Statement 25](#_Toc18580605)

[5.4.1.5 Propagate-Type Statement 25](#_Toc18580606)

[5.4.1.6 Propagate-From Statement 27](#_Toc18580607)

[5.4.1.7 Safety Equation Statements 27](#_Toc18580608)

[5.4.1.7.1 Eq Statements 27](#_Toc18580609)

[5.4.2 Analysis Statement 28](#_Toc18580610)

[5.4.2.1 Max N Faults Analysis 29](#_Toc18580611)

[5.4.2.2 Probabilistic Analysis 29](#_Toc18580612)

[5.4.3 Fault Activation Statement 30](#_Toc18580613)

[5.4.4 Hardware Fault Statement 31](#_Toc18580614)

[5.4.4.1 Duration, Probability, and Propagation-Type Statements 32](#_Toc18580615)

[6 Library and Custom Made Fault Nodes 32](#_Toc18580616)

[6.1 Nondeterministic Failure Values 33](#_Toc18580617)

[6.2 Nested Data Structures 34](#_Toc18580618)

[7 In Depth Examples 35](#_Toc18580619)

[7.1 The Sensor Example 35](#_Toc18580620)

[7.1.1 AADL Architecture 35](#_Toc18580621)

[7.1.2 AGREE Behavioral Model 36](#_Toc18580622)

[7.1.3 Safety Model 37](#_Toc18580623)

[7.1.4 The 4 types of Analysis Results 38](#_Toc18580624)

[7.1.4.1 Verify All Layers with Max N Faults present 39](#_Toc18580625)

[7.1.4.2 Verify All Layers with Faults Present: Probabilistic Analysis 41](#_Toc18580626)

[7.1.4.3 Generate Minimal Cut Sets with Max N Faults 42](#_Toc18580627)

[7.1.4.4 Generate Minimal Cut Sets with Probability Threshold 43](#_Toc18580628)

[7.2 Byzantine Examples 44](#_Toc18580629)

[7.2.1 Asymmetric Fault Implementation 44](#_Toc18580630)

[7.2.2 Mitigation Strategy 45](#_Toc18580631)

[7.2.3 Color Exchange Example 45](#_Toc18580632)

[7.2.3.1 Color Exchange Architecture in AADL 46](#_Toc18580633)

[7.2.3.2 Color Exchange Mitigation Strategy 46](#_Toc18580634)

[7.2.3.3 Color Exchange Fault Model and Analysis 49](#_Toc18580635)

[7.2.4 Process ID Example 50](#_Toc18580636)

[7.2.4.1 PID Example AADL Architecture 50](#_Toc18580637)

[7.2.4.2 PID Example AGREE Behavioral Model with Mitigation Strategy 51](#_Toc18580638)

[7.2.4.3 PID Example Fault Model 52](#_Toc18580639)

[7.2.4.4 PID Example Analysis Results 53](#_Toc18580640)

[8 The Tool Suite (Safety Annex, AGREE, AADL) 54](#_Toc18580641)

[8.1 Tool Suite Overview 54](#_Toc18580642)

[8.2 Installation 55](#_Toc18580643)

[8.2.1 Install OSATE 55](#_Toc18580644)

[8.2.2 Install Safety Annex 56](#_Toc18580645)

[8.2.3 Install SMT Solver 58](#_Toc18580646)

[8.2.4 Set AGREE Analysis Preferences 60](#_Toc18580647)

[8.3 Development Environment Installation 61](#_Toc18580648)

[8.3.1 Install OSATE Development Environment 61](#_Toc18580649)

[8.3.2 Download Safety Annex Source Code 62](#_Toc18580650)

[8.3.3 Github Branches 62](#_Toc18580651)

[8.3.4 Run OSATE 62](#_Toc18580652)

[9 Appendices 63](#_Toc18580653)

[9.1 Appendix 1: Fault Library 63](#_Toc18580654)

# Table of Figures

[Figure 1: Toy Example for Safety Annex and AGREE 8](#_Toc18580655)

[Figure 2: AADL Code for Toy Example with AGREE and Safety Annexes 10](#_Toc18580656)

[Figure 3: Fault Hypothesis Example 11](#_Toc18580657)

[Figure 4: Import Menu Option 12](#_Toc18580658)

[Figure 5: Importing Toy Example Project 13](#_Toc18580659)

[Figure 6: Workspace After Importing Toy Example 14](#_Toc18580660)

[Figure 7: AGREE and Safety Analysis Dropdown Menu 15](#_Toc18580661)

[Figure 8:AGREE Verification Results 15](#_Toc18580662)

[Figure 9: Counterexample from Safety Analysis 17](#_Toc18580663)

[Figure 10: Generated Excel File for Counterexample 18](#_Toc18580664)

[Figure 11: Medical Device Example 21](#_Toc18580665)

[Figure 12: Safety Annex Grammar 22](#_Toc18580666)

[Figure 13: Toy Example System A 23](#_Toc18580667)

[Figure 14: Toy Example System A Safety Annex 24](#_Toc18580668)

[Figure 15: Fault Node Definition 24](#_Toc18580669)

[Figure 16: Fault node wrapping output of AADL component 25](#_Toc18580670)

[Figure 17: Sender-Receiver AADL Model 27](#_Toc18580671)

[Figure 18: Fault Statement With Asymmetric Propagate Type 27](#_Toc18580672)

[Figure 19: Propagation Statement Example 28](#_Toc18580673)

[Figure 20: Max One Fault Analysis Statement Example 30](#_Toc18580674)

[Figure 21: Probability Threshold Example 30](#_Toc18580675)

[Figure 22: Hardware Fault Statement 33](#_Toc18580676)

[Figure 23: Sensor Example Architecture 37](#_Toc18580677)

[Figure 24: AADL and AGREE Sensor Subcomponent 38](#_Toc18580678)

[Figure 25: Top Level Reactor System and Top Level Property 38](#_Toc18580679)

[Figure 26: Fault Definition for Pressure Sensor 39](#_Toc18580680)

[Figure 27: Fault Node Definition for Sensor Fault 39](#_Toc18580681)

[Figure 28: Subcomponent Organization of Sensor System 39](#_Toc18580682)

[Figure 29: Verification Results for Max 2 Faults on Sensor Example 41](#_Toc18580683)

[Figure 30: Counterexample from Sensor Analysis with 2 Faults Active 41](#_Toc18580684)

[Figure 31: Asymmetric Fault Implementation Strategy 45](#_Toc18580685)

[Figure 32: Color Exchange Architecture for Asymmetric Modeling and Mitigation 47](#_Toc18580686)

[Figure 33: PID Example Architecture 51](#_Toc18580687)

[Figure 34: Outputs for Node in PID Example 53](#_Toc18580688)

[Figure 35: Overview of Safety Annex/AGREE/OSATE Tool Suite 56](#_Toc18580689)

[Figure 36: OSATE Loading Screen 57](#_Toc18580690)

[Figure 37: Safety Analysis Menu Item 58](#_Toc18580691)

[Figure 38: Windows 10 System Control Panel 59](#_Toc18580692)

[Figure 39: System Properties Dialog Box 60](#_Toc18580693)

[Figure 40: Environment Variables Dialog Box 60](#_Toc18580694)

[Figure 41: System Variable Text Edit Box 61](#_Toc18580695)

[Figure 42: AGREE Analysis Preferences 62](#_Toc18580696)

# Introduction

System safety analysis techniques are well-established and are a required activity in the development of safety-critical systems. Model-based systems engineering (MBSE) methods and tools based on formal methods now permit system-level requirements to be specified and analyzed early in the development process. While model-based development methods are widely used in the aerospace industry, they are only recently being applied to system safety analysis.

The Safety Annex for the Architecture Analysis and Design Language (AADL) provides the ability to reason about faults and faulty component behaviors in AADL models. In the Safety Annex approach, we use formal assume-guarantee contracts to define the nominal behavior of system components. The nominal model is then verified using the Assume Guarantee Reasoning Environment (AGREE). The Safety Annex provides a way to weave faults into the nominal system model and analyze the behavior of the system in the presence of faults. The Safety Annex also provides a library of common fault node definitions that is customizable to the needs of system and safety engineers.

The Safety Annex supports model checking and quantitative reasoning by attaching behavioral faults to components and then using the normal behavioral propagation and proof mechanisms built into the AGREE AADL annex. This allows users to reason about the evolution of faults over time, and produce counterexamples demonstrating how component faults lead to system failures. It can serve as the shared model to capture system design and safety-relevant information, and produce both qualitative and quantitative description of the causal relationship between faults/failures and system safety requirements.

This Users Guide is organized as follows. Section 2 provides a brief overview of AADL, AGREE, and the Safety Annex. Section 3 gives examples and explanations of the grammar and language of the safety annex. Section 4 provides a detailed approach for the tool suite and downloads.

## Github Repositories

Throughout this guide, we often refer users to the Safety Annex Github repository for access to examples and models. This is located at: <https://github.com/loonwerks/AMASE/tree/develop/examples>

We also refer to the AGREE Users Guide for certain syntactical elements and more descriptions on how this annex is used. This is located at:

<https://github.com/smaccm/smaccm/tree/master/documentation/agree>

# Brief Overview of AADL, AGREE, and the Safety Annex

The safety annex is meant to be used in the context of an AADL model that has been annotated with AGREE. AGREE models the components and their connections as they are described in AADL and the safety annex provides fault definitions to these components and connections. This section provides a very brief introduction to AADL, AGREE, and the safety annex through the use of a very simple model.

Suppose we have a simple architecture with three subcomponents A, B, and C, as shown in Figure 1.



Figure 1: Toy Example for Safety Annex and AGREE

We want to show using AGREE that the system level property (Output < 50) holds, given the guarantees provided by the components and the system assumption (Input < 10). We also want to be able to model faults on each of these components. Some possible faults are shown in the diagram of Figure 1.

In order to represent this model in AADL, we construct an AADL package. Packages are the structuring mechanism in AADL; they define a namespace where we can place definitions. We define the subcomponents first, then the system component. The complete AADL is shown in Figure 2 below.

**package** Integer\_Toy

**public**

**with** Base\_Types;

**with** faults;

**system** A

**features**

Input: **in** **data** **port** Base\_Types::Integer;

Output: **out** **data** **port** Base\_Types::Integer;

**annex** agree {\*\*

**assume** "A input range" **:** Input **<** 20**;**

**guarantee** "A output range" **:** Output **<** 2**\***Input**;**

\*\*};

**annex** safety {\*\*

**fault** stuck\_at\_fault\_A "Component A output stuck" **:** faults**.**fail\_to **{**

**inputs:** val\_in **<-** Output**,** alt\_val **<-** **prev(**Output**,** 0**);**

**outputs:** Output **<-** val\_out**;**

**probability:** 5.0E-5 **;**

**duration:** **permanent;**

**}**

\*\*};

**end** A ;

**system** B

**features**

Input: **in** **data** **port** Base\_Types::Integer;

Output: **out** **data** **port** Base\_Types::Integer;

**annex** agree {\*\*

**assume** "B input range" **:** Input **<** 20**;**

**guarantee** "B output range" **:** Output **<** Input **+** 15**;**

\*\*};

**annex** safety {\*\*

**fault** stuck\_at\_fault\_B "Component B output nondeterministic" **:** faults**.**fail\_to **{**

**eq** nondet\_val **:** **int;**

**inputs:** val\_in **<-** Output**,** alt\_val **<-** nondet\_val**;**

**outputs:** Output **<-** val\_out**;**

**probability:** 5.0E-9 **;**

**duration:** **permanent;**

**}**

\*\*};

**end** B ;

**system** C

**features**

Input1: **in** **data** **port** Base\_Types::Integer;

Input2: **in** **data** **port** Base\_Types::Integer;

Output: **out** **data** **port** Base\_Types::Integer;

**annex** agree {\*\*

**eq** mode **:** **int;**

**guarantee** "mode always is increasing" **:** mode **>=** 0 **->** mode **>** **pre(**mode**);**

**guarantee** "C output range" **:** Output **=** **if** mode **=** 3 **then** **(**Input1 **+** Input2**)** **else** 0**;**

\*\*};

**end** C ;

**system** top\_level

**features**

Input: **in** **data** **port** Base\_Types::Integer;

Output: **out** **data** **port** Base\_Types::Integer;

**annex** agree {\*\*

**eq** mode **:** **int;**

**assume** "System input range " **:** Input **<** 10**;**

**guarantee** "mode is always positive" **:** mode **>=** 0**;**

**guarantee** "System output range" **:** Output **<** 50**;**

\*\*};

**end** top\_level;

**system** **implementation** top\_level.Impl

**subcomponents**

A\_sub : **system** A ;

B\_sub : **system** B ;

C\_sub : **system** C ;

**connections**

IN\_TO\_A : **port** Input -> A\_sub.Input

{Communication\_Properties::Timing => immediate;};

A\_TO\_B : **port** A\_sub.Output -> B\_sub.Input

{Communication\_Properties::Timing => immediate;};

A\_TO\_C : **port** A\_sub.Output -> C\_sub.Input1

{Communication\_Properties::Timing => immediate;};

B\_TO\_C : **port** B\_sub.Output -> C\_sub.Input2

{Communication\_Properties::Timing => immediate;};

C\_TO\_Output : **port** C\_sub.Output -> Output

{Communication\_Properties::Timing => immediate;};

**annex** agree{\*\*

**assign** mode **=** C\_sub**.**mode**;**

\*\*};

**annex** safety{\*\*

**analyze** **:** **probability** 1.0E-7

--analyze : max 1 fault

\*\*};

**end** top\_level.Impl;

toy1

Figure 2: AADL Code for Toy Example with AGREE and Safety Annexes

In Figure 2, **system**s define hierarchical "units" of the model. They communicate over **port**s**,** whichare typed. Systems do not contain any internal structure, only the interfaces for the system.

A **system** **implementation** describes an implementation of the system including its internal structure. For this example, the only system whose internal structure is known is the "top level" system, which contains subcomponents A, B, and C. We instantiate these subcomponents (using A\_sub, B\_sub, and C\_sub) and then describe how they are connected together. In the connections section, we must describe whether each connection is *immediate* or *delayed.* Intuitively, if a connection is *immediate,* then an output from the source component is *immediately* available to the input of the destination component (i.e., in the same frame). If they are *delayed*, then there is a one-cycle delay before the output is available to the destination component (delayed frame).

**Note:** Top level analysis can be performed only within a system implementation. For more information, see Section 5.4.2.

After the AGREE annexes are added to each of the components in the model and verification is complete, the safety annexes can be added to each of the components.

**annex** safety{\*\*

**analyze** **:** **max** 1 **fault**

\*\*};

Figure 3: Fault Hypothesis Example

## Using the Safety Annex AADL Plugin

The example project used in the rest of this section can be retrieved from the following link:

<https://github.com/loonwerks/AMASE/tree/develop/examples>. Assuming the necessary tools are installed (see section 6), the model can be imported by choosing File > Import:



Figure 4: Import Menu Option

Then choosing "Existing Project into Workspace."



Figure 5: Importing Toy Example Project

and navigate to the unzipped directory after pressing the Next button. Figure 6 shows what the model looks like when loaded in the AGREE/OSATE tool. The project that we are working with is called Toy\_Example\_Safety.



Figure 6: Workspace After Importing Toy Example

Open the Integer\_Toy.aadl model by double-clicking on the file in the AADL Navigator pane. To invoke the safety analysis, we select the Top\_Level.Impl system implementation in the outline pane on the right. We then select “Safety Analysis” in the menu and then run AGREE. We can choose “AGREE > Verify Single Layer” from the AGREE menu as shown in Figure 7.



Figure 7: AGREE and Safety Analysis Dropdown Menu

As AGREE runs, you should see checks for “Contract Guarantees”, ”Contract Assumptions”, and “Contract Consistency” as shown in Figure 8.



Figure 8:AGREE Verification Results

If “Safety Analysis” was checked by the user, this will run the analysis and will change the AGREE contracts accordingly.

When a property fails in AGREE, there is an associated counterexample that demonstrates the failure. To see the counterexample, right-click the failing property (in this case: "System output range") and choose "View Counterexample in Console" to see the values assigned to each of the variables referenced in the model. Figure 9 shows the counterexample that is generated by this failure in the console window given one permanent fault in the system.

It is worth noting in the counterexample of Figure 9 that the faults assigned to components A and B are listed as “Component A output stuck” and “Component B output stuck nondeterministic.” These are the strings assigned to the fault definitions from Figure 2.

It is also possible that each fault has a probability of occurrence. In the Toy Example safety annexes, an arbitrary probability is assigned to each fault for the illustrative purposes. A top level probabilistic threshold is assigned. Assuming independence of faults, safety analysis proceeds by determining if there are sets of faults that will cause the system to fail given this threshold. Figure 7 shows the analysis of the Toy Example given a top level probability threshold of 1.0E-7. The reason this passes the threshold is due to the fact that the most problematic of the faults is with component B (nondeterministic failure) and the probability of component B fault is 1.0E-9. This is beyond the threshold assigned at the lower level.



Figure 9: Counterexample from Safety Analysis

For working with complex counterexamples, it is often necessary to have a richer interface. It is also possible to export the counterexample to Excel by right-clicking the failing property and choosing "View Counterexample in Excel". **Note: In order to use this capability, you must have Excel installed on your computer. Also, you must associate .xls files in Eclipse with Excel.** To do so, the following steps can be taken:

1. Choose the "Preferences" menu item from the Window menu, then
2. On the left side of the dialog box, choose General > Editors > File Associations, then
3. Click the "Add…" button next to "File Types" and then
4. Type "\*.xls" into the text box.

The .xls file type should now be selected.

1. Now choose the "Add…" button next to "Associated Editors"
2. Choose the "External Programs" radio button
3. Select "Microsoft Excel Worksheet" and click OK.

The generated Excel file for the example is shown in Figure 10.



Figure 10: Generated Excel File for Counterexample

# Safety Annex Language

In this chapter we present the syntax and semantics of the input language of the Safety Annex. We refer readers to the AGREE Users Guide for a thorough description of lexical elements, types, and other syntactical details.

## Syntax Overview

Before describing the details of the language, we provide some general notes about the syntax. productions enclosed in parentheses (‘()’) indicate a set of choices in which a vertical bar (‘|’) is used to separate alternatives in the syntax rules. Any characters in single quotes describe concrete syntax (e.g. ‘←’, ‘;’, ‘:’). Examples of grammar fragments are also written in the Courier font. Sometimes one of the following characters is used at the beginning of a rule as a shorthand for choosing among several alternatives:

1) The \* character indicates repetition: zero or more occurrences and the + character indicates required repetition: one or more occurrences.

2) A ? character indicates that the preceding token is optional.

The Safety Annex is built on top of the AADL 2.0 architecture description language as well as the AGREE language. The Safety Annex formulas are found in an AADL annex which extends the grammar of both AADL and AGREE. Generally, the annex follows the conventions of AADL in terms of lexical elements and types with some small deviations (which are noted in the AGREE Users Guide). The Safety Annex operates over a relatively small fragment of both AADL syntax and AGREE syntax. We will not build up the language starting from the smallest fragments, but instead refer the user to the AGREE User Manual: <https://github.com/smaccm/smaccm/tree/master/documentation/agree>

AADL describes the interface of a component in a *component type*. A *component type* contains a list of *features* that are inputs and outputs of a component and possibly a list of AADL properties. A *component implementation* is used to describe a specific instance of a *component type*. A *component implementation* contains a list of subcomponents and a list of connections that occur between its subcomponents and features.

The syntax for a component’s contract exists in an AGREE annex placed inside of the *component type*. AGREE syntax can also be placed inside of annexes in a *component implementation* or an AADL package. Syntax placed in an annex in an AADL package can be used to create libraries that can be referenced by other components.

The syntax for a component’s faults exists in a Safety annex placed inside of the *component type* or in a *component implementation*.During grammar examples and descriptions, it will be clear which syntax goes in the type and which goes in implementation.

## Lexical Elements and Types

For a more thorough description of lexical elements and types, we refer to the AGREE User Guide. Here is a brief description of commonly used lexical elements.

Comments always start with two adjacent hyphens and span to the end of the line. Here is an example:

-- Here is a comment.

-- a long comment may be split onto

-- two or more consecutive lines

An identifier is defined as a letter followed by zero or more letters, digits, or single underscores:

ID ::= identifier\_letter ( ('\_')? letter\_or\_digit)\*

letter\_or\_digit ::= identifier\_letter | digit

identifier\_letter ::= ('A'..'Z' | 'a'..'z')

digit ::= (0..9)

Some example identifiers are: count, X, Get\_Name, Page\_Count. **Note: Identifiers are case insensitive.** Thus Hello, HeLlo, and HELLO all refer to the same entity in AADL.

Boolean and numeric literal values are defined as follows:

Literal :: = Boolean\_literal | Integer\_literal | Real\_literal

Integer\_literal ::= decimal\_integer\_literal

Real\_literal ::= decimal\_real\_literal

decimal\_integer\_literal ::= ('–')? numeral

decimal\_real\_literal ::= ('–')? numeral '.' numeral

numeral ::= digit\*

Boolean\_literal are: true, false.

Examples of Integer\_literals are: 1, 31, -1053

Examples of Real\_literals are: 3.1415, 0.005, 7.01

String elements are defined with the following syntax:

STRING ::= "(string\_element)\*"

string\_element ::= "" | non\_quotation\_mark\_graphic\_character

Primitive data types (bool, int, real) have been built into the AGREE language and are hence part of the Safety annex language. For more information on types, see the AGREE Users Guide.

Safety annex requires reasoning about AADL Data Implementations. Consider the following example from a model of a medical device:

**data** Alarm\_Outputs

**end** Alarm\_Outputs;

**data** **implementation** Alarm\_Outputs.Impl

**subcomponents**

Is\_Audio\_Disabled : **data** Base\_Types::Boolean;

Notification\_Message : **data** Base\_Types::Integer ;

Log\_Message\_ID : **data** Base\_Types::Integer ;

**end** Alarm\_Outputs.Impl;

**placeholder**

Figure 11: Medical Device Example

One can reference the fields of a variable type *Alarm\_Outputs.Impl* by placing a dot after the variable:

*Alarm.Is\_Audio\_Disabled*, *Alarm.Notification\_Message*, or *Alarm.Log\_Message\_ID*.

## Subclauses

Safety annex subclauses can be embedded in *system, process,* and *thread* components. Safety subclauses are of the form:

**annex** safety {\*\*

-- safety spec statements here...

\*\*};

From within the subclause (annex), it is possible to refer to the features and properties of the enclosing component. A simplified description of the top-level grammar for Safety annex is shown in Figure 12.

SpecStatement: 'fault' ID (STRING)? ':' faultDefName '{' (FaultSubcomponent)\* '}'

| 'analyze' ':' AnalysisBehavior

| ‘hw\_fault ':' ID (STRING)? ':' '{' (HWFaultSubcomponent)\* '}'

| 'propagate\_from' ':' '{' (SourceFaultList) '@' (SourceCompPath) '}'

‘to’ '{' (DestFaultList) '@' (DestCompPath) '}'

AnalysisBehavior: 'max' Int\_Literal 'fault'

| 'probability' Real\_Literal

FaultSubcomponent: 'inputs' ':' NamedID '<-' Expr (','NamedID '<-' Expr)\* ';'

| 'outputs' ':' NestedDotID '<-' NamedID (','NestedDotID '<-' NamedID)\* ';'

| 'duration' ':' TemporalConstraint (Interval)? ';'

| 'probability' ':' Real\_Literal ';'

| 'enabled' ':' TriggerCondition ';'

| 'propagate\_type' ':' PropagationTypeConstraint ';'

| SafetyEqStatement

HWFaultSubcomponent: 'duration' ':' TemporalConstraint (Interval)? ';'

| 'probability' ':' Real\_Literal ';'

| 'propagate\_type' ':' PropagationTypeConstraint ';'

PropagationTypeConstraint: 'asymmetric'

| 'symmetric'

TemporalConstraint: 'permanent'

SafetyEqStatement: 'eq' (Arg (',' Arg)\*) ('=' Expr)? ';'

**placeholder**

Figure 12: Safety Annex Grammar

A Safety subclause consists of a spec statement which consists of a sequence of statements. Safety subclauses can occur either within an AADL component or component implementation.

In order to fully describe the grammar of the Safety Annex, we will provide a running example as each grammar component is described. The Toy Example given in Figure 2 contains a system component “A” (Figure 13) which has an input and an output. The contract (in AGREE) regarding the behavior of the output is that the range is 2 times the input.

We will use this AADL code fragment to illustrate the grammar and usage of the Safety Annex below.

To define a fault on this component, it must be clear that the fault changes the output of said component, so this is what it will be attached to behind the scenes. There are numerous ways that the component can fail. It could fail to zero, some arbitrary number, or perhaps it will simply output the input value without multiplying it. Depending on the component in a real system model, these faults will vary. For now, we will look at an example of a fault that gets stuck at the previous value that was sent out from component A.

**system** A

**features**

Input: **in** **data** **port** Base\_Types::Integer;

Output: **out** **data** **port** Base\_Types::Integer;

**annex** agree {\*\*

**assume** "A input range" **:** Input **<** 20**;**

**guarantee** "A output range" **:** Output **<** 2**\***Input**;**

\*\*};

**annex** safety {\*\*

**fault** stuck\_at\_fault\_A "Component A output stuck" **:** faults**.**fail\_to **{**

**inputs:** val\_in **<-** Output**,** alt\_val **<-** **prev(**Output**,** 0**);**

**outputs:** Output **<-** val\_out**;**

**probability:** 5.0E-5**;**

**duration:** **permanent;**

**}**

\*\*};

**end** A ;

Figure 13: Toy Example System A

## Spec Statement

The subclause consists of the keywords “**annex safety** {\*\* \*\*}” and can contain one or more spec statements. The following shows the syntax of a spec statement:

SpecStatement: 'fault' ID (STRING)? ':' faultDefName '{' (FaultSubcomponent)\* '}'

| 'analyze' ':' AnalysisBehavior

| ‘hw\_fault ':' ID (STRING)? ':' '{' (HWFaultSubcomponent)\* '}'

| 'propagate\_from' ':' '{' (SourceFaultList) '@' (SourceCompPath) '}'

‘to’ '{' (DestFaultList) '@' (DestCompPath) '}'

A simple example is :

**annex** safety {\*\*

**fault** stuck\_at\_fault\_A "Component A output stuck" **:** faults**.**fail\_to **{**

-- fault subcomponents here...

**}**

\*\*};

This shows the subclause (**annex**) and a single spec statement (**fault**).

The location and usage of the spec statements depend on which component in AADL we are in, so initially the “**fault**” spec statement will be described.

### Fault Statement

The Safety annex spec statement can contain multiple Fault Subcomponent statements. The following is a simplified version of the syntax of a Fault Subcomponent statement:

FaultSubcomponent: 'inputs' ':' NamedID '<-' Expr (','NamedID '<-' Expr)\* ';'

| 'outputs' ':' NestedDotID '<-' NamedID (','NestedDotID '<-' NamedID)\* ';'

| 'duration' ':' TemporalConstraint (Interval)? ';'

| 'probability' ':' Real\_Literal ';'

| 'enabled' ':' TriggerCondition ';'

| 'propagate\_type' ':' PropagationTypeConstraint ';'

| SafetyEqStatement

Each *fault* spec statement corresponds with one fault node definition that will wrap the output of a single AADL component. Continuing with the Toy Example, we refer to the fault subcomponents located in the fault definition given in Figure 14.

**annex** safety {\*\*

**fault** stuck\_at\_fault\_A "Component A output stuck" **:** faults**.**fail\_to **{**

**inputs:** val\_in **<-** Output**,** alt\_val **<-** **prev(**Output**,** 0**);**

**outputs:** Output **<-** val\_out**;**

**probability:** 5.0E-5**;**

**duration:** **permanent;**

**}**

\*\*};

Figure 14: Toy Example System A Safety Annex

Immediately following the keyword “**fault**,” which denotes the SpecStatement of the grammar, the user gives a unique fault name, or ID. The ID is used as an internal identification to the fault described in the spec statement. The STRING is a description of the fault and will be shown to the user during verification. The fault definition name (a NestedDotID) corresponds with a fault contained in a library of faults. Each of the faults is an AGREE node definition that is placed within an AADL package and included in AADL package file. These faults can then be referenced by the Safety annex. In the case when the user wishes to design custom faults, refer to the AGREE User Guide description of nodes (3.6.6 Node Definitions). In this example, the fault node “**fail\_to**” is located in a separate file called “**faults**.” A library of node definitions is provided in Appendix 1 Section 7.1 for convenience. This particular node definition is given in Figure 15.

****

Figure 15: Fault Node Definition

In the case of faults on a component with multiple outputs, the subclause will contain more than one spec statement; one for each of the outputs affected by a fault.

#### Input Statement

Input statements are where the parameters of the fault node definition are linked to expressions which assign the node parameters a value.

Notice the “**input**” keyword in Figure 14 . Not to be confused with the AADL subcomponent input, this refers to the *input to the fault node*. The fault node will wrap the AADL component output and hence that AADL output becomes input to the node definition. This is depicted in Figure 16.



Figure 16: Fault node wrapping output of AADL component

The list of inputs provided correspond with the number and types of arguments in the fault node definition. These inputs will be passed to the fault node call in the order received. Notice in Figure 15 the “fail\_to” node has three arguments: *val\_in*, *alt\_val*, and *trigger*. The first argument is required to be called *val\_in* and corresponds with the output of the AADL component that is being manipulated by the fault. In the fault subcomponent (input), you can see that the output of system A (“Output”) is being passed in as an argument to the fault node *val\_in* field.

**inputs:** val\_in **<-** Output**,** alt\_val **<-** **prev(**Output**,** 0**);**

For this particular fault node, *alt\_val* is the preferred fail-to value. Using a “prev” statement (see AGREE Users Guide for more details), the value of the output in the previous step is used as the fail-to value. Alternatively, users can enter an actual value (e.g. 0.0 or -1.0) or let *alt\_val* be completely nondeterministic. For an example of nondeterminism in a fail-to value, see section 5.4.1.8 Safety Equation Statements.

Record types in AADL are supported and their fields can be used in input and output statements.

**Note: The trigger value:** As can be seen in the fail\_to fault node definition (Figure 16), the final argument is a *trigger*. This trigger is NOT passed in by the user but is used behind the scenes as a triggering mechanism. There is no way at this time to manually specify a trigger without the use of dependent faults (hardware faults, Section 5.4.3).

#### Output Statement

Output statements will specify which output will be affected by the fault node output. Since components may have more than one output, and fault nodes may return a list of values, the fault node output must be linked in this way.

**outputs:** Output **<-** val\_out**;**

For the running example, we see that “*val\_out*” is the output of the fault node and “*Output*” is the output of the AADL component. This is the output that will be changed by the triggering of the fault. In the case of a fault node definition having more than one return value, the output statement would be organized into a list much like the example for input statements in section 3.5.1.

Record types in AADL are supported and their fields can be used in input and output statements.

#### Probability Statement

Currently the annex supports top level probabilistic analysis through the use of analysis statements. An analysis statement is given at the top level of the system implementation under analysis and will specify the type of analysis to perform. Probabilistic analysis is described in Section 5.4.2.2. In order to properly use the probabilistic analysis, there is a probability associated with each fault in the model. This is given as:

**probability:** 5.0E-5**;**

The probability of occurrence will depend on the fault, the AADL component, and the real probability of failure given hardware specification guidelines.

#### Duration Statement

A duration statement specifies that the fault is permanent. A permanent fault will remain indefinitely and has no such interval in the statement.

An example of a permanent fault is shown in the running example.

**duration: permanent;**

\*Transient faults are currently not supported in the safety annex. This will be implemented in future work. The only possible faults at this time are permanent.

#### Propagate-Type Statement

Users have the ability to define either asymmetric or symmetric faults. A symmetric fault affects the output of the component it is attached to and all ports connected to this output will see the same fault behavior. An example of this is with the running Toy Example. There is no “**propagate\_type**” statement and thus the default *symmetric* type is used.

An asymmetric fault can be applied in the case when an output fans out to multiple receivers. These receivers will see in the nominal case the same value being output from the sender component. At times it is beneficial to model a case when the fault affects this fan out output slightly differently for each receiving connection and thus they may see different values coming from the same source. These are also known as Byzantine faults. A graphical example of this is shown in Figure 17.



Figure 17: Sender-Receiver AADL Model

In this simple model, the top level system sends two global Boolean inputs to the Sender component. The Sender acts as an or gate for its behavior on the output. This output from the sender fans to two receivers. The contracts on the receivers simply state that input = output. These are passed to the top level system.

In this type of connection arrangement, it is expected in the nominal model that both receivers get the same input from the sender. When a normal symmetric fault is activated on the sender output, both receivers still get the same faulty value. Of course this is at times useful in a fault model, but what is also of use is the possibility that the receivers are actually seeing different values. In this case, an *asymmetric* fault is defined and the receivers may receive varied inputs.

The syntax of such a fault must contain a *“propagate\_type”* statement within the fault statement. These can be properly applied to the output of any component that has multiple receiving components. An example that corresponds to the sender-receiver model (Figure 17) is shown in Figure 18.



Figure 18: Fault Statement With Asymmetric Propagate Type

As seen in Figure 18, the rest of the fault definition is the same as in the symmetric case and the library of fault nodes can be used to define the faulty output.

For a simple example of this fault in action, see the “examples” section of our github repository under “Byzantine\_Example.” <https://github.com/loonwerks/AMASE/tree/develop/examples/Byzantine_Example>

Note: The fault value that can override the output of the sender component can only affect *less* than the total number of connections. The user cannot specify which connections are affected or how many. For example, if the sender component connects to receivers 1, 2, and 3 and this kind of fault is active, there may be 2 or less receivers that see the faulty value as input. If that sender asymmetric fault is active, there will be either 1 or 2 of the receivers that get the faulty value as input. Never 3 and never 0 (IF that fault is active).

#### Propagate-From Statement

Users can specify fault dependencies outside of fault statements, typically in the system implementation where the system configuration that causes the dependencies becomes clear (e.g., binding between SW and HW components, co-location of HW components). This is because fault propagations are typically tied to the way components are connected or bound together; this information may not be available when faults are being specified for individual components. Having fault propagations specified outside of a component’s fault statements also makes it easier to reuse the component in different systems. The *propagate-from* statement is used in conjunction with a hardware fault statement outlined in Section 5.4.3. An example of a fault dependency specification is shown in Figure 19, showing that the *valve\_failed* fault at the shutoff subcomponent triggers the *pressure\_fail­\_blue* fault at the selector subcomponent.

****

Figure 19: Propagation Statement Example

In terms of the fault analysis, this equates to the following. If the shutoff valve fails (“*valve\_failed@shutoff*”), the selector valve on the blue line also will fail – regardless of connection ports, probabilities, or other factors. The shutoff valve fault is a common cause occurrence for the selector valve failure. In any probabilistic analyses or max fault analyses, this selector valve fault comes for free, so to speak.

In order to use a propagate-from statement, a hardware fault statement must be present in each of the components (source and sink). See Section 5.4.3 for a full description of a hardware fault statement.

#### Safety Equation Statements

To allow flexibility in assigning failure values, various kinds of equation statements are defined for the Safety Annex. This extends the AGREE equation statement by adding three new kinds of equations.

##### Eq Statements

A Safety Equation Statement is identical to an AGREE Equation Statement. Equation statements can be used to create local variable declarations within the body of an AGREE subclause or within a Safety annex fault statement. An example of an equation statement is:

**eq** equation **:** **int** = 9**;**

In this example, we create an integer variable with the value of 9. Variables defined with equation statements can be thought of as ''intermediate'' variables or variables that are not meant to be visible in the architectural model (unlike component outputs or inputs). Equation statements can define variables explicitly by setting the equation equal to an expression immediately after it is defined. Equation statements can also define variables implicitly by not setting them equal to anything. This would capture complete nondeterminism for fault values. An example of this is:

**eq** equation **:** **int;**

To use an equation statement within a fault definition, the equation statement would be defined as above and then used in the input statement to link with the fault node. An example of passing an eq variable into a fault node is shown below and is specifically a nondeterministic eq statement. This fault definition corresponds to component A of the Toy Example but is changed slightly to allow for nondeterministic illustration.

**annex** safety {\*\*

**fault** stuck\_at\_fault\_A "Component A output stuck" **:** faults**.**fail\_to **{**

**eq** nondet\_val **:** **real;**

**inputs:** val\_in **<-** Output**,** alt\_val **<-** nondeterm\_val**;**

**outputs:** Output **<-** val\_out**;**

**probability:** 5.0E-5**;**

**duration:** **permanent;**

**}**

\*\*};

Equation statements can define more than one variable at once by writing them in a comma delimited list. One might do this to constrain a list of variables to the results of a node statement that has multiple return values or to more cleanly list a set of implicitly defined variables.

### Analysis Statement

An analysis statement is given in the system implementation under analysis. This is shown in Figure 20 and Figure 21. By referencing the Toy Example, you can see that the analysis statement is within the top level system implementation (Figure 2). This statement specifies the type of analysis to perform. Only one type is permitted to be specified for a single analysis run. There are two kinds of analysis that can be requested by the user. Maximum number of faults present in the system or a probabilistic analysis. These are described in detail in this section.

**Important Notes:**

1. **Analysis statements must be added to each layer of a system in order for analysis to proceed correctly. Every implementation must contain an analysis statement that applies to that layer.**
2. **At most one of the analysis statements (max n faults or probability) must be present.**
3. **The nominal model must prove in the absence of faults. If not, the fault analysis results cannot be trusted.**

#### Max N Faults Analysis

As shown in Figure 20, the user can specify the maximum number of active faults in a system. In this way, it can be determined if the system is resilient to a certain number of faults.

**annex** safety {\*\*

**analyze** **:** **max** 1 **fault**

\*\*};

Figure 20: Max One Fault Analysis Statement Example

The options for max N fault analysis:

1. The user can perform monolithic max N fault analysis. This flattens the model and attempts to prove that with at most N faults active, the top level properties prove. If not, the model checker returns a counterexample with one of the combinations of N faults.
2. The user can perform compositional max N fault analysis. Each of the analysis statements in each layer is used during the proof to see if max N faults are present *in a given layer*, do the properties *for that layer* prove. A counterexample may be returned that shows that the top level properties prove, but something lower in the model does not. That does not necessarily mean that the top level properties are unaffected by the faults, it just means that the analysis was run compositionally. If those violated properties are part of the set of support for the top level properties, then they also would be violated. It is suggested that this is used to gather information on subsystems and see problematic subcomponents of a system.
3. The user can perform compositional max N fault analysis to collect all minimal cut sets up to cardinality N. In this case, lower level faults and problematic subcomponents will be seen in light of the top level properties.

In all of these cases, it is assumed that the nominal model proves. If it does not, the results from the fault analysis will be skewed (or return as errors).

#### Probabilistic Analysis

In order for the probabilistic analysis to run, probabilities must be assigned to each fault definition as shown in the Toy Example of Figure 2. The syntax of probabilistic analysis is shown in Figure 21 with a top level threshold of 1.0E-7.

**annex** safety {\*\*

**analyze** **:** **probability** 1.0E-7

\*\*};

Figure 21: Probability Threshold Example

There are two options for probabilistic analysis.

1. The user can run monolithic analysis on the model with the given probability threshold. The possible fault combinations (assuming the top level probability threshold and independence between faults) are calculated. If these possible faults are active and the top level properties do not hold, then the model checker returns a counterexample for the user with ONE of these possible fault combinations.
2. The user can run compositional analysis on the model with the given probability threshold. In this case, ALL possible combinations that cause violation of the top level property withinthis threshold are provided to the user.

In all of these cases, it is assumed that the nominal model proves. If it does not, the results from the fault analysis will be skewed (or return as errors).

For more information on running the various types of analysis, we refer readers to Section 7. This section details specific examples (available in the GitHub repository) and goes through the steps for all forms of analysis and how to read the results.

### Fault Activation Statement

A fault activation statement links an eq statement in the AGREE annex with the activity of a specific fault definition. This allows users to reason over active faults. The eq statement must be in an AGREE annex of the implementation which holds a Safety Annex defining the fault activation statement. The eq statement must be Boolean nondeterministic.

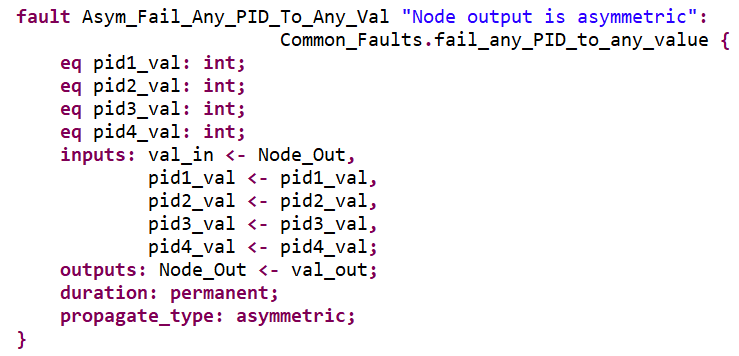


The fault activation statement is located in the Safety Annex of this implementation.



It refers to the name given to the fault and the name of the instantiated component with this fault defined.

The fault statement is located in the Safety Annex of the Node interface. The point of interest to note is the name given to the fault (Asym\_Fail\_Any\_PID\_To\_Any\_Val) matches the fault activation statement.



The component “node1” is an instantiation of an abstract system type and thus these fault activation statements must reside in the implementation Safety Annex.

The value assigned to the eq statement is true if the fault is activated and false otherwise. Within the AGREE annex that holds these eq statements, they can be used within contracts. A good example of this can be found in the PID example of Section 6.2.4.

### Hardware Fault Statement

Failures in hardware (HW) components can trigger behavioral faults in the software (SW) or system (SYS) components that depend on them. For example, a CPU failure may trigger faulty behavior in threads bound to that CPU. In addition, a failure in one HW component may trigger failures in other HW components located nearby, such as cascading failure caused by a fire or water damage.

Faults propagate in AGREE as part of a system’s nominal behavior. This means that any propagation in the HW portion of an AADL model would have to be artificially modeled using data ports and AGREE behaviors in SW. This is less than ideal as there may not be concrete behaviors associated with HW components. In other words, faulty behaviors mainly manifest themselves on the SW/SYS components that depend on the hardware components.

To better model faults at the system level dependent on HW failures, we have introduced a fault model element for HW components. In comparison to the basic fault statement, users are not specifying behavioral effects for the HW failures, nor data ports to apply the failure. An example of a model component fault declaration is shown in Figure 22. This example is taken from the Wheel Brake System model (WBS\_arch4\_v2) found in <https://github.com/loonwerks/AMASE/tree/develop/examples>. The grammar for a hardware fault subcomponent is shown below.

HWFaultSubcomponent: 'duration' ':' TemporalConstraint (Interval)? ';'

| 'probability' ':' Real\_Literal ';'

| 'propagate\_type' ':' PropagationTypeConstraint ';'

The example shows the failure of a hardware valve component.



Figure 22: Hardware Fault Statement

Users must specify fault dependencies outside of fault statements using propagation statements. For more information on propagation type statements, see section 5.4.1.5.

To summarize, the fault statement (Figure 22) belongs in the source and sink components and defines the type of fault that occurs. The propagate-from statement (Section 5.4.1.6) is located in the implementation with the hypothesis statement. This links these dependent faults and uses their information throughout the analysis.

#### Duration, Probability, and Propagation-Type Statements

For descriptions of these, please refer to sections 5.4.1.3, 5.4.1.4, and 5.4.1.5. The syntax and explanations are identical in both independent and dependent faults.

# Library and Custom Made Fault Nodes

A library of fault nodes is available through the GitHub repository (LINK). These include commonly used fault definitions for integer, real, and Boolean types such as *inverted­\_fail* for Boolean and *stuck\_at\_zero* for integer and real. A description of the fault nodes is provided here for your convenience and more information can be found in the AGREE Users Guide Section 3.6.6 Node Definitions.

To access and use this library of commonly used fault node definitions, download the Fault\_Library.aadl library file from the directory listed above and place in your Osate project directory. This can be referred to within said project.

A basic fault node is shown below:



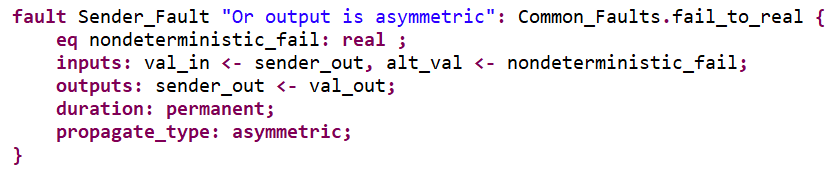
The node name is “fail\_to\_zero” and the parameters are *val\_in* and *trigger* with types int and bool respectively. For the fault node definitions, the last parameter must always be *trigger* and is ONLY used internally. The user cannot pass a trigger into this node call without an exception being thrown.

Between the “let” and “tel” keywords are the behavioral descriptions of the fault model. This defines how the output of the AADL component (val\_in parameter) will fail. In this case, if the trigger is active, the output of this fault node is zero. Otherwise it remains the same, i.e. the AADL component output is unchanged.

Many of the commonly used fault node definitions may be helpful, but in some cases it is necessary for the user to define fault models specific to a domain or model.

## Nondeterministic Failure Values

To capture nondeterministic fail-to values, an eq statement is defined in the Safety Annex and left unassigned.



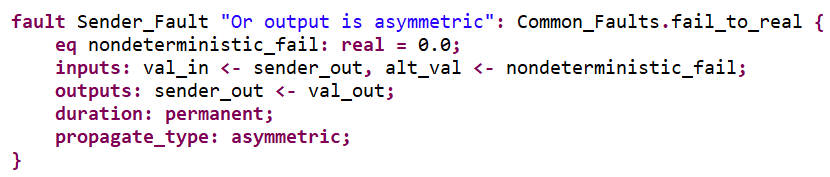
This unassigned eq variable is passed in as a parameter to a fault node. This is seen in the

*alt\_val <- nondeterministic\_fail* statement in the fault definition above. This fault statement refers to the fault node *fail\_to\_real* which is provided below.



This value can then be assigned anything during analysis, which means that if a value will cause a property to fail, the model checker will return that value in a counterexample.

Notice that this is almost the same if a determined fail-to value is desired. The node call is the same, but the eq statement will have a value assigned:



To restrict the range of nondeterminism, the eq statement can be defined in AGREE (using the same syntax as with the Safety Annex eq statements) and through the use of assert statements, this range can be set. For information on assert statements, see AGREE Users Guide Section 3.6.9.

## Nested Data Structures

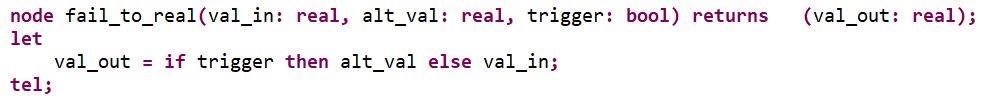
There are two ways that a fault node can be used with a nested datatype of a components output. One way is to pass into the fault node the field corresponding to the primitive type matching the fault node parameter type. The other way is to pass in the entire data structure and define within the fault node which field(s) change when the fault is active.

The first option is quite simple. This running example is found in the GitHub repository example directory under ByzantineExampleNestedTypes (LINK). The datatype is called commBus.impl and has one field, NODE\_VAL, which is of primitive type real.

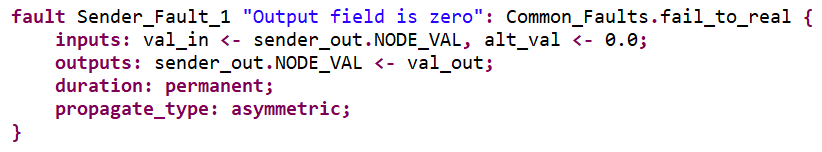


A particular component (Sender.aadl) has an output of type commBus.impl and we wish to model a fail to zero value on the NODE\_VAL field.

The fault node can be defined as the following:



And the fault definition in Sender.aadl Safety Annex can be written as:

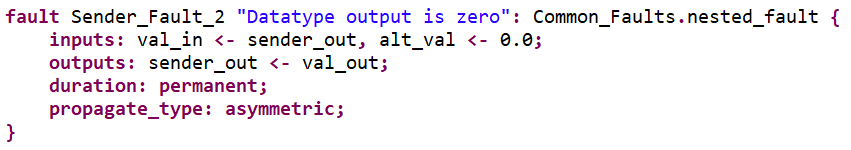


Thus, the field that may fail is passed directly into the fault node.

The second way to perform the same action provides the user with a richer way to create the fault model within the node definition body. In this case, the entire data structure is passed into the fault node and the failure behavior references the specific fields of the structure. The node is defined as follows.



The syntax for resetting the data structure fields is shown in the node definition above. The Sender fault in the Safety Annex for said component is:



Slightly out of context of this simple example is another node definition that can be found in the example directory (This example is titled PIDByzantineAgreement\_v2 and a detailed description of the project is given in Section 7.2.4). The reason this node definition is included here is to show that multiple fields can be altered in a fault, or even given certain conditions, different fields may be altered.



# In Depth Examples

The Toy Example was sufficient to illustrate the syntax and other grammatical elements, but more in-depth examples will be used to show how more complex fault models can be built and how to run and view the analysis results. The first example is a sensor system which utilizes symmetric faults in the fault model. The second example shows a more complex Byzantine problem and its corresponding mitigation. Both of these examples are found in the Github repository.

[links]

## The Sensor Example

### AADL Architecture

In order to exercise the tool and test results, we created a reactor system example. The Reactor System contains three subsystems: Temperature Sensors, Pressure Sensors, and Radiation Sensors. Each of these subsystems contain three sensors and commands shutdown of the system when readings are out of the normal range (i.e. high temp, high pressure, high radiation). The top level system property is that the system shall shut down if and only if there are high readings on any of the sensors. The AADL architecture diagram is shown in Figure 23.



Figure 23: Sensor Example Architecture

There are majority voting subcomponents that perform a check on all sensor outputs. Thus if one pressure sensor reports high pressure, but the other two are in normal range, the system will output normal. This voting behavior mitigates the situation in which one fault is present in a sensor causing it to report erroneous values. Thus adding a voter makes the system resilient up to one sensor fault in each pressure and temperature subsystem.

Note: In the AADL example, there are two implementations of the Reactor System; one with voting and one without. The remainder of this example description deals with the voting case, but it is easy to run the analysis on the non-voting version and compare results.

### AGREE Behavioral Model

The contracts on the sensors state that when pressure (or temp/radiation) is above threshold, output a shutdown command. The AADL and AGREE code describing a pressure sensor is shown in Figure 24.



Figure 24: AADL and AGREE Sensor Subcomponent

The higher level subsystem contracts (Pressure Reactor, Temperature Reactor, and Radiation Reactor) verify that the shutdown command is only issued when there are actually high values in the environment. The top level Reactor System checks the actual environmental inputs against the shutdown commands from all subsystems and states that we shut down only when it is necessary.



Figure 25: Top Level Reactor System and Top Level Property

### Safety Model

Each sensor can have a failure of getting stuck at indicating high or low. This equates to two possible faults per sensor. A stuck low fault for the pressure sensor is shown in Figure 26.



Figure 26: Fault Definition for Pressure Sensor

Given the spec statements for the type of sensor used, the probability is determined for this specific kind of sensor failure. For this example, we chose . When a fault occurs, it is assumed to be permanent. The underlying fault model (*CommonFaults.stuck\_false*) is shown in Figure 27.



Figure 27: Fault Node Definition for Sensor Fault

For more information on creating fault node definitions, see Section [CREATE THIS SECTION, DANIELLE].

### The 4 types of Analysis Results

Section 5.4.2 discusses the types of analysis that can be run in a given model and this section outlines the results from these analysis runs on the Reactor System. **For accurate fault analysis results, the nominal system model must verify for both monolithic and compositional for all layers.**

Recall that the Sensor System is a multi-level system. The organization is as follows:



Figure 28: Subcomponent Organization of Sensor System

The top level (Reactor System) has three types of subcomponents (pressure, temperature, and radiation reactors). These three have two types of subcomponents (sensor and voter). In running any type of analysis, one must have analysis statements (see Section 5.4.2) in each implementation. Thus not only will we define analysis statements in the Reactor System (topmost level), but also the implementations in lower levels (pressure, temperature, and radiation reactor subsystems). For each layer of the model, the number of faults or probability threshold is restricted.

#### Verify All Layers with Max N Faults present

Instructions for run:

* Put max *n* analysis statement in ALL system implementations (Reactor\_Sys\_Ctrl.W\_Voting, Reactor\_Pressure\_Ctrl.W\_Voting, Reactor\_Temp\_Ctrl.W\_Voting, Reactor\_Radiation\_Ctrl.W\_Voting). Make sure probabilistic statements are commented out.
* Select the implementation in AADL.
* Select in menu: AGREE -> Verify in the presence of faults.
* Select in menu: AGREE -> Verify all layers (or Verify single layer)
* Results and/or counterexample will be displayed.

Recall that when max N fault analysis is selected with *Verify all Layers*, the compositional approach verifies from the top down. Thus the top level analysis statement only applies to faults within the top level. In this example, there are none. All faults are located on leaf level components one layer removed from the top. Thus, each analysis statement in the implementations must constrain the number of faults active *at this level*.

This can be seen by setting the number of active faults at the top level (Reactor\_Sys.aadl -> Reactor\_Ctrl.W\_Voting) to be *analyze: max 1 faults*. Then at the lower level subsystem implementations (Reactor\_Temp.aadl -> Reactor\_Temp\_Ctrl.W\_Voting or Reactor\_Pressure.aadl -> Reactor\_Pressure\_Ctrl.W\_Voting), set the analysis constraint to 2 faults. It can be seen in the analysis results that the top level property proves while the lower level properties are failing. The results are shown in Figure 29. Upon examination of the counterexamples, it is seen that 2 faults are active in these layers which cause the violation of that layer’s property. The counterexample in Eclipse view is shown in Figure 30. This counterexample shows that two faults are active in the temperature system (Reactor\_Temp\_Ctrl.W\_Voting). The analysis statement in this level corresponds with this result.

This result is intuitive given the voting implementation. If the majority of the sensors show an erroneous high (or low) reading at the same time, the shutdown command would be sent erroneously.



Figure 29: Verification Results for Max 2 Faults on Sensor Example



Figure 30: Counterexample from Sensor Analysis with 2 Faults Active

The third subsystem, Reactor\_Radiation.aadl, uses a different variation of a voting mechanism. All three sensors must agree in order for the command to be valid. Thus we can expect that in this subsystem, one fault will cause a violation of the property *Shut down when and only when we should*.

By entering a fault number constraint at this level (Reactor\_Radiation\_Ctrl.W\_Voting) of max 1 fault and running the analysis with no other faults allowed in the other two subsystems, it can be seen that the property is violated with one active *stuck\_low* fault.

**Note:** It is important to realize that in these analysis runs, it is shown that the safety property at the top level proves in the presence of faults according to the results shown in the pane. This is not entirely true and should not be interpreted as such. ***If the lower level guarantees are failing due to faults and these guarantees are used in the proof of the top level safety property, then this safety property will fail in the presence of these faults****.* In order to see a more comprehensive analysis with respect to the top level safety property, user should also perform *Generate Minimal Cut Sets* analysis. This shows which faults can cause violation of the safety property at the top level.

#### Verify All Layers with Faults Present: Probabilistic Analysis

Instructions for run:

* Put probabilistic statements in top system implementation. (Reactor\_Sys\_Ctrl.W\_Voting) Make sure max *n* statement is commented out.
* Select the implementation in AADL.
* Select in menu: AGREE -> Verify in the presence of faults.
* Select in menu: AGREE -> Verify Monolithically
* Results and/or counterexample will be displayed.

Note: **Probabilistic analysis cannot be run with Verify all layers.** In order to run compositional probabilistic analysis, one must refer to Section 7.1.4.4.

Since the analysis proceeds monolithically, the probability threshold given at the top level is used for the overall system threshold. If there is a combination of faults that exceed the given threshold, the counterexample with show *one* such combination.

In the Sensor example, each fault is assumed to occur with probability and the top level threshold is . Upon running monolithic analysis, a top level property fails.



The counterexample reveals one of the fault combinations that exceed the threshold. If two of the temperature sensors are stuck at low, then the top level property cannot be proven.



If the user wishes to see all such combinations, refer to Section 7.1.4.4.

#### Generate Minimal Cut Sets with Max N Faults

Instructions for run:

* Put max *n* analysis statement in top level system implementation and make *n* equal to any integer you wish for the max cardinality of the minimal cut sets generated. This statement goes in the Safety Annex located in Reactor\_Sys\_Ctrl.W\_Voting. Make sure probability analysis statement is commented out.
* Select the implementation in AADL.
* Select in menu: AGREE -> Generate Minimal Cut Sets.
* File with printed MinCutSets will be displayed.

The only system implementation analysis statement that is used in this analysis is at the top level (Reactor\_Ctrl.W\_Voting).

The number of faults constraint applies to the cardinality of the minimal cut sets. Thus if we analyze with 1 fault, this restricts the cardinality of the cut set to 1. Likewise, *n* faults restrict the cardinality of the cut sets to be less than or equal to *n*.

Cardinality 1 produces 3 MinCutSets, one for each of the radiation sensors stuck low. Given that the radiation system utilizes a voter that requires all three sensors to agree, these results are easy to see.

Cardinality 2 produces 15 MinCutSets. These are:

3 Cut Sets: Each radiation sensor stuck low

3 Cut Sets: Combinations of temp sensors stuck low (sensors 1,2; sensors 1,3; sensors 2,3)

3 Cut Sets: Combinations of pressure sensors stuck low (sensors 1,2; sensors 1,3; sensors 2,3)

3 Cut Sets: Combinations of temp sensors stuck high (sensors 1,2; sensors 1,3; sensors 2,3)

3 Cut Sets: Combinations of pressure sensors stuck high (sensors 1,2; sensors 1,3; sensors 2,3)

Cardinality 3 produces 16 MinCutSets. These include all cut sets of cardinality 1 and 2 (15 sets total) and one additional for all three radiation sensors stuck high.

Setting the *n* value higher than 3 produces no more cut sets than the 16 shown in *n = 3* analysis.

We provide a snippet of the text file generated for this analysis to explain to the reader its contents.

Minimal Cut Sets for property violation:

property lustre name: safety\_\_\_GUARANTEE0

property description: Shut down when and only when we should

Total 15 Minimal Cut Sets

Minimal Cut Set # 1

Cardinality 2

[

Fault # 1

original fault name, description: pressure\_sensor\_stuck\_at\_low, "Pressure sensor stuck low"

lustre component, fault name: Reactor\_Pressure\_Ctrl, reactor\_Pressure\_Ctrl\_fault\_\_independently\_\_active\_\_Pressure\_Sensor2\_\_Pressure\_Sensor2\_\_fault\_1

failure rate, default exposure time: 1.0E-5, 1.0

Fault # 2

original fault name, description: pressure\_sensor\_stuck\_at\_low, "Pressure sensor stuck low"

lustre component, fault name: Reactor\_Pressure\_Ctrl, reactor\_Pressure\_Ctrl\_fault\_\_independently\_\_active\_\_Pressure\_Sensor1\_\_Pressure\_Sensor1\_\_fault\_1

failure rate, default exposure time: 1.0E-5, 1.0

];

At the top of the file, the total number of cut sets are displayed (15 in this case). The min cut sets are then displayed textually. Within the square brackets contain all faults within a given cut set. Fault #1 corresponds to a pressure sensor 2 stuck low, fault #2 corresponds to pressure sensor 1 stuck low. For each property violation, a group of minimal cut sets are described. These are the cut sets for that particular property.

#### Generate Minimal Cut Sets with Probability Threshold

Instructions for run:

* Put probabilistic analysis statement in top level system implementation (safety annex located in Reactor\_Sys\_Ctrl.W\_Voting). Make sure max *n* statement is commented out.
* Select the implementation in AADL.
* Select in menu: AGREE -> Generate Minimal Cut Sets.
* File with printed MinCutSets will be displayed.

When running probabilistic analysis through compositional reasoning, we expect to have only the minimal cut sets whose combined probability exceeds the top level threshold. Thus, any combination of the 16 total minimal cut sets will be shown depending on the faults’ associated probabilities.

Given that each sensor could fail with probability and the top level probability threshold set at , there are a total of 15 minimal cut sets. These correspond to stuck low failures of each radiation sensor (3 total), each combination of the temp sensors stuck and high and stuck at low (6 total) and likewise the combinations for the pressure sensors being stuck both high and low (6 total).

## Byzantine Examples

A Byzantine or asymmetric fault is a fault presenting different symptoms to different observers. In an asymmetric fault definition, a fault is tied to an AADL component that has a fanned output to multiple components. In this section, we show two examples and the mitigation strategy implemented in AGREE.

### Asymmetric Fault Implementation

Figure 31 explains our approach to the implementation of Byzantine faults. In this figure, component A has an output, *out1*, that fans out to 4 components (B, C, D, and E). Each of those components has an input. (Note: in the architectural model, the circular “CN” nodes are not present. It is simply a connection from component A to the other 4 components.)



Figure 31: Asymmetric Fault Implementation Strategy

An asymmetric fault is defined for Component A as follows:

**fault** asymmetric\_fault\_Comp\_A "Component A output asymmetric" **:** faults**.**fail\_to **{**

**inputs:** val\_in **<-** Output**,** alt\_val **<-** **prev(**Output**,** 0**);**

**outputs:** Output **<-** val\_out**;**

**probability:** 5.0E-5**;**

**duration:** **permanent;**

**propagate\_type:** **asymmetric;**

**}**

This fault defines an asymmetric failure on Component A that when active, is stuck at a previous value (**prev(**Output**,** 0**)**). This can be interpreted as some connected components may only see the previous value of Comp A output and others may see the correct value *when* the fault is active.

This fault definition is injected into the communication nodes and which connected components see a failure value is completely nondeterministic. Any number of the communication node faults (0…all) may be active upon activation of the main asymmetric fault.

In order to mitigate these types of faults, special attention must be paid to the behavioral model. This example was developed in order to show how to use the Safety Annex in this situation as well as the mitigation through AGREE contracts.

### Mitigation Strategy

Since Byzantine faults may present different symptoms (or message values) to different observers. The objective of the agreement protocol is for all correct (non-failed) nodes to eventually reach agreement on a value sent by another node. There are *n* nodes, possibly *m* failed nodes. The protocol requires *n > 3m* nodes to handle a single fault. The point is to achieve distributed agreement and coordinated decisions.

The properties that must be shown given this protocol are as follows:

* All correct (non-failed) nodes eventually reach a decision regarding the value they have been given. In this solution, nodes will agree in *m+1* time steps or rounds of communication.
* All correct nodes agree on the same value (distributed agreement).
* If the source node is a correct, all other correct nodes agree on the value that was originally sent by the source. If the source node is failed, all other nodes must agree on some (somewhat arbitrary) value.

### Color Exchange Example

As a simpler lead in to the main Process ID example, we created a Color Exchange example. This was to view both the asymmetric implementation and the mitigation strategy from a simpler point of view before extending it to a larger model.

First we will describe the smaller example and then the PID example and mitigation will hopefully be clearer.

#### Color Exchange Architecture in AADL

A simple diagram of the color exchange architecture is shown in Figure 32. It consists of a leader node that sends a message (a color) to three other nodes. The “color” is represented in AADL as a Boolean value (true = green, false = blue). This is a single output that fans to the receiving nodes.

Each node reports to the top level what color they have seen. In the nominal case, every receiving node will see the same value, since it is a single fanned output.

Without any mitigation strategy, it is clear that an asymmetric fault will cause the receiving nodes to report different colors to the top level. Thus any contracts stating that they see the same thing will be violated if one fault is active.



Figure 32: Color Exchange Architecture for Asymmetric Modeling and Mitigation

This model does not include any mitigation and can be explored by accessing the GitHub repository:

[LINK]

#### Color Exchange Mitigation Strategy

The mitigation strategy will allow a single asymmetric fault to be present (on leader node) and all receiving nodes will agree on a color value at the end of two communication rounds.



Notice that in this strategy, if two of the receiving nodes were to receive the erroneous value “blue,” that the majority vote would have them all agree on blue as the color. This is a faulty value, but as long as all nodes respond the same, it is following the mitigation strategy. (Recall the goal: If the source node is failed, all other nodes must agree on some (somewhat arbitrary) value.)

The mitigation in AGREE is shown in the following contracts. The leader node behavior is such that it’s color in the nominal case is always green (true). This occurs from the first step onward.



The receiving nodes have additional ports between them in order to share what they received in the first step. These are all fanned outputs, thus can have an asymmetric fault present. There are a total of 4 possible asymmetric faults and our mitigation strategy can show resilience up to 1 active asymmetric fault.

These contracts define the behavior over those ports (output what they saw from the leader in the previous step to the other receiving nodes: guarantee 1) and the behavior of what they report to the top level (majority voting on what everyone saw: guarantee 2).



The contracts at the top level cover both cases.

* Case 1: If the source node is a correct, all other correct nodes agree on the value that was originally sent by the source.
* Case 2: If the source node is failed, all other nodes must agree on some (somewhat arbitrary) value.

These correspond to two guarantees.



The first guarantee covers case 1 and will be violated in the presence of any asymmetric fault. This is expected.

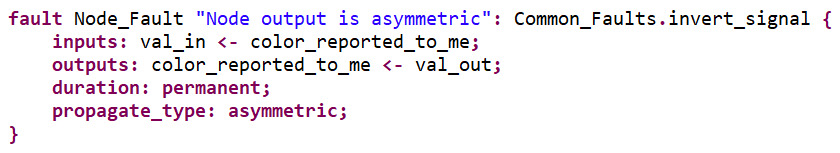
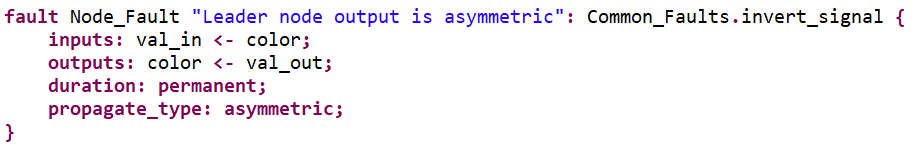


The second guarantee can access which node has an active fault and thus guarantee that the non-failed nodes will agree amongst themselves on a color.

#### Color Exchange Fault Model and Analysis

In the nominal case, all properties are verified. This can be seen by running *Verify all layers* or *Verify single layer* in the AGREE analysis menu.

The faults are defined on every node, the leader and receivers.



Verification Results:

* Nominal model: Both top level guarantees are verified. All nodes output the correct value and all agree.
* Fault model with one active fault: The first guarantee (when no fault is present, all 4 nodes agree) fails. This is expected when faults are present. The second guarantee (all non-failed nodes agree) is verified with one active fault.
* Fault model with two active faults: Both guarantees fail. This is expected since in order to be resilient up to two active faults *f*, we would need 3*f* + 1 = 7 nodes.

This model is in Github and is called ColorByzantineAgreement. [LINK]

### Process ID Example

We now wish to extend the concept of “leader node” to all nodes. In this example, the Boolean type of the first small model is extended to integers and each node passes its own process id (PID) to the other three nodes. At the end of the algorithm, the nodes choose the node with the max PID to be “leader.”

Note: This model was built in 6 increments and thus has v1 – v6 in the packages. To view and run the complete model, go to Top\_Level\_v6.aadl. Version 6 is the model described in the upcoming sections.

#### PID Example AADL Architecture

The basic architecture is shown in Figure 33. In the first time step, all nodes send their PIDs to each other. In the second time step, each node reports to all the others what it saw in the previous step. At that point, all nodes reach an agreement regarding what they saw based on what everyone reported using a majority voting protocol. The max PID node is chosen as the leader.



Figure 33: PID Example Architecture

The output of the nodes (Node\_Out) is a data port that passes a Node\_Msg data structure. This structure has 4 fields corresponding to a specific node and its PID message.



Likewise, the inputs of a node (Nodei\_In) carry this same message.

Each node has a PID and a number. These are encoded in AADL as Properties.



The contracts rely on these properties to determine which node performs in which way. This is described in the next section on the AGREE behavioral model.

#### PID Example AGREE Behavioral Model with Mitigation Strategy

Similar to the color example, the nodes pass their PID (which is an AADL Property value) from the first step onward. Since a node is abstract and the PID applies to specific nodes, the PID must be accessed for this instantiation and contracts sent specific to that node. We show one of the four contracts that support this behavior.



Thus, any of the instantiated nodes will respond by either passing its own PID (if node 1) or passing the previously seen value *from* node 1.

The top level contracts mirror the color example. There are two varieties: one which is expected to fail when a fault is active and one that should not. In the nominal model, all nodes agree on each nodes PID value, thus there are 4 contracts to this affect, one for each node PID.



The second variety of contract at the top level is:



If node i failed, then all other nodes agree on the PID of node i. Notice the use of fault activation statements in order to reference a faulty component. For more information on fault activation statements, see Section 5.4.3.

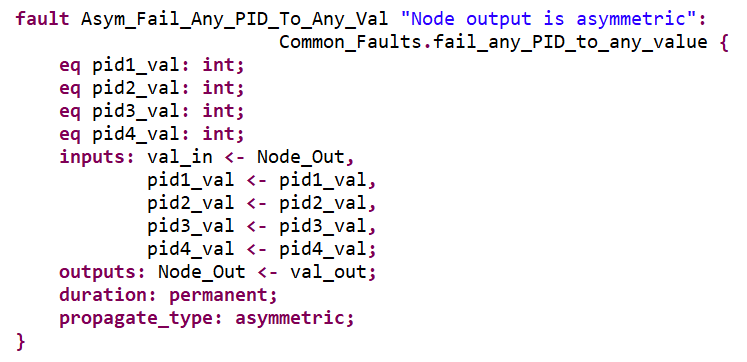
#### PID Example Fault Model

Since each node has multiple fanned outputs, a fault is defined for each one. To illustrate this, look at one node and its outputs.



Figure 34: Outputs for Node in PID Example

An asymmetric fault is defined for each of these outputs as follows:



And the fault node definition (fail\_any\_pid\_to\_any\_value) is:



This node definition takes the nominal output as an argument (val\_in) and four failure values (pid1\_val, … , pid4\_val). As seen in the fault statement, the pidi\_val values are nondeterministic eq statements. Thus setting these equal to the val\_in fields creates a nondeterministic failure value for those fields.

If a fault is active on a node, any one of those fields can be failed to any value.

#### PID Example Analysis Results

Verification can be run using max n fault and *Verify in the presence of faults* or by *Generating Minimal Cut Sets* with cardinality less than or equal to n. In either case, the safety annex analysis statement in Top\_Level\_v6.aadl -> Top\_Level.impl will need to have max n faults.

Verification Results:

* Nominal model: All top level guarantees are verified. All nodes output the correct value and all agree.
* Fault model with one active fault: The first four guarantee (when no fault is present, all nodes agree) fail. This is expected when faults are present. The last four guarantees (all non-failed nodes agree) is verified with one active fault.
* Fault model with two active faults: All 8 guarantees fail. This is expected since in order to be resilient up to two active faults *f*, we would need 3*f* + 1 = 7 nodes.

This model is in Github and is called PIDByzantineAgreement\_v2. [LINK]

# The Tool Suite (Safety Annex, AGREE, AADL)

In this chapter we present an overview of the Safety Annex/AGREE/OSATE tool suite, followed by installation instructions for the tool suite, and a description of the main features of the tool suite.

## Tool Suite Overview

Figure 23 shows an overview of the AGREE/OSATE tool suite. As presented in the figure, OSATE is an Eclipse plugin that serves as the IDE for creating AADL models. Both AGREE and the Safety Annex run as plugins in OSATE. OSATE provides both a language (AADL annex to annotate the models with assume-guarantee behavioral contracts in the case of AGREE and an AADL annex to annotate the model with faults in the case of the safety annex) and a tool (for compositional verification of the contracts reside in AADL models). AGREE translates an AADL model and its contract annotations into Lustre and then queries the JKind model checker to perform the verification. JKind invokes a backend Satisfiability Modulo Theories (SMT) solver (e.g., Yices or Z3) to validate if the guarantees are valid in the compositional setting. The safety annex uses an extension point in AGREE to access the AGREE program and insert the faults into the AGREE contracts. Then that program is translated into Lustre and the JKind model checker is queried to perform the verification/safety analysis.



Figure 35: Overview of Safety Annex/AGREE/OSATE Tool Suite

## Installation

Installing the Safety Annex/AGREE/OSATE Tool Suite consists of 4 main steps, described in each of the following sections.

### Install OSATE

Binary releases of the OSATE tool suite for different platforms are available at: [http://www.aadl.info/aadl/osate/stable/.](http://www.aadl.info/aadl/osate/stable/) Choose the most recent version of OSATE that is appropriate for your platform. For example, at the time of writing this document, the most current release of OSATE is 2.3.6, available for download from <https://osate-build.sei.cmu.edu/download/osate/stable/2.3.6/products/>.

After following the OSATE download instructions found on the OSATE download site (above). The splash screen shown in Figure 24 should appear, and OSATE should begin loading.



Figure 36: OSATE Loading Screen

If OSATE loads successfully, continue to the next step in the installation process. If not, and you are running Windows, the most likely culprit involves mismatches between the 32-bit and 64-bit version of OSATE and the bit-level of the Windows OS. Please check to see whether the version of OSATE matches the bit-level of your version of Windows OS. If running Windows 10, this information can be found in the System Control Panel as shown below in Figure 26. Note that this information is also required for downloading the correct version of the SMT Solver in the next installation step.

**Note: In OSATE release 2.3.6, AGREE is not up to date. Please perform this next step in order to reinstall AGREE in this release.**

### Install Safety Annex

To install Safety Annex with OSATE 2.3.6, first uninstall the AGREE that comes with OSATE 2.3.6 installation, by clicking “Help” -> “About OSATE2” -> “Installation Details”, and select “Agree” from the list of installed software, and click “Uninstall…”. In the Uninstall Details window, confirm to uninstall “Agree” by clicking “Finish”, and click “No” when it prompts to restart OSATE. Then in OSATE, click “Help” menu and select “Install New Software…”

In the Install window, place the following update site link for Safety Annex to the “Work with” field, and hit the enter key:

<https://raw.githubusercontent.com/loonwerks/AMASE/master/safety-update-site/site.xml>

Select both “Agree” and “Safety Annex” in the list of available tools to install, and click “Next”. Accept the terms of license agreement for the tools, and click “Finish”, and click “OK” on the Security Warning window about unsigned content, then click “Yes” to restart OSATE.

To test whether the safety annex has been correctly installed, a Safety Annex menu should appear in OSATE as shown in Figure 25.



Figure 37: Safety Analysis Menu Item



Figure 38: Windows 10 System Control Panel

### Install SMT Solver

Either one of the following SMT solvers can be used as the underlying symbolic solver invoked by the JKind model checker: Yices from SRI, or Z3 from Microsoft, Inc.

To download Yices, navigate to the Yices install page at: <http://yices.csl.sri.com/> and download the version of Yices appropriate for your platform.

To download Z3, navigate to the z3 install page at: <https://github.com/Z3Prover/z3/releases> and download the version of Z3 appropriate for your platform.

Either tool must be unzipped and placed in a directory somewhere in the file system. Then this directory must be added to the system path. For directions on how to add directories to your path, please see <http://stackoverflow.com/questions/14637979/how-to-permanently-set-path-on-linux> for Linux, and see <http://architectryan.com/2012/10/02/add-to-the-path-on-mac-os-x-mountain-lion/#.VszAv_krJph> for Mac OS. In Linux, you must add the path to your config file, usually .bashrc.

To add directories to your system path in Windows, first navigate to the System Control Panel and choose the "Advanced system settings" button on the left side of the panel. The system properties dialog will appear. Choose the "Advanced" tab in the dialog as shown in Figure 27 then click "Environment variables".



Figure 39: System Properties Dialog Box

The environment variables dialog box is shown in Figure 28.



Figure 40: Environment Variables Dialog Box

In order to make the application available to all user accounts choose the PATH environment variable in the "System variables" section and click "Edit…". This will bring up a text edit box, as seen in Figure 29. If the existing path string in the text edit box does not end with a semicolon (‘;’), add a semicolon first, then append the path to the SMT solver’s "bin" directory, and click "OK" on the dialogs. The bin directory for the Yices tool is underneath the main Yices directory, e.g., C:\Apps\ yices-2.4.2-x86\_64-pc-mingw32-static-gmp\yices-2.4.2\bin. The bin directory for the Z3 tool is underneath the main z3 directory, e.g., C:\Apps\z3-4.4.1-x64-win\z3-4.4.1-x64-win\bin.



Figure 41: System Variable Text Edit Box

To test whether Yices has been correctly installed on either Windows or Linux, open up a command prompt window and type: yices --version. A version number for Yices matching the installed version should be displayed.

To test whether z3 has been correctly installed on either Windows or Linux, open up a command prompt window and type: z3 -version. A version number for Z3 matching the installed version should be displayed.

### Set AGREE Analysis Preferences

Use the SMT solver of your choice (Yices or Z3) and set the AGREE Analysis preferences as shown in Figure 30.

Window -> Preferences -> AGREE -> Analysis



Figure 42: AGREE Analysis Preferences

At this point, you are ready to import the Toy Example project (see Section 4.1) and begin your own safety analysis.

## Development Environment Installation

An alternate installation guide is provided here. In these installation directions, the OSATE Development Environment is installed and the Safety Annex is compiled from source code and not through the update site.

### Install OSATE Development Environment

Follow the directions for installing the OSATE Development Environment provided on the following website:

<http://osate.org/setup-development.html>

In *Step 3: Select the Eclipse Platform* of the directions provided in the OSATE website, select the the 2018-09 release of Photon

In *Step 5:* *Set Required Variables* of the directions provided in the OSATE website, make sure that in all Github repositories the *HTTPS (read-only, anonymou*s) option is selected.

Notes and Possible Issues:

* Step 5 looks slightly different since the last OSATE update. There are fewer Github repositories in this list.
* In many installations, OSATE will display a message stating that it cannot perform the required operation. It will then attempt to make the correct installation. This takes time, but it does perform the desired operation.
* It’s recommended to start the installation from empty, new folders (e.g., for git checkout, workspace, and OSATE installation) to avoid problems with installation.

### Download Safety Annex Source Code

The Safety Annex source code should be cloned in a local directory using the Github repository:

https://github.com/loonwerks/AMASE.git

If this repository is on the target machine, the Safety Annex can be imported into the OSATE development environment folder titled: *Other Projects*. This repository includes a number of directories and the one containing the Safety Annex source code is titled: *safety\_annex/plugins.*

### Github Branches

After the development environment is set up and all repositories are imported, make sure that the *smaccm* branch is *develop*, and the *amase* branch is *master*. Those should be the default branches when checking out the git repository. If not, right click on the project folder and select *Team*.

Team -> Switch To -> New Branch

Then type into the textbox the branch as required.

### Run OSATE

At this point, everything should be in place to run the OSATE environment. Select the drop down menu next to the green “play” button on the menu. In the drop down menu that appears, select OSATE. This compiles the source code and the OSATE environment should appear after loading.

To test whether the safety annex has been correctly installed, a Safety Annex menu should appear in OSATE as shown in Figure 20. At this point, you are ready to import the Toy Example project (see Section 4.1) and begin your own safety analysis.

# Other Information

This section compiles the remaining pieces of information that may in some cases be useful to know.

## Defining Multiple Faults on a Single Output

When an output has multiple faults defined, the effects of such depend on what analysis is run.

* Verify in the Presence of Faults:

All faults are considered as model elements and if any of the defined faults cause violations of properties, the model checker will find these faults. Which is displayed to the user is model checker dependent.

* Generate Minimal Cut Sets:

All combinations that contribute to property violation will be displayed in the min cut sets.

The user should be aware that multiple fault definitions on the same output cannot have different return types. An example of this is if the output of a component is a nested data type. There are numerous ways faults can be defined for nested types (see Section 7.2.4.3 for an example of a fault node dealing with data type fields). One cannot define two faults on the output in such a way that one references the field type and one references the overall data type. An example of this is shown in Figure 43.

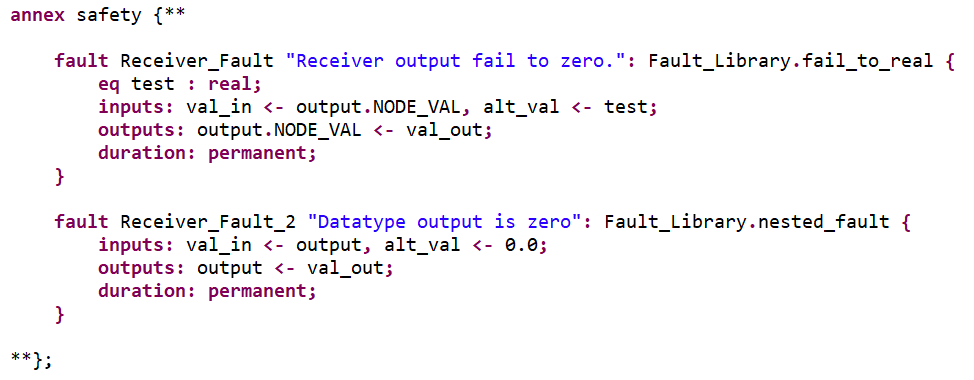


Figure 43: Different Return Types on Single Output

To reiterate, what is shown in the above figure is *not* supported. Only fault nodes with equal return types are allowed to be defined on the same output.

# Appendices

## Appendix 1: Fault Library

A library of commonly used fault node definitions can be found in any of the examples in the Github repository, but one is included here for convenience. Create a new AADL package within the project that will be using the file. Name the new package whatever you wish (often it is called “faults” or “common\_faults”) and then copy the following code into the body of the package.

**annex** agree {\*\*

**node** invert\_boolean**(**val\_in**:** **bool,** trigger**:** **bool)** **returns** **(**val\_out**:bool);**

**let**

val\_out **=** **if** trigger **then** **(not** val\_in**)** **else** val\_in**;**

**tel;**

**node** fail\_to\_zero**(**val\_in**:** **int,** trigger**:** **bool)** **returns** **(**val\_out**:** **int);**

**let**

val\_out **=** **if** trigger **then** **(**0**)** **else** val\_in**;**

**tel;**

**node** fail\_to\_one**(**val\_in**:** **int,** trigger**:** **bool)** **returns** **(**val\_out**:** **int);**

**let**

val\_out **=** **if** trigger **then** **(**1**)** **else** val\_in**;**

**tel;**

**node** fail\_to\_real**(**val\_in**:** **real,** alt\_val**:** **real,** trigger**:** **bool)** **returns** **(**val\_out**:** **real);**

**let**

val\_out **=** **if** trigger **then** alt\_val **else** val\_in**;**

**tel;**

\*\*};