Safety Annex Users Guide

Version 0.9

Danielle Stewart: University of Minnesota, dkstewar@umn.edu
Jing (Janet) Liu: Collins Aerospace, jing.liu@collins.com
Darren Cofer: Collins Aerospace, darren.cofer@collins.com
Mike Whalen: University of Minnesota, mwwhalen@umn.edu
Mats Heimdahl: University of Minnesota, heimdahl@umn.edu

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 Propagate Type Statement describing asymmetric faults.
0.7	7/9/2019	Danielle Stewart	Added running example to grammar description.
0.8	9/30/2019	Danielle Stewart Janet Liu	Added examples and installation updates.
0.8.1	1/19/2020	Janet Liu	Updated installation info and figure references.
0.8.2	4/1/2020	Danielle Stewart	Clarified fault activation statements, changed to new AGREE syntax, update user's guide.
0.9	5/1/2020	Danielle Stewart	Added interval statements, new install Instructions, new menu items.

Table of Contents

Table of Figures

1 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. We also provide a library of common fault node definitions that is customizable to the needs of system and safety engineers. This is found in Appendix 1: Fault Library.

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.

1.1 Github Repositories

Throughout this guide, we often refer users to the Safety Annex Github repository for access to the plugin and documentation [2]. https://github.com/loonwerks/AMASE/tree/master

The examples referenced throughout the User's Guide can be found in the *examples* directory of the GitHub repository [3]. https://github.com/loonwerks/AMASE/tree/master/examples

We also refer to the AGREE Users Guide [1] for certain syntactical elements and more descriptions on how this annex is used. This can be found at the following website: https://github.com/loonwerks/formal-methods-workbench/tree/master/documentation/agree

1.2 Document Overview

Section 2 provides installation instructions. This is followed in section 3 by a brief overview of AADL, AGREE, and the Safety Annex. The safety annex language and syntax is described in section 4. Sections 5 and 6 give more details and examples in the use of the safety annex. Section 7 provides some troubleshooting and tips. This is followed by appendices in section 8 and references in section 9.

2 Installations: Safety Annex, AGREE, AADL

In this section we present a brief overview of the Safety Annex/AGREE/OSATE tool suite followed by installation instructions.

2.1 Tool Suite Overview

Figure 1 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. The plugins work with OSATE to provide both a language (i.e., AGREE is an AADL annex to annotate the models with assume-guarantee behavioral contracts; Safety Annex is an AADL annex to annotate the model with faulty behaviors) and a tool (for 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.

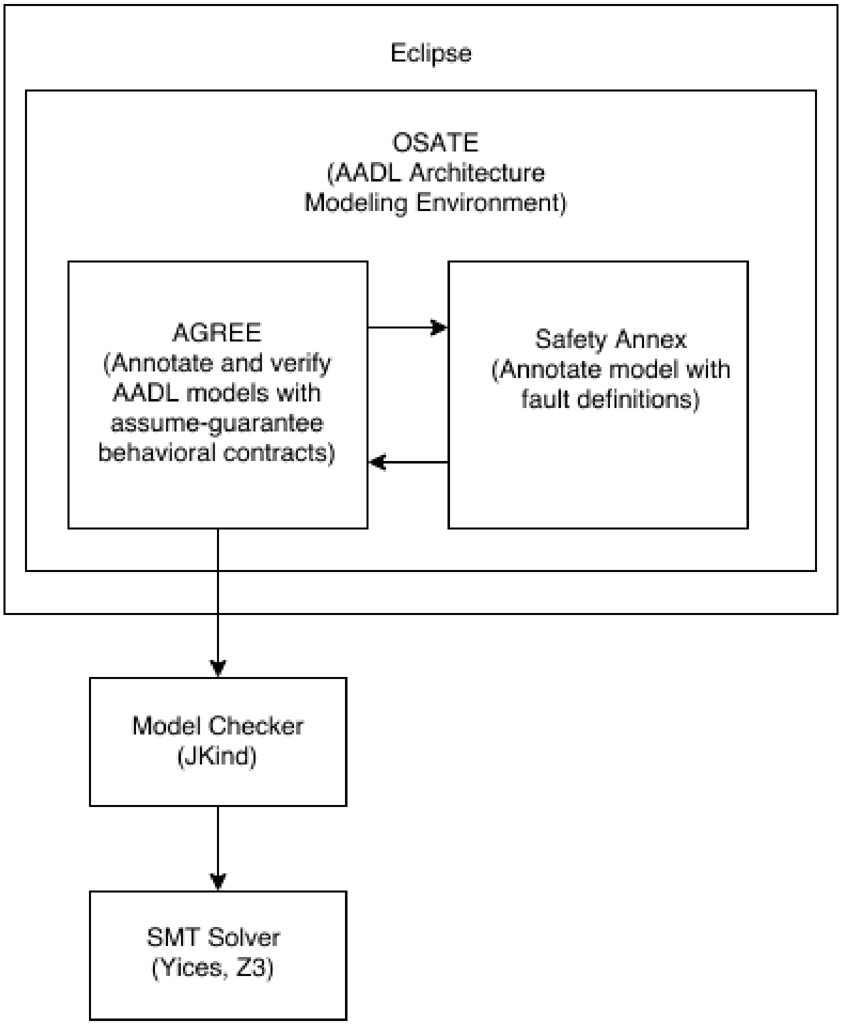


Figure 1: Overview of Safety Annex/AGREE/OSATE Tool Suite

2.2 Installation

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

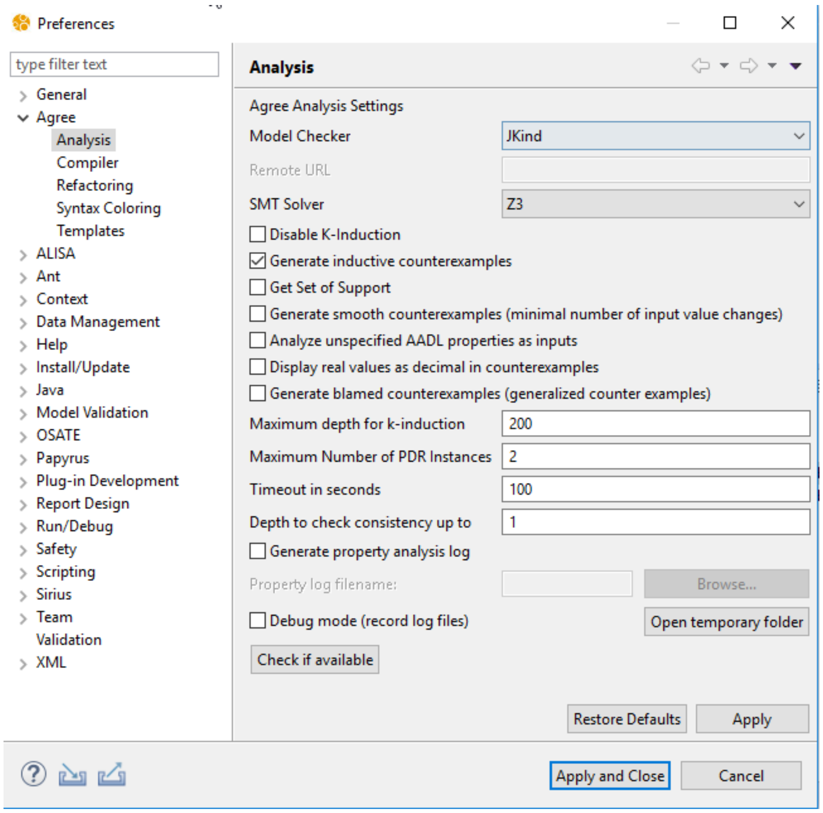
2.2.1 Install OSATE with Safety Annex and AGREE

(1) Install OSATE 2.7.0 found at the following link: https://osate.org/download-and-install.html
(2) Go to OSATE menu Help -> Install New Software.
(3) In the update site field, enter the following website: https://raw.githubusercontent.com/loonwerks/AMASE/master/safety-update-site/safety-annex_0.9
(4) Select "Safety Annex" and accept all licenses. OSATE will request a restart to complete the install. Do this.

After OSATE has restarted, a new menu item should appear called “AGREE.” The AGREE menu contains the Safety Annex options for fault analysis, as shown in Figure 9. The versions installed are: Osate 2.7.0, AGREE 2.4.0, Safety Annex 0.9.

2.2.2 Set AGREE Analysis Preferences

Use the SMT solver Z3 that comes with the Safety Annex/AGREE release and set the AGREE Analysis preferences as shown in Figure 2 by selecting the “Window” menu -> Preferences -> AGREE -> Analysis.



Agree Analysis Settings

Model Checker: JKind

Remote URL:

SMT Solver: Z3

☐ Disable K-Induction

☑ Generate inductive counterexamples

☐ Get Set of Support

☐ Generate smooth counterexamples (minimal number of input value changes)

☐ Analyze unspecified AADL properties as inputs

☐ Display real values as decimal in counterexamples

☐ Generate blamed counterexamples (generalized counter examples)

Maximum depth for k-induction: 200

Maximum Number of PDR Instances: 2

Timeout in seconds: 100

Depth to check consistency up to: 1

☐ Generate property analysis log

Property log filename:

☐ Debug mode (record log files)

Figure 2: AGREE Analysis Preferences

At this point, users are ready to import the projects (e.g., the Toy Example project in Section 3.1) and begin safety analysis. Figure 2 shows the AGREE Analysis Preference pane as an illustration of the possible preferences. In order to use the Safety Annex, JKind is required for the model checker. In order to collect all minimal cut sets, Z3 is required for the SMT solver. The rest of the values in this figure are not required and they can be changed according to the user’s preference. Consider having the “Maximum Number of PDR Instances” to be greater than zero if having multiple cores (e.g., for 8 cores or more, set to at least 2); this enables the model checker to give definite answer (valid/invalid) for some properties with previously unknown verification results.

<u>Troubleshooting:</u> If verification runs are giving unknown results, check the PDR settings and increase if needed. This may solve the problem.

2.3 Development Environment Installation

If a user wishes to import the Safety Annex plugin into an Eclipse Development Environment, the source code can be found in the GitHub repository [2] under the directory *safety_annex/plugin*. If any questions or problems arise during safety annex installation or for a detailed guide on development environment install, contact the authors of this document.

Required projects for development environment:

- AGREE: https://github.com/loonwerks/AGREE.git
- Z3: https://github.com/loonwerks/z3-plugin.git
- Safety Annex: https://github.com/loonwerks/AMASE.git

3 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 3.

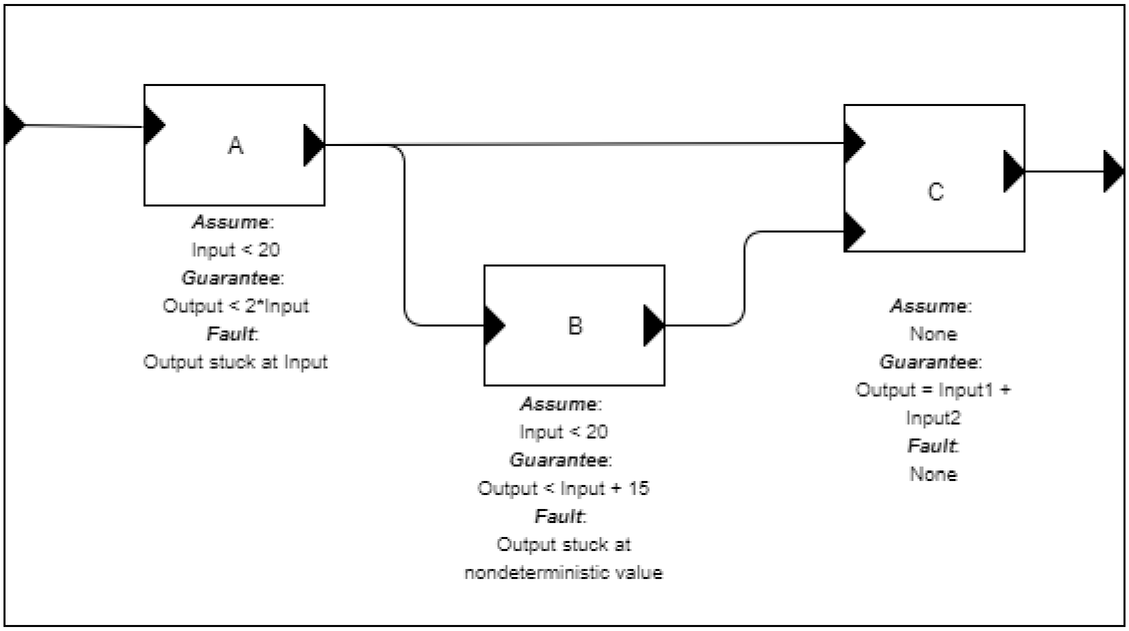


Figure 3: Toy Example for Safety Annex and AGREE

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 3.

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 4 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;
 end Integer_Toy;
```

Figure 4: AADL Code for Toy Example with AGREE and Safety Annexes

In Figure 4, **`system`**s define hierarchical "units" of the model. They communicate over **`port`**s, which are 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 4.4.2.

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 5: Fault Hypothesis Example

3.1 Using the Safety Annex AADL Plugin

The example project used in the rest of this section can be retrieved from the GitHub repository in the examples directory [3] and is called *Toy_Example_Safety*. Assuming the necessary tools are installed (see section 2.2), the model can be imported by choosing File > Import:

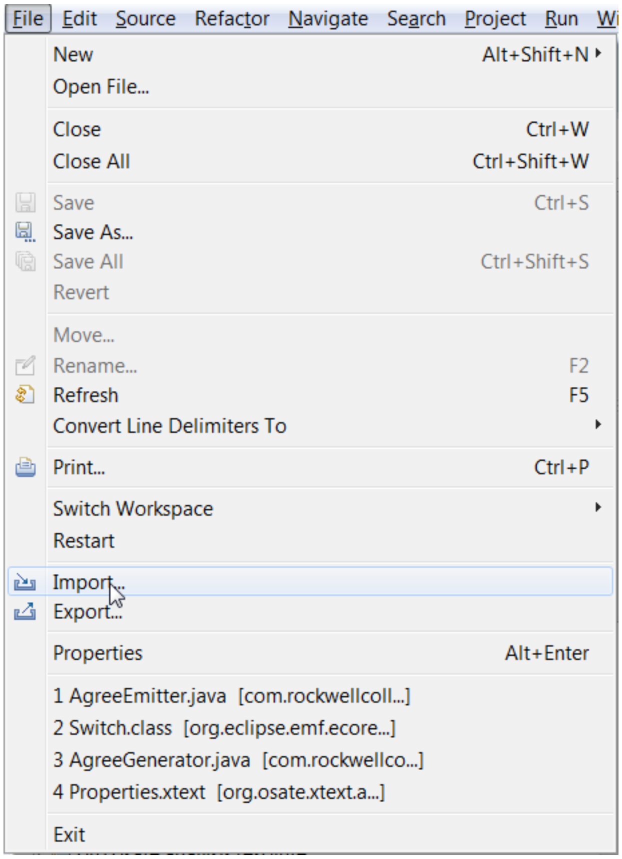
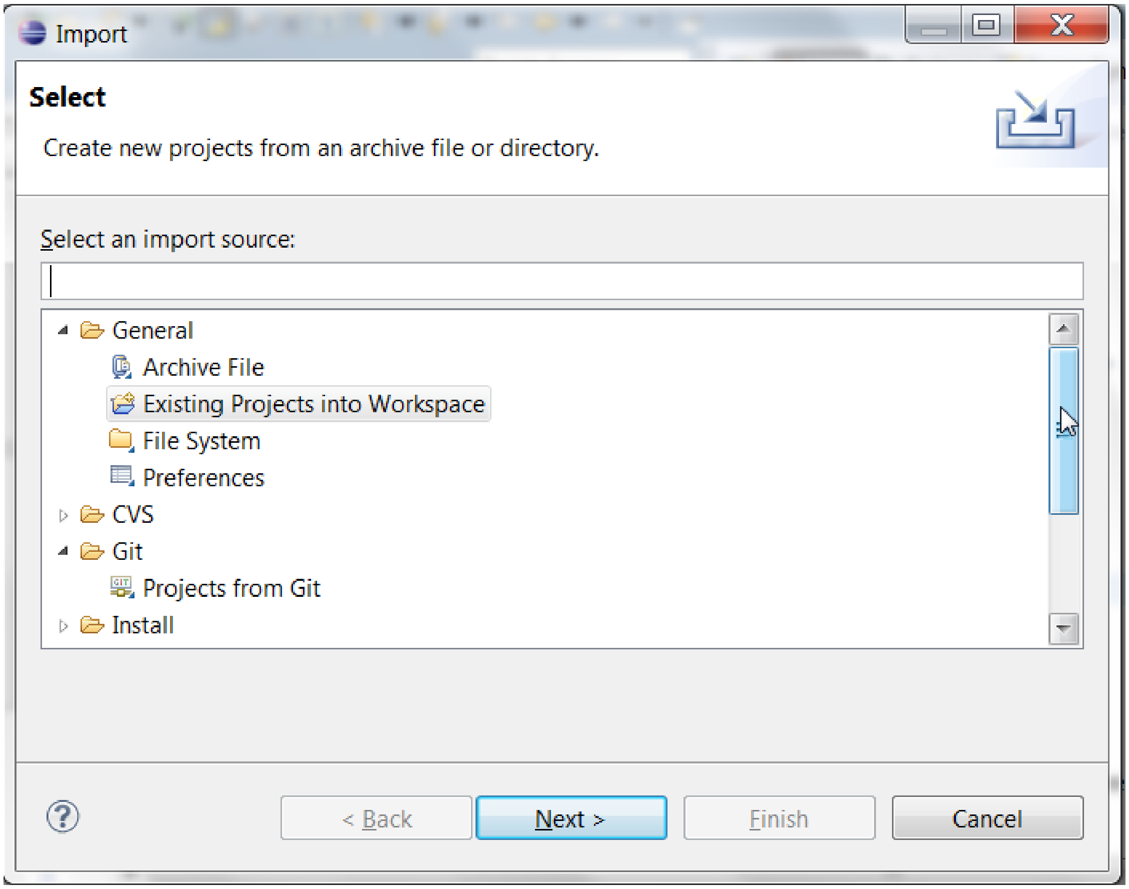


Figure 6: Import Menu Option

Then choosing "Existing Project into Workspace."



Import

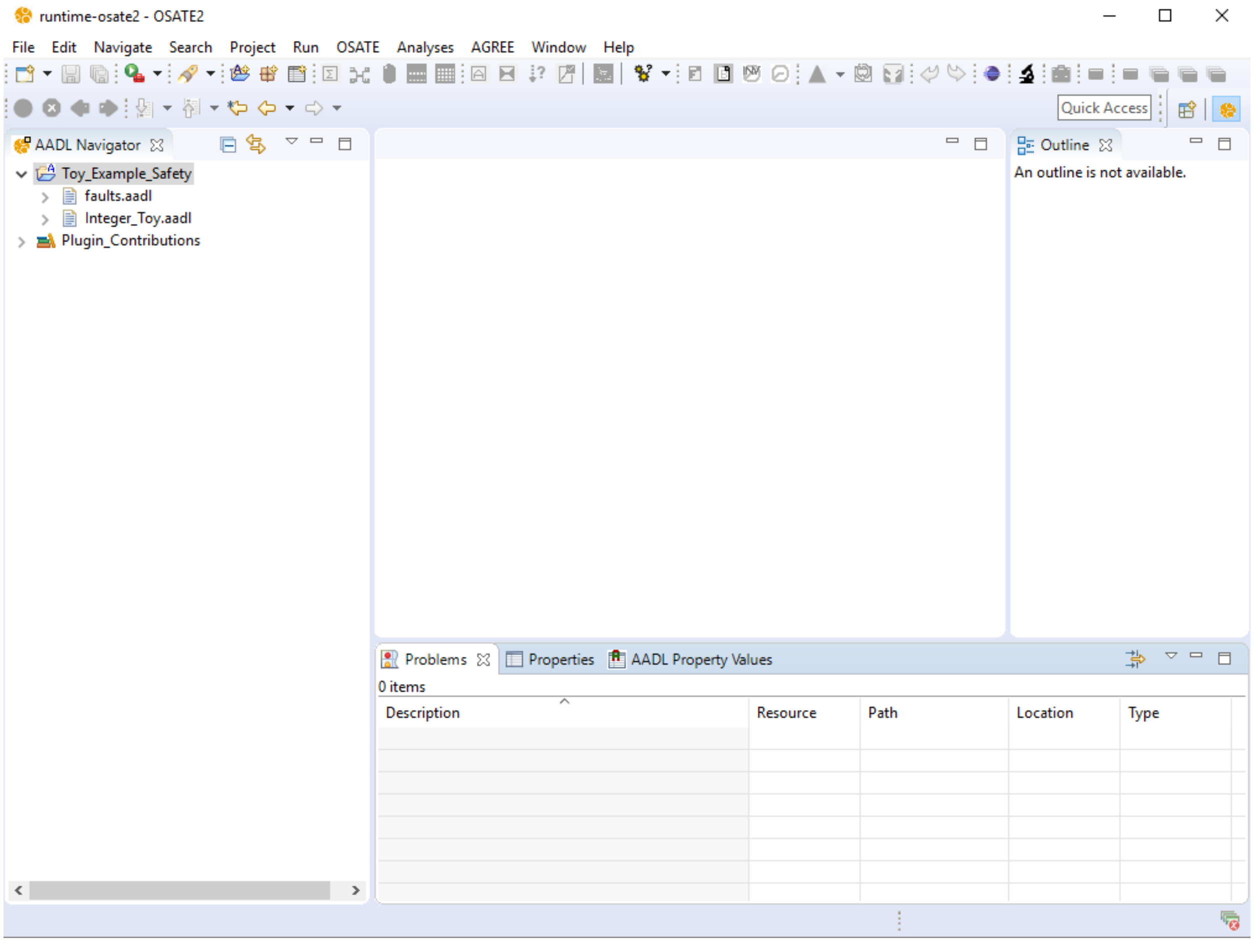
Select

Create new projects from an archive file or directory.

Select an import source:

Figure 7: Importing Toy Example Project

and navigate to the unzipped directory after pressing the Next button. Figure 8 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.



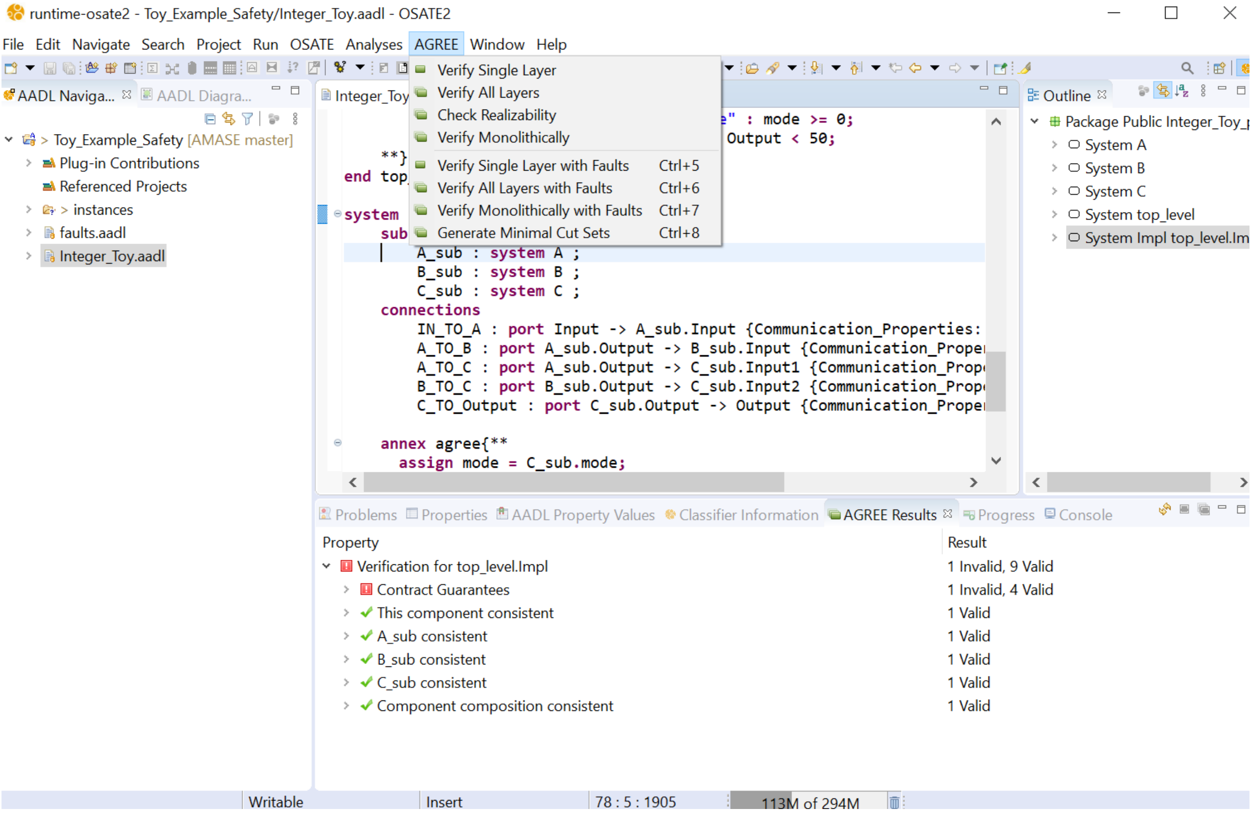
An outline is not available.

0 items

Description	Resource	Path	Location	Type

Figure 8: 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 "Verify Single Layer with Faults" from the AGREE menu as shown in Figure 9.

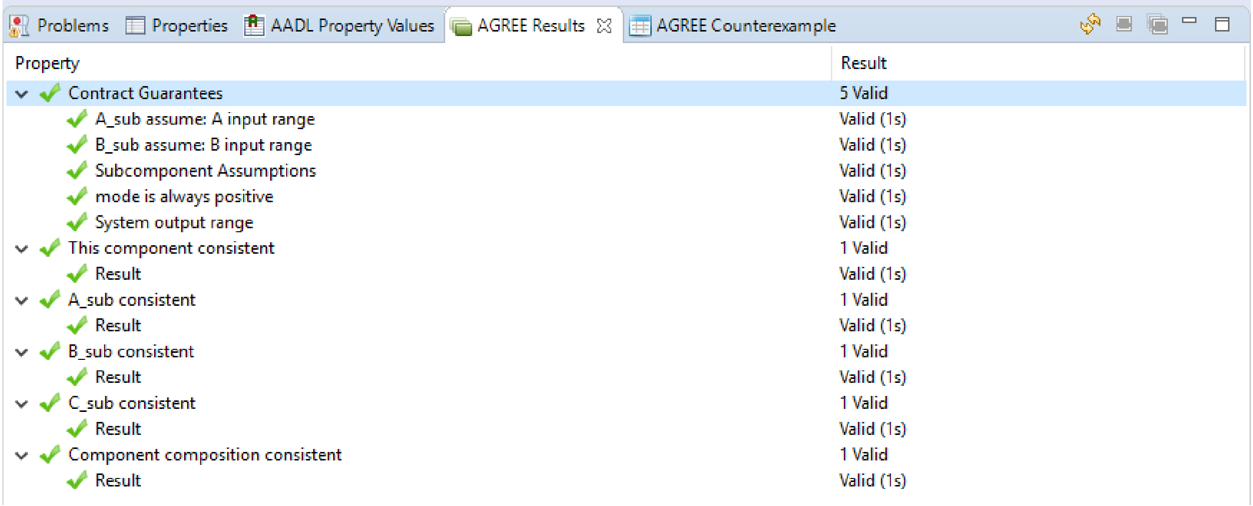


```
                                  e" : mode >= 0;
                                   Output < 50;
    **}
end top

system
    sub
        A_sub : system A ;
        B_sub : system B ;
        C_sub : system C ;
    connections
        IN_TO_A : port Input -> A_sub.Input {Communication_Properties:
        A_TO_B : port A_sub.Output -> B_sub.Input {Communication_Prope
        A_TO_C : port A_sub.Output -> C_sub.Input1 {Communication_Prop
        B_TO_C : port B_sub.Output -> C_sub.Input2 {Communication_Prop
        C_TO_Output : port C_sub.Output -> Output {Communication_Prope

    annex agree{**
      assign mode = C_sub.mode;
```

Property	Result
Verification for top_level.Impl	1 Invalid, 9 Valid
Contract Guarantees	1 Invalid, 4 Valid
This component consistent	1 Valid
A_sub consistent	1 Valid
B_sub consistent	1 Valid
C_sub consistent	1 Valid
Component composition consistent	1 Valid

Figure 9: AGREE and Safety Analysis Dropdown Menu

To invoke AGREE nominal analysis (no faults are considered in the analysis), select in the AGREE menu: "Verify Single Layer." As AGREE runs, you should see checks for "Contract Guarantees", "Contract Assumptions", and "Contract Consistency" as shown in Figure 10.



Property	Result
Contract Guarantees	5 Valid
A_sub assume: A input range	Valid (1s)
B_sub assume: B input range	Valid (1s)
Subcomponent Assumptions	Valid (1s)
mode is always positive	Valid (1s)
System output range	Valid (1s)
This component consistent	1 Valid
Result	Valid (1s)
A_sub consistent	1 Valid
Result	Valid (1s)
B_sub consistent	1 Valid
Result	Valid (1s)
C_sub consistent	1 Valid
Result	Valid (1s)
Component composition consistent	1 Valid
Result	Valid (1s)

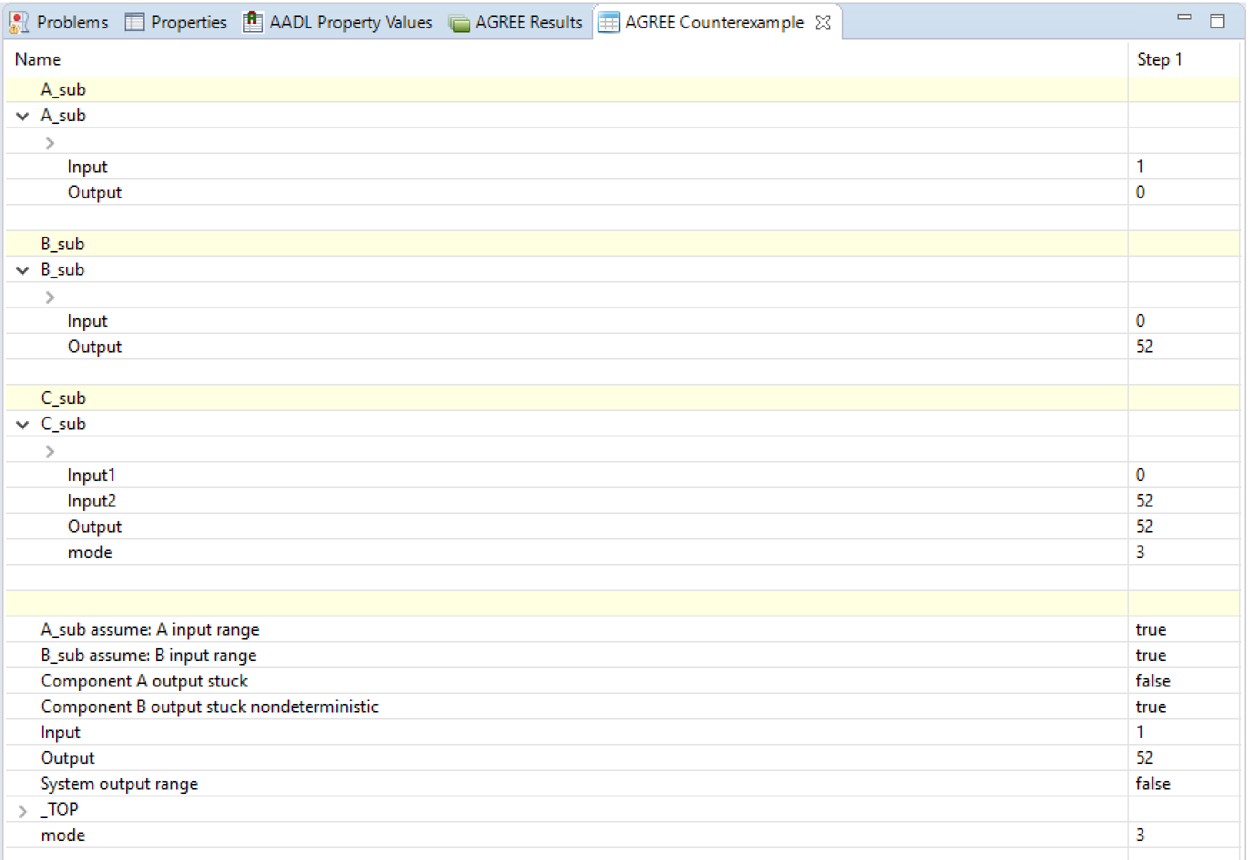
Figure 10:AGREE Verification Results

If "Verify in the Presence of Faults" was checked by the user, this will run the analysis and will change the AGREE contracts accordingly. The user can also run "Generate Minimal Cut Sets" on this model and get a textual representation of the minimal cut sets that cause violation of the property at the top level. For more information on types of analysis options, see Section 6.1.4.

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 11 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 11 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 4.

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.



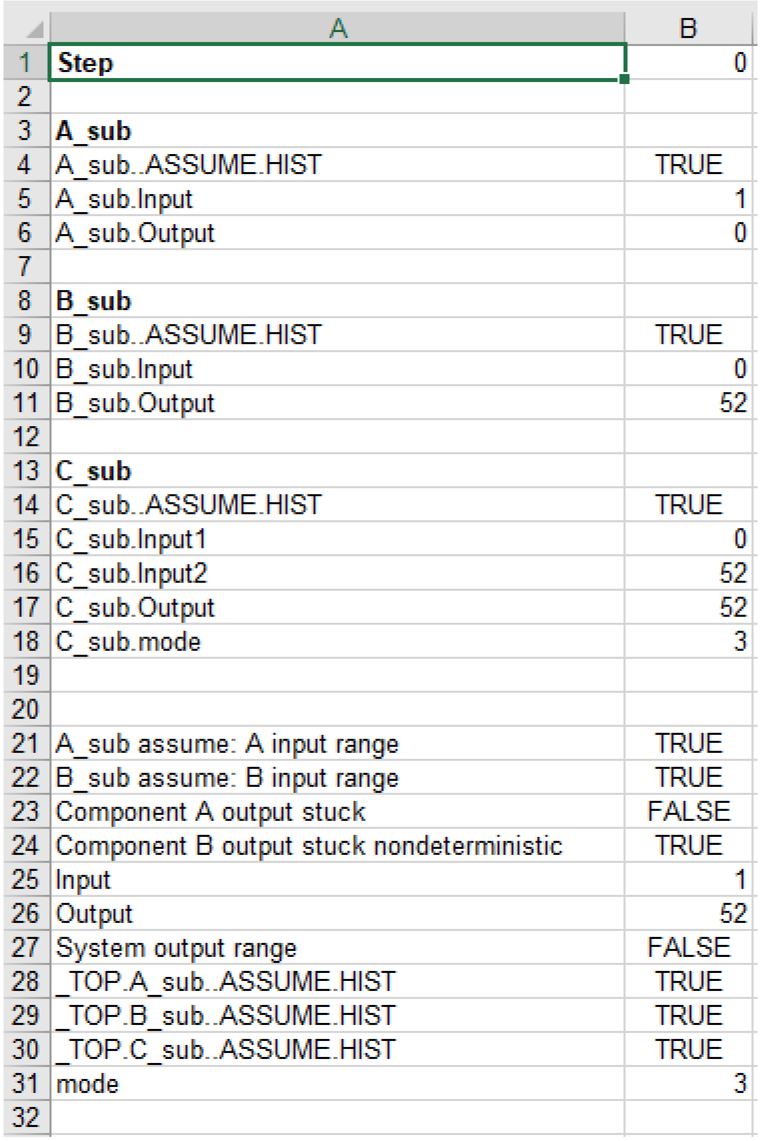
Name	Step 1
A_sub	
⌄ A_sub	
›	
Input	1
Output	0
B_sub	
⌄ B_sub	
›	
Input	0
Output	52
C_sub	
⌄ C_sub	
›	
Input1	0
Input2	52
Output	52
mode	3
A_sub assume: A input range	true
B_sub assume: B input range	true
Component A output stuck	false
Component B output stuck nondeterministic	true
Input	1
Output	52
System output range	false
› _TOP	
mode	3

Figure 11: 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.
5. Now choose the "Add…" button next to "Associated Editors"
6. Choose the "External Programs" radio button
7. Select "Microsoft Excel Worksheet" and click OK.

The generated Excel file for the example is shown in Figure 12.



	A	B
1	**Step**	0
2		
3	**A_sub**	
4	A_sub..ASSUME.HIST	TRUE
5	A_sub.Input	1
6	A_sub.Output	0
7		
8	**B_sub**	
9	B_sub..ASSUME.HIST	TRUE
10	B_sub.Input	0
11	B_sub.Output	52
12		
13	**C_sub**	
14	C_sub..ASSUME.HIST	TRUE
15	C_sub.Input1	0
16	C_sub.Input2	52
17	C_sub.Output	52
18	C_sub.mode	3
19		
20		
21	A_sub assume: A input range	TRUE
22	B_sub assume: B input range	TRUE
23	Component A output stuck	FALSE
24	Component B output stuck nondeterministic	TRUE
25	Input	1
26	Output	52
27	System output range	FALSE
28	_TOP.A_sub..ASSUME.HIST	TRUE
29	_TOP.B_sub..ASSUME.HIST	TRUE
30	_TOP.C_sub..ASSUME.HIST	TRUE
31	mode	3
32		

Figure 12: Generated Excel File for Counterexample

4 Safety Annex Language

In this section 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.

4.1 Syntax Overview

Before describing the details of the language, we provide some general notes about the syntax. Elements 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 [1].

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.

4.2 Lexical Elements and Types

For a more thorough description of lexical elements and types, we refer to the AGREE User Guide [1]. 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;
```

Figure 13: 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*.

4.3 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 14.

```
SpecStatement: 'fault' ID (STRING)? ':'  faultDefName '{' (FaultSubcomponent)* '}'
             | 'analyze' ':'  AnalysisBehavior
             | ‘hw_fault ':'  ID (STRING)? ':'  '{' (HWFaultSubcomponent)* '}'
             | 'propagate_from' ':'  '{' (SourceFaultList) '@' (SourceCompPath)  '}'
                     ‘to’   '{' (DestFaultList) '@' (DestCompPath)  '}'
             | 'fault_activation':' agreeVarName=ID ('@' agreeCompPath=NestedDotID)?
                     ‘=’   (faultName=ID) '@' (faultCompPath=NestedDotID)  '}'


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 ';'
       | 'disable' ':' Bool_Literal ';'
       | 'propagate_type' ':' PropagationTypeConstraint ';'
       | SafetyEqStatement

HWFaultSubcomponent:  'duration' ':' TemporalConstraint (Interval)? ';'
                   |     'probability' ':' Real_Literal ';'
                   |     'propagate_type' ':' PropagationTypeConstraint ';'


PropagationTypeConstraint: 'asymmetric'
                        |     'symmetric'

TemporalConstraint: 'permanent'

SafetyEqStatement: 'eq' (Arg (',' Arg)*) ('=' Expr)? ';'
                |     'safety_interval' Arg '=' Interval ';'
```

Figure 14: 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 4 contains a system component “A” (Figure 15) 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 injected 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 performing the required operation. 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 15: Toy Example System A

4.4 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.

4.4.1 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:

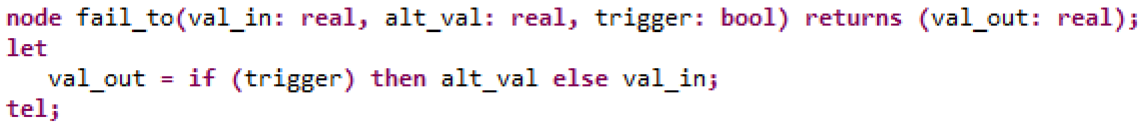
```
FaultSubcomponent:  'inputs' ':' NamedID '<-' Expr (','NamedID '<-' Expr)* ';'
       | 'outputs' ':' NestedDotID '<-' NamedID   (','NestedDotID '<-' NamedID)* ';'
       | 'duration' ':' TemporalConstraint (Interval)? ';'
       | 'probability' ':' Real_Literal ';'
       | 'enabled' ':' TriggerCondition ';'
       | '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 16.

```
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 16: 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 [1] description of nodes. 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 8.1 for convenience. This particular node definition is given in Figure 17.



```
node fail_to(val_in: real, alt_val: real, trigger: bool) returns (val_out: real);
let
   val_out = if (trigger) then alt_val else val_in;
tel;
```

Figure 17: 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.

4.4.1.1 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 16 . 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 18.

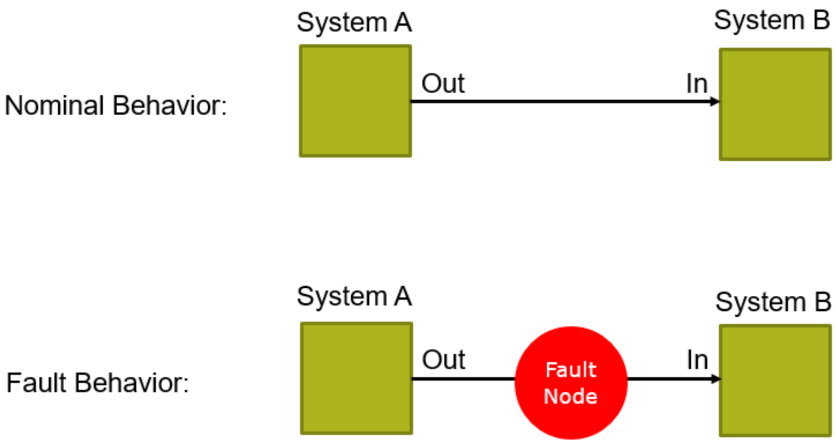


Figure 18: Fault node wrapping output of AADL component

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 17 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 4.4.1.8.

Record types in AADL are supported can be used in input and output statements (see Section 5.2).

Note: The trigger value: As can be seen in the fail_to fault node definition (Figure 17), 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 4.4.4).

4.4.1.2 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.

Record types in AADL are supported and can be used in input and output statements of a fault.

4.4.1.3 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 4.4.2.2. 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.

4.4.1.4 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.

4.4.1.5 Disable Statement

At times, it is useful to disable faults from the analysis without needing to delete or comment out the fault definition. In this case, the user can define an optional disable statement in the fault definition.

```
disable : true;
```

This removes the fault from being considered in the analysis. Any fault definition without a disable statement defined is by default included in the analysis.

4.4.1.6 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 19.

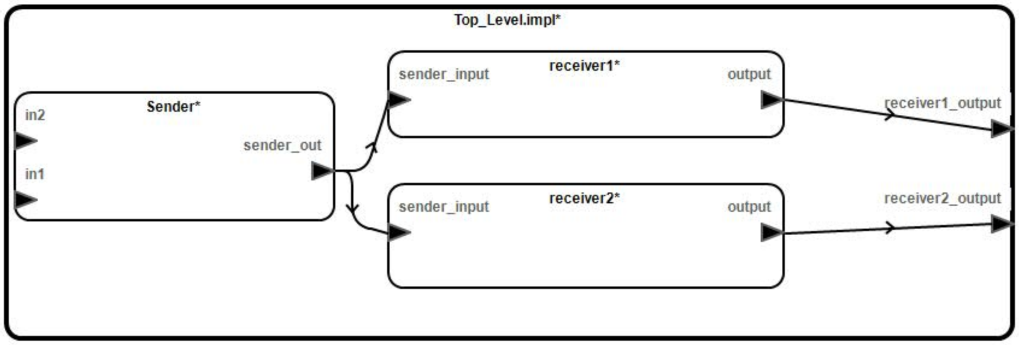


Figure 19: Sender-Receiver AADL Model

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 is sent 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. When it is necessary to model the receivers seeing different values from the sender, an *asymmetric* fault can be defined.

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 19) is shown in Figure 20.

```
fault Sender_Fault "Or output is zero" : Common_Faults::fail_to_real {
              inputs: val_in <- Output, alt_val <- 0.0;
              outputs: Output <- val_out;
              duration: permanent;
              propagate_type: assymetric;
}
```

Figure 20: Fault Statement With Asymmetric Propagate Type

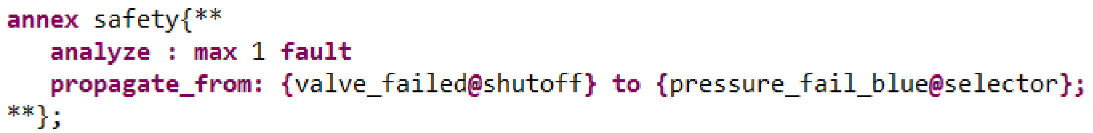
As seen in Figure 20, the rest of the fault definition is the same as in the symmetric case and the library of fault nodes can be used to define the faulty output.

For a simple example of this fault in action, see the examples directory of our GitHub repository under *Byzantine_Example* [3].

Note: The fault value that can override the output of the sender component can effect *less than or equal to* the total number of connections. The user cannot specify which connections are affected or how many. The model checker will explore all possible scenarios during property verification.

4.4.1.7 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 4.4.4. An example of a fault dependency specification is shown in Figure 21, showing that the *valve_failed* fault at the shutoff subcomponent triggers the *pressure_fail_blue* fault at the selector subcomponent.



```
annex safety{**
   analyze : max 1 fault
   propagate_from: {valve_failed@shutoff} to {pressure_fail_blue@selector};
**};
```

Figure 21: 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 or the type of analysis run. 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, the source fault must be a hardware fault. See Section 4.4.4 for a full description of a hardware fault statement.

4.4.1.8 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 two new kinds of equations.

4.4.1.8.1 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 internal variables used inside the scope of the component. 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.

4.4.1.8.2 Interval Statements

Similar to the eq statement, an interval statement can provide flexibility with assigning fail-to values for faults on components.

```
        Safety_interval example : int = [0,3];
```

A statement like this restricts the failure values to be integers from 0 to 3. This can be passed into a fault node input as shown below.

```
annex safety {**
        fault stuck_at_fault_A "Component A output stuck" : faults::fail_to {
               safety_interval example : int = [0,3];
               inputs: val_in <- Output, alt_val <- example;
               outputs: Output <- val_out;
              probability: 5.0E-5;
               duration: permanent;
       }
**};
```

The following interval types are supported:

- Integer and real
- Open, e.g., (0,1)
- Closed, e.g., [1.1, 10.1]
- Combinations of open/closed, e.g., (-2, 4]

The type used in the interval (real/int) must match that of the fault node input parameter.

4.4.2 Analysis Statement

An analysis statement is given in the system implementation under analysis. This is shown in Figure 22 and Figure 23. By referencing the Toy Example, you can see that the analysis statement is within the top level system implementation (Figure 4). 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) When verifying in the presence of faults with AGREE, to verify any layer with faults, the analysis statements must be added to the system implementation of that layer.

(2) At most one of the analysis statements (max n faults or probability) can be present.

(3) The nominal model must prove in the absence of faults before proceeding to generate minimal cut set analysis. Otherwise, no valid minimal cut sets can be generated for the system.

4.4.2.1 Max N Faults Analysis

As shown below, 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 22: Max N Fault Threshold Example

The options for max N fault analysis:

(1) The user can perform *monolithic max N fault analysis* in the presence of faults by specifying max N fault analysis statement in the model and select “Verify Monolithically with Faults” in AGREE menu. 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* in the presence of faults by specifying max N fault analysis statement in the model and select “Verify All Layers with Faults” in AGREE menu. (Note: to perform a single layer of analysis, select “Verify Single Layer with Faults” in AGREE menu.) 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 used in the proof (as shown in “Set of Support” when “Get Set of Support” is turned on for AGREE analysis preferences), 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 generate all minimal cut sets* up to cardinality N by specifying max N fault analysis statement in the top level of the model to be verified and select “Generate Minimal Cut Sets”. In this case, lower level faults and problematic subcomponents will be revealed in the minimal cut sets generated for the violation 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).

4.4.2.2 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 4. The syntax of probabilistic analysis is shown in Figure 23 with a top level threshold of 1.0E-7.

```
annex safety {**
	analyze : probability 1.0E-7
**};
```

Figure 23: Probability Threshold Example

There are two options for probabilistic analysis.

(1) The user can run *monolithic probabilistic analysis* on the model with the given probability threshold by specifying probabilistic fault analysis statement in the model and select “Verify Monolithically with Faults” in AGREE menu. The possible fault combinations are calculated based on the top level probability threshold and independence between faults. 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.

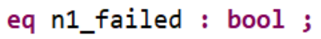
(2) The user can run *compositional probabilistic analysis* on the model with the given probability threshold by specifying the probabilistic analysis statement in the model and select “Generate Minimal Cut Sets”. In this case, the minimal cut sets that can violate the top level property within this threshold are provided to the user.

In all of these cases, the properties need to be verified valid in the nominal model first. If it does not, the results from the fault analysis will return the empty set for “Generate Minimal Cut Sets”.

For more information on running the various types of analysis, we refer readers to Section 6. This section details specific examples (available in the GitHub repository) and goes through the steps for all forms of analysis and how to interpret the results.

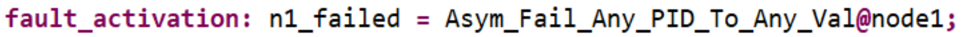
4.4.3 Fault Activation Statement

The capability to specify whether or not a fault is activated can be done in Safety Annex through the use of a fault statement. First, users need to declare boolean *eq* variables in the AGREE annex of the AADL component type or implementation at the “top level” of the analysis. In other words, if analysis is being run from *compA.impl*, then the eq variables need to be defined in either *compA* (type) AGREE annex or in *compA.impl* (implementation) AGREE annex.



```
eq n1_failed : bool ;
```

Users can then assign the activation status of specific faults to those *eq* variables in Safety Annex of the AADL system implementation where the AGREE verification applies. As per the example in the previous paragraph, this corresponds to *compA.impl* Safety Annex. The fault activation statement is located in the Safety Annex of this implementation:



```
fault_activation: n1_failed = Asym_Fail_Any_PID_To_Any_Val@node1;
```

It refers to the fault name (e.g., “Asym_Fail_Any_PID_To_Any_Val” as defined in the following fault definition) and the name of the component instance in the AADL system implementation (e.g., “node1”) where the fault is defined in that component.

```
fault Asym_Fail_Any_PID_To_Any_Val "Node output is asymmetric" :
                          Common_Faults::fail_any_PID_to_any_value {
              eq pid1_val : real;
              eq pid2_val : real;
              eq pid3_val : real;
              eq pid4_val : real;
              inputs: val_in <- Node_Out,
                      pid1_val <- pid1_val,
                      pid2_val <- pid2_val,
                     pid3_val <- pid3_val,
                     pid4_val <- pid4_val;
              outputs: Node_Out <- val_out;
              duration: permanent;
              propagate_type: asymmetric;
}
```

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.

4.4.4 Hardware Fault Statement

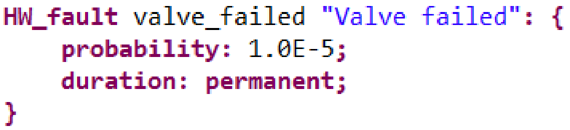
Failures in hardware (HW) components can trigger behavioral faults in the software (SW) or system (SYS) components that depend on them. For example, a CPU failure may trigger faulty behavior in threads bound to that CPU. In addition, a failure in one HW component may trigger failures in other HW components 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 24. This example is taken from the Wheel Brake System model (WBS_arch4_v2) [3]. 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.



```
HW_fault valve_failed "Valve failed": {
    probability: 1.0E-5;
    duration: permanent;
}
```

Figure 24: Hardware Fault Statement

Users must specify fault dependencies outside of fault statements using propagation statements. For more information on propagation type statements, see section 4.4.1.6.

Nested dependencies are not supported. As an example of this, assume the notation $f_i \rightarrow \{f_j\}$ means that f_j is dependent on f_i.

The user cannot define the following:
$f_1 \rightarrow \{f_2, f_3\}$

$f_3 \rightarrow \{f_4\}$

Instead, they should define in this case:
$f_1 \rightarrow \{f_2, f_3, f_4\}$

To summarize, the fault statement (Figure 24) belongs in the source and destination components and defines the type of fault that occurs. The propagate-from statement (Section 4.4.1.7) is located in the AADL system implementation where the AGREE verification applies. This links these dependent faults and uses their information throughout the analysis.

Note: At the time of this current release, dependencies are only implemented while using *Verify in the Presence of Faults* analysis. They are not used in the *Generate Min Cut Set* analysis. See Section 6.1.4 for a discussion on the types of analysis available.

For a simple example on dependent faults and propagations, see *HW_Fault_Propagation* example located in GitHub [3].

4.4.4.1 Duration, Probability, and Propagation-Type Statements

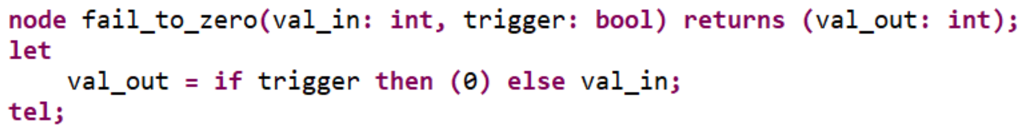
For the descriptions of the Duration, Probability, and Propagation-Type statements in Hardware Fault Statements, please refer to sections 4.4.1.3, 4.4.1.4, and 4.4.1.6. The syntax and explanations are identical in both independent and dependent faults.

5 Library and Custom Made Fault Nodes

A library of fault nodes is available in this document (Appendix 1: Fault Library). 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. Fault node follows the same conventions as AGREE node. A description of the fault nodes is provided here for your convenience and more information can be found in the AGREE Users Guide Section under Node Definitions [1].

To use this library of commonly used fault node definitions, follow the directions found in Appendix 1: Fault Library and place in users' OSATE project directory. This can be referred to within said project. To use the faults defined in the fault library, simply include the AADL package that contains the fault library (by using AADL "with" clause) in the AADL package where the Safety Annex resides. Users can use the commonly used fault node definition provided there, or create new fault node definitions

A basic fault node is shown below:



```
node fail_to_zero(val_in: int, trigger: bool) returns (val_out: int);
let
    val_out = if trigger then (0) else val_in;
tel;
```

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. Users cannot pass a trigger into this node call.

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 will become zero. Otherwise it remains the same, i.e. the AADL component output value will be unchanged.

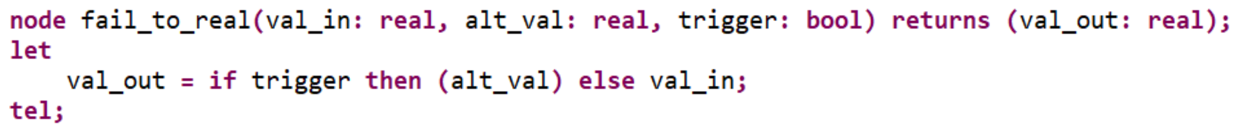
Many of the commonly used fault node definitions may be helpful, but in some cases it is necessary for the user to define fault models specific to a domain or model.

5.1 Nondeterministic Failure Values

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

```
fault Sender_Fault "Or output is zero" : Common_Faults::fail_to_real {
              eq nondeterministic_fail : real ;
              inputs: val_in <- Output, alt_val <- nondeterministic_fail;
              outputs: Output <- val_out;
              duration: permanent;
              propagate_type: asymmetric;
}
```

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.



```
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;
```

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 predefined fail-to value is desired. The node call is the same, but the eq statement in the deterministic fail-to value case will have a value assigned like below:

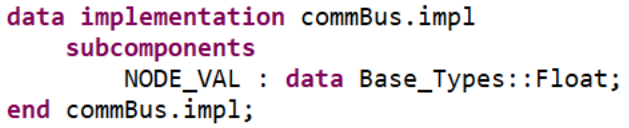
```
fault Sender_Fault "Or output is zero" : Common_Faults::fail_to_real {
              eq nondeterministic_fail : real = 0.0 ;
              inputs: val_in <- Output, alt_val <- nondeterministic_fail;
              outputs: Output <- val_out;
              duration: permanent;
              propagate_type: asymmetric;
}
```

5.2 Nested Data Structures

When defining a fault model for an output that is of a nested data type, one must pass the entire output to the fault model and specify which fields have certain failure values to guarantee expected fault behavior. Passing a field of a data implementation into a fault node can leave other fields of that output unconstrained, thus creating unwanted behavior.

The following example can be found in the GitHub repository example directory under *ByzantineExampleNestedTypes* [3].

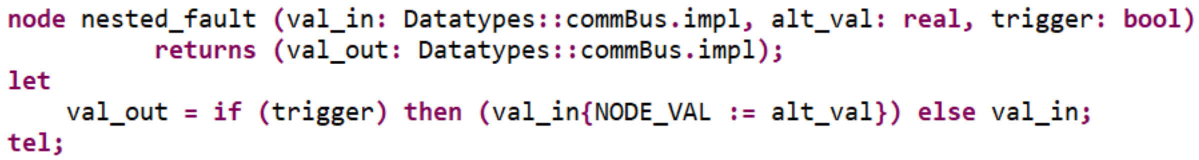
The datatype is called commBus.impl and has one field, NODE_VAL, which is of primitive type real.



```
data implementation commBus.impl
    subcomponents
        NODE_VAL : data Base_Types::Float;
end commBus.impl;
```

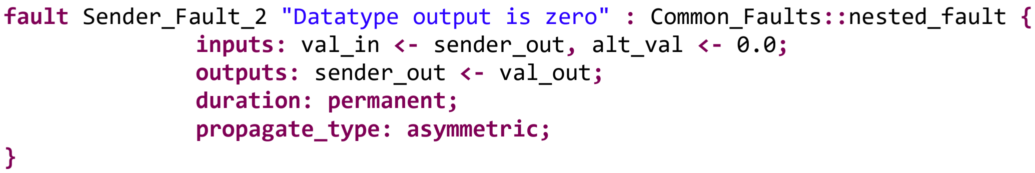
A particular component (Sender.aadl) has an output of type commBus.impl and we wish to model a fail to zero value on the NODE_VAL field.

The 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.



```
node nested_fault (val_in: Datatypes::commBus.impl, alt_val: real, trigger: bool)
          returns (val_out: Datatypes::commBus.impl);
let
    val_out = if (trigger) then (val_in{NODE_VAL := alt_val}) else val_in;
tel;
```

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:



```
fault Sender_Fault_2 "Datatype output is zero" : Common_Faults::nested_fault {
              inputs: val_in <- sender_out, alt_val <- 0.0;
              outputs: sender_out <- val_out;
              duration: permanent;
              propagate_type: asymmetric;
}
```

We provide another node definition that can also be found in the example is titled *PIDByzantineAgreement* in the example directory. A detailed description of the project is provided in Section 6.2.4. This example shows that multiple fields can be altered in a fault node definition.

```
--allow each field of the output record to fail to random value
node fail_any_PID_to_any_value (val_in: Node_Msg.Impl, pid1_val: int, pid2_val: int,
    pid3_val: int, pid4_val: int, trigger: bool) returns (val_out: Node_Msg.Impl);
let
    val_out =
        if (trigger)
            then(val_in{Node1_PID_from_Node1 := pid1_val}
                       {Node2_PID_from_Node2 := pid2_val}
                       {Node3_PID_from_Node3 := pid3_val}
                       {Node4_PID_from_Node4 := pid4_val})
         else val_in;
tel;
```

6 In Depth Examples

In this section, examples are presented 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 Byzantine problem and its corresponding mitigation. Both of these examples can be found in the GitHub repository [3].

6.1 The Sensor Example

6.1.1 AADL Architecture

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 25.

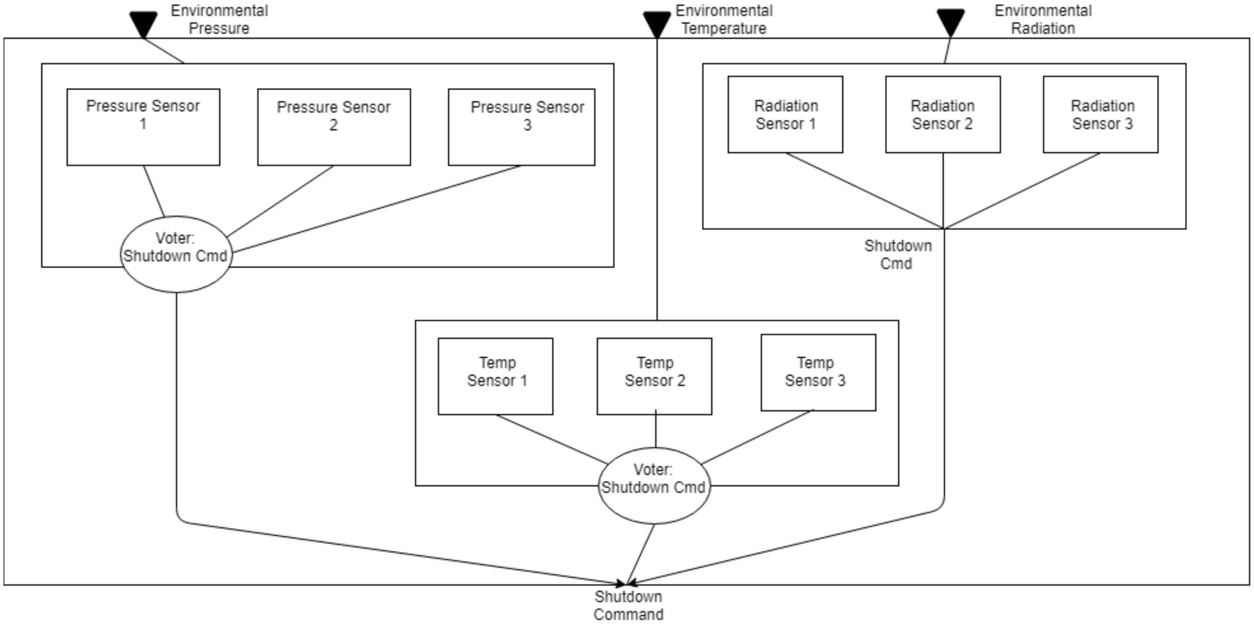


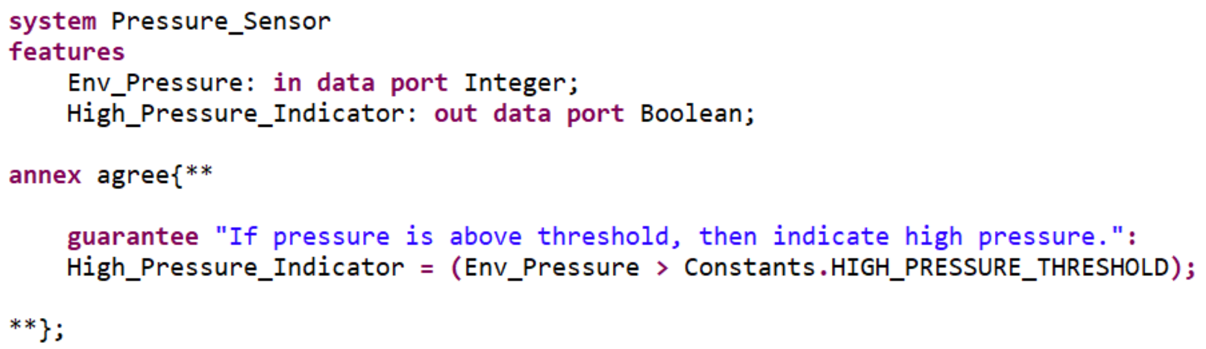
Figure 25: Sensor Example Architecture

There are majority voting subcomponents that perform a check on the pressure and temperature sensor outputs. For instance, 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. In this case, adding a voter makes the system resilient up to one sensor fault in the pressure and temperature subsystem.

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

6.1.2 AGREE Behavioral Model

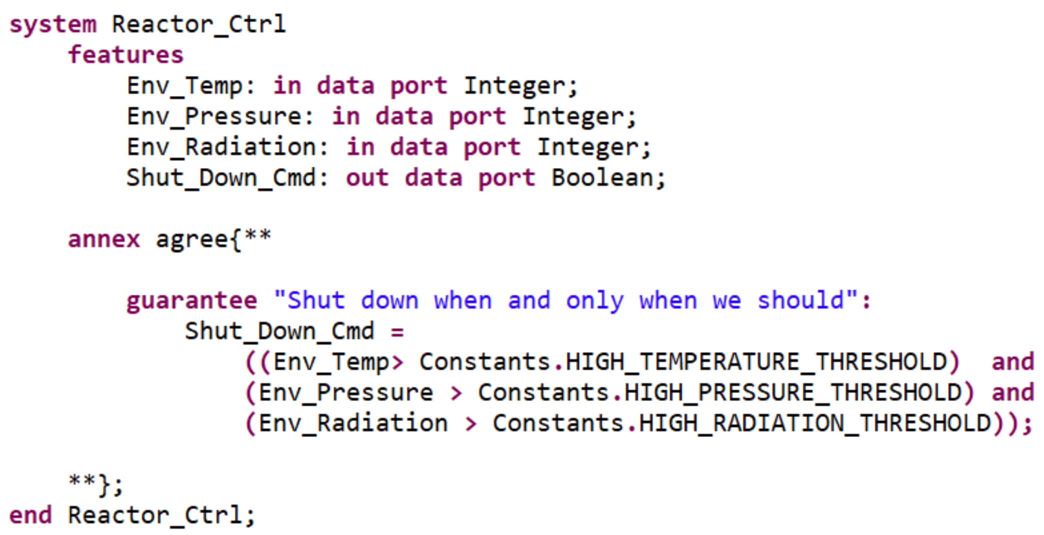
The AGREE contracts on the sensors (the leaf level components in the architecture hierarchy) state that when pressure (or temp/radiation) is above threshold, output a shutdown command. The AADL and AGREE code describing a pressure sensor is shown in Figure 26.



```
system Pressure_Sensor
features
    Env_Pressure: in data port Integer;
    High_Pressure_Indicator: out data port Boolean;

annex agree{**

    guarantee "If pressure is above threshold, then indicate high pressure.":
    High_Pressure_Indicator = (Env_Pressure > Constants.HIGH_PRESSURE_THRESHOLD);

**};
```

Figure 26: AADL and AGREE Sensor Subcomponent

The intermediate level subsystem contracts for the Pressure Reactor, Temperature Reactor, and Radiation Reactor specify 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 specifies that the system shuts down only when it should.

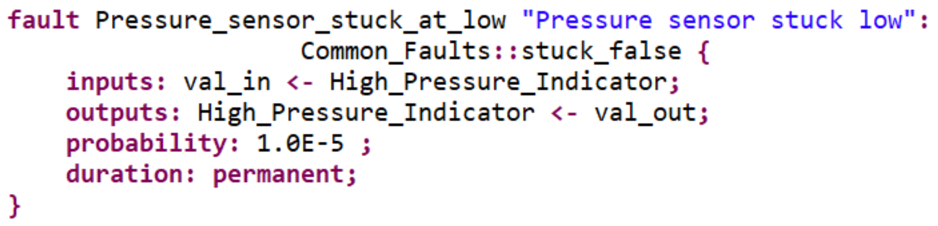


```
system Reactor_Ctrl
    features
        Env_Temp: in data port Integer;
        Env_Pressure: in data port Integer;
        Env_Radiation: in data port Integer;
        Shut_Down_Cmd: out data port Boolean;

    annex agree{**

        guarantee "Shut down when and only when we should":
            Shut_Down_Cmd =
                ((Env_Temp> Constants.HIGH_TEMPERATURE_THRESHOLD)  and
                (Env_Pressure > Constants.HIGH_PRESSURE_THRESHOLD) and
                (Env_Radiation > Constants.HIGH_RADIATION_THRESHOLD));

    **};
end Reactor_Ctrl;
```

Figure 27: Top Level Reactor System and Top Level Property

6.1.3 Safety Model

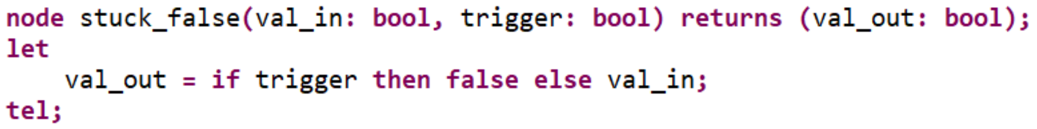
Each sensor can have a failure of getting stuck at indicating high or low. This equates to two possible faults per sensor. A stuck low fault for the pressure sensor is shown in Figure 28.



```
fault Pressure_sensor_stuck_at_low "Pressure sensor stuck low":
                    Common_Faults::stuck_false {
    inputs: val_in <- High_Pressure_Indicator;
    outputs: High_Pressure_Indicator <- val_out;
    probability: 1.0E-5 ;
    duration: permanent;
}
```

Figure 28: 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, e.g., $1.0\ x10^{-6}$ in this example. When a fault occurs, it is assumed to be latching (i.e., permanent). The underlying fault model (*CommonFaults.stuck_false*) is shown in Figure 29.



```
node stuck_false(val_in: bool, trigger: bool) returns (val_out: bool);
let
    val_out = if trigger then false else val_in;
tel;
```

Figure 29: Fault Node Definition for Sensor Fault

For more information on creating fault node definitions, see Section 5.

6.1.4 The 4 types of Analysis Results

Section 4.4.2 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 valid for all layers (i.e., "Verify All Layers" in AGREE analysis).

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

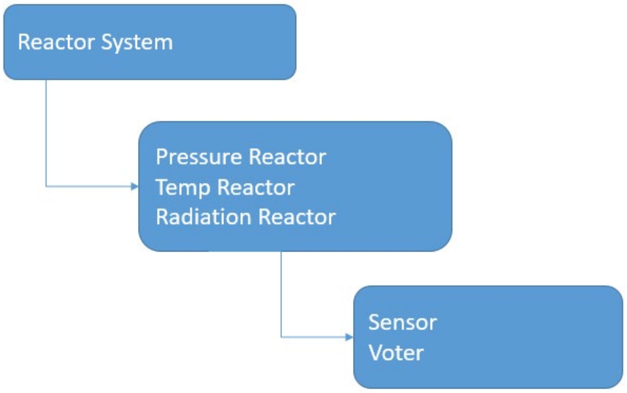


Figure 30: Subcomponent Organization of Sensor System

The top level (Reactor System) has three types of subcomponents (pressure, temperature, and radiation reactors). These three subcomponents each has two types of subcomponents themselves (sensor and voter). When selecting "Verify in the presence of faults", users need to define analysis statement in the implementation of the Reactor System (topmost level), and also the implementations of lower level system (pressure, temperature, and radiation reactor subsystems). In this case, for AGREE verification performed at each layer of the model, the number of faults or probability threshold is restricted per the analysis statement at that level. When selecting "Generate Minimal Cut Sets", users need to define the analysis statement in the implementation of the Reactor system (topmost level). In this case, for AGREE verification performed compositionally, the number of faults or probability threshold is restricted per the analysis statement at the top level.

The example analysis statement given in the system implementation under analysis are shown in Figure 22 and Figure 23. This statement specifies the type of analysis to perform. Only one type (max N or probabilistic) is permitted to be specified in the implementation for a single analysis run. These are illustrated in detail in this section.

6.1.4.1 Verify All Layers with Max N Faults present

Instructions for performing the analysis:

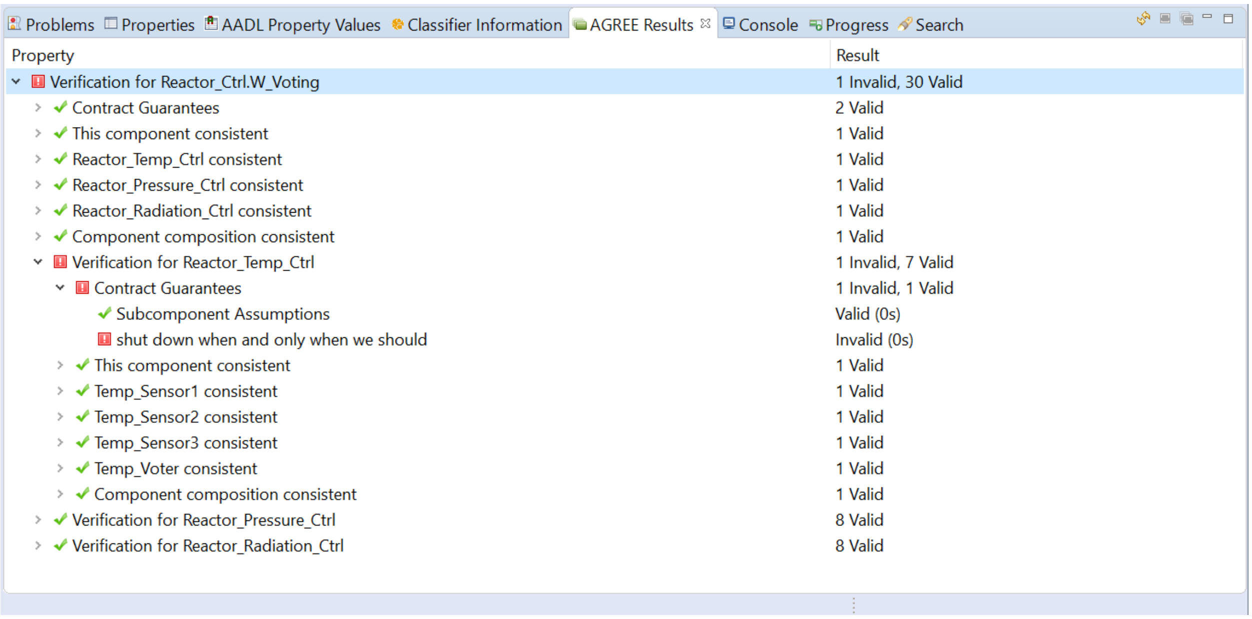
- Place max *n* fault analysis statement in each layer of the AAL system implementation (Reactor_Sys_Ctrl.W_Voting, Reactor_Pressure_Ctrl.W_Voting, Reactor_Temp_Ctrl.W_Voting, Reactor_Radiation_Ctrl.W_Voting). Make sure probabilistic statements are commented out.
- Select the top level system implementation in AADL.
- Select in menu bar: AGREE -> Verify All Layers with Faults.
- Results will be displayed and users can right click invalid property results to see the counterexample.

When max N fault analysis is selected with *Verify All Layers with Faults*, the compositional approach verifies from the top down. Each analysis statement in the implementations must constrain the number of faults active at that level. The top level analysis statement only applies to faults within the top level. In this example, there are no faults at the top level. All faults are located on leaf level components one layer removed from the top.

In this example, the number of active faults at the top level (Reactor_Sys.aadl -> Reactor_Ctrl.W_Voting) to be analyzed is set to *max 1 faults*. At the lower level subsystem implementations (Reactor_Temp.aadl -> Reactor_Temp_Ctrl.W_Voting or Reactor_Pressure.aadl -> Reactor_Pressure_Ctrl.W_Voting), the analysis constraint is set to *max 2 faults*. It can be seen in the analysis results that the top level property proves while the lower level properties are failing.

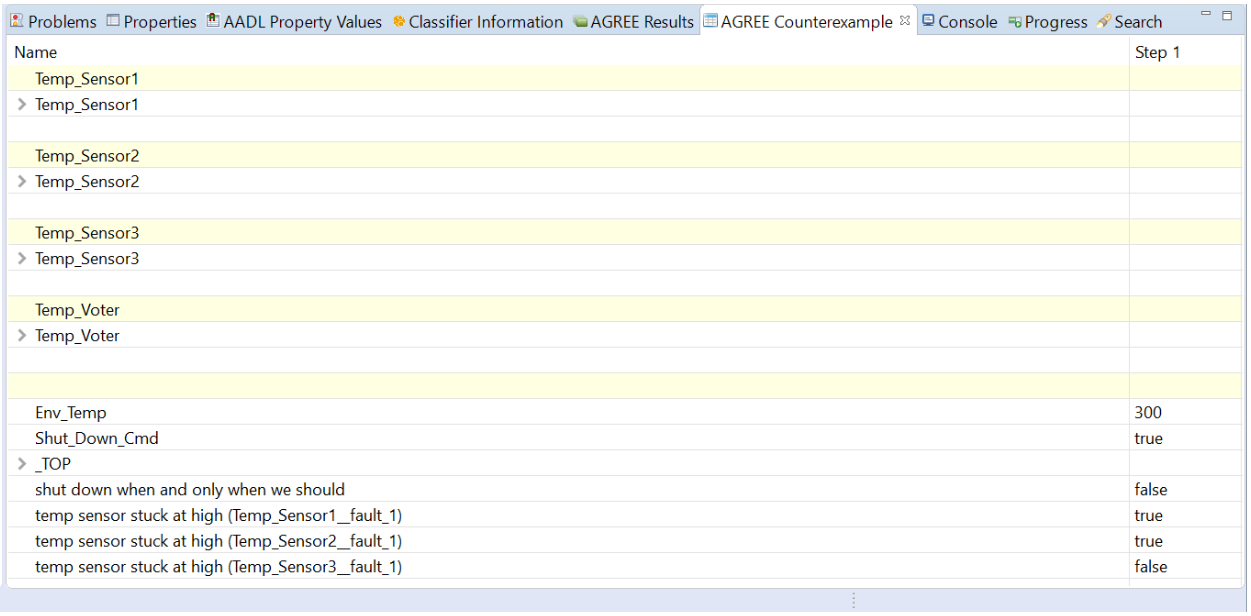
The results are shown in Figure 31. Upon examination of the counterexamples, it is seen that 2 faults are active in these layers which cause the violation of that layer's property. The counterexample in Eclipse view is shown in Figure 32. This counterexample shows that two faults are active in the temperature 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.



Property	Result
🟥 Verification for Reactor_Ctrl.W_Voting	1 Invalid, 30 Valid
✓ Contract Guarantees	2 Valid
✓ This component consistent	1 Valid
✓ Reactor_Temp_Ctrl consistent	1 Valid
✓ Reactor_Pressure_Ctrl consistent	1 Valid
✓ Reactor_Radiation_Ctrl consistent	1 Valid
✓ Component composition consistent	1 Valid
🟥 Verification for Reactor_Temp_Ctrl	1 Invalid, 7 Valid
🟥 Contract Guarantees	1 Invalid, 1 Valid
✓ Subcomponent Assumptions	Valid (0s)
🟥 shut down when and only when we should	Invalid (0s)
✓ This component consistent	1 Valid
✓ Temp_Sensor1 consistent	1 Valid
✓ Temp_Sensor2 consistent	1 Valid
✓ Temp_Sensor3 consistent	1 Valid
✓ Temp_Voter consistent	1 Valid
✓ Component composition consistent	1 Valid
✓ Verification for Reactor_Pressure_Ctrl	8 Valid
✓ Verification for Reactor_Radiation_Ctrl	8 Valid

Figure 31: Verification Results for Max 2 Faults on Sensor Example



Name	Step 1
Temp_Sensor1	
Temp_Sensor1	
Temp_Sensor2	
Temp_Sensor2	
Temp_Sensor3	
Temp_Sensor3	
Temp_Voter	
Temp_Voter	
Env_Temp	300
Shut_Down_Cmd	true
_TOP	
shut down when and only when we should	false
temp sensor stuck at high (Temp_Sensor1__fault_1)	true
temp sensor stuck at high (Temp_Sensor2__fault_1)	true
temp sensor stuck at high (Temp_Sensor3__fault_1)	false

Figure 32: 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.

Dependent faults (HW Faults) are incorporated into this form of analysis.

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.

6.1.4.2 Verify All Layers with Faults Present: Probabilistic Analysis

Instructions for performing the analysis:

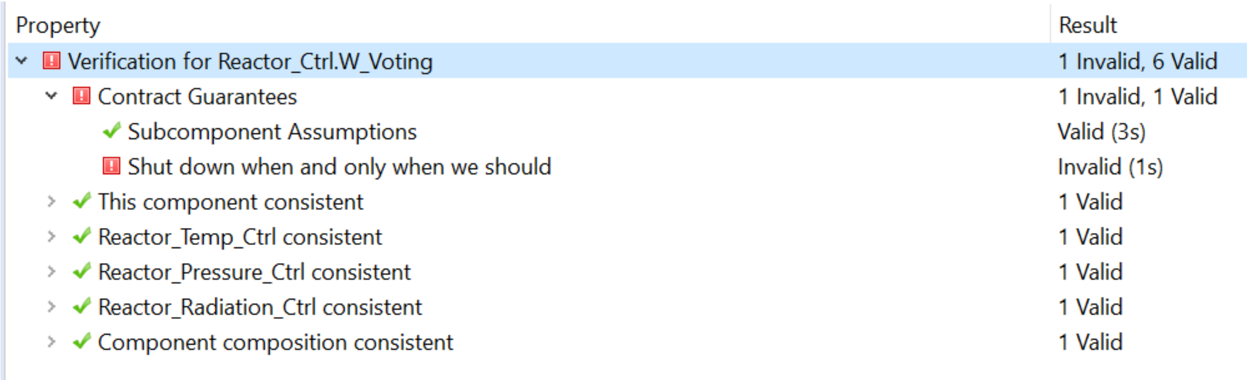
- Place probabilistic statements in the top level AADL system implementation. (Reactor_Sys_Ctrl.W_Voting) Make sure max *n* statement is commented out.
- Select the top level system implementation in AADL.

- Select in menu bar: AGREE -> Verify Monolithically with Faults.
- Results will be displayed and users can right click invalid property results to see the counterexample.

Note: **Probabilistic analysis cannot be run with Verify All Layers with Faults.** In order to run compositional probabilistic analysis, one must Generate Minimal Cut Sets and refer to Section 6.1.4.4.

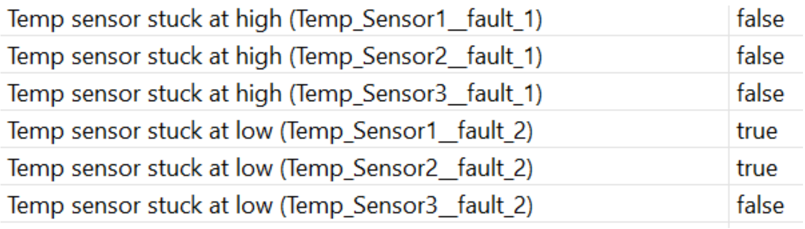
Since the analysis proceeds monolithically, the probability threshold given at the top level is used for the overall system threshold. If there is a combination of faults that exceed the given threshold, the counterexample with show *one* such combination.

In the Sensor example, each fault is assumed to occur with probability $1.0\ x\ 10^{-5}$ and the top level threshold is $1.0\ x\ 10^{-10}$. Upon running monolithic analysis, the top level property fails.



Property	Result
❗ Verification for Reactor_Ctrl.W_Voting	1 Invalid, 6 Valid
❗ Contract Guarantees	1 Invalid, 1 Valid
✓ Subcomponent Assumptions	Valid (3s)
❗ Shut down when and only when we should	Invalid (1s)
✓ This component consistent	1 Valid
✓ Reactor_Temp_Ctrl consistent	1 Valid
✓ Reactor_Pressure_Ctrl consistent	1 Valid
✓ Reactor_Radiation_Ctrl consistent	1 Valid
✓ Component composition consistent	1 Valid

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.



Temp sensor stuck at high (Temp_Sensor1__fault_1)	false
Temp sensor stuck at high (Temp_Sensor2__fault_1)	false
Temp sensor stuck at high (Temp_Sensor3__fault_1)	false
Temp sensor stuck at low (Temp_Sensor1__fault_2)	true
Temp sensor stuck at low (Temp_Sensor2__fault_2)	true
Temp sensor stuck at low (Temp_Sensor3__fault_2)	false

If the user would like to see all such combinations, refer to Section 6.1.4.4.

Dependent faults (HW Faults) are incorporated into this form of analysis.

6.1.4.3 Generate Minimal Cut Sets with Max N Faults

Instructions for performing the analysis:

- Place max *n* analysis statement in top level AADL system implementation and make *n* equal to the max cardinality of the minimal cut sets users would like to see 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 top level system implementation in AADL.
- Select in menu bar: AGREE -> Generate Minimal Cut Sets.
- Files 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.

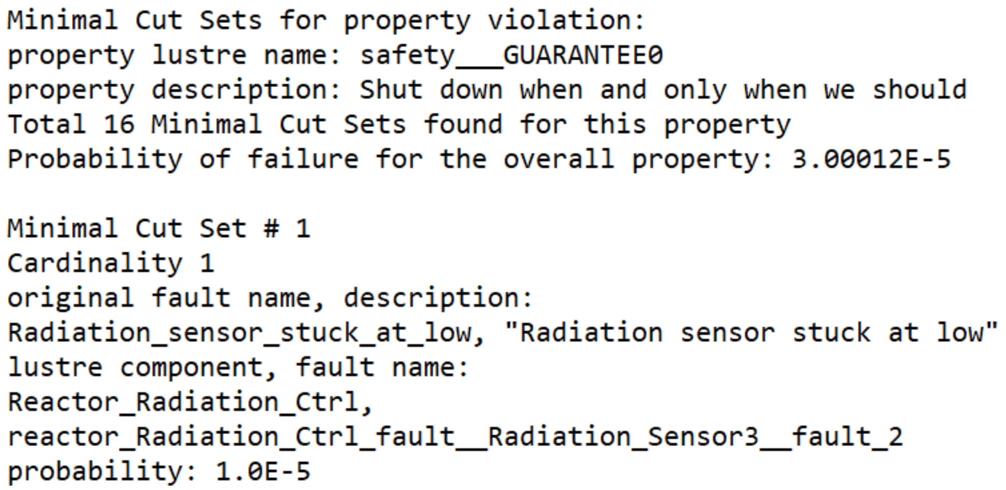
Cardinality 2 produces 15 MinCutSets. These are:

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 16 Minimal Cut Sets found for this property
Probability of failure for the overall property: 3.00012E-5

Minimal Cut Set # 1
Cardinality 1
original fault name, description:
Radiation_sensor_stuck_at_low, "Radiation sensor stuck at low"
lustre component, fault name:
Reactor_Radiation_Ctrl,
reactor_Radiation_Ctrl_fault__Radiation_Sensor3__fault_2
probability: 1.0E-5
```

At the top of the file, the total number of cut sets are displayed (16 in this case). The min cut sets are then displayed textually. The probability of failure for the property is given; this is the calculated probability given the cut sets and assumes independence. The first minimal cut set is shown in the snippet above. It has cardinality 1 (only one fault is contained in this set). That is the radiation sensor fault. When cardinality is greater than one, the faults are displayed within square brackets. The cut sets are shown in order of decreasing probability.

Dependent faults (HW Faults) are *not* incorporated into this form of analysis.

6.1.4.4 Generate Minimal Cut Sets with Probability Threshold

Instructions for performing the analysis:

- Place 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 top level system implementation in AADL.
- Select in menu bar: AGREE -> Generate Minimal Cut Sets.
- Files 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 at or beyond the top level threshold.

Given that each sensor could fail with probability $1.0\ x10^{-5}$ and the top level probability threshold set at $1.0\ x10^{-10}$, 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 at high and stuck at low (6 total) and likewise the combinations for the pressure sensors being stuck both high and low (6 total).

Dependent faults (HW Faults) are *not yet* incorporated into this form of analysis.

6.2 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 1-n output to multiple components. In this section, we show two examples and the mitigation strategy implemented in AGREE.

6.2.1 Asymmetric Fault Implementation

Figure 33 explains our approach to the implementation of Byzantine faults. In this figure, component A has an output, *out1*, that is connected to 4 components (B, C, D, and E). Each of those receiving 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.)

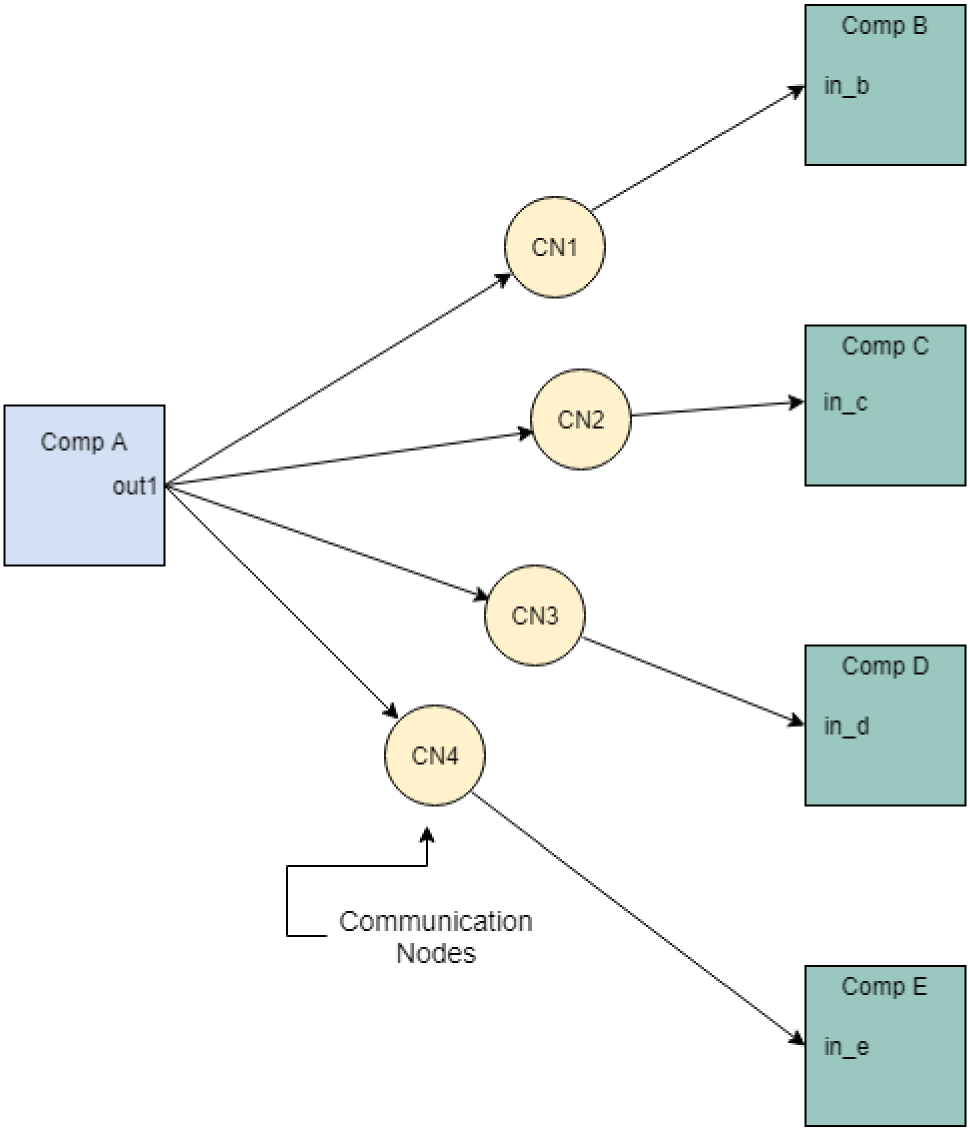


Figure 33: Asymmetric Fault Implementation Strategy

An example 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 such 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.

6.2.2 Mitigation Strategy

In order to mitigate the above described types of faults, mitigation strategies are implemented through AGREE contracts in the behavioral model. Since Byzantine faults may present different symptoms (or message values) to different observers. The objective of the mitigation protocol is for all correct (non-failed) nodes to eventually reach agreement on a value sent by another node. There are *n* nodes, possibly *f* failed nodes. The protocol requires *n > 3f* nodes to handle a single fault. The point is to achieve distributed agreement and coordinated decisions. In this solution, nodes will agree in *f+1* time steps or rounds of communication.

The properties that must be proven on the protocol implementation are as follows:

- All correct (non-failed) 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 predetermined default value.

6.2.3 Color Exchange Example

We created a Color Exchange example 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.

6.2.3.1 Color Exchange Architecture in AADL

A simple diagram of the color exchange architecture is shown in Figure 34. 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 is sent 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.

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.

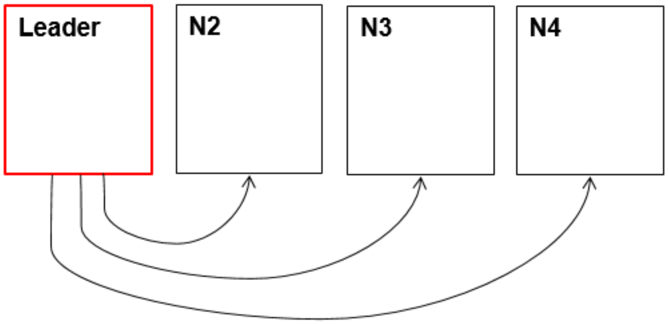
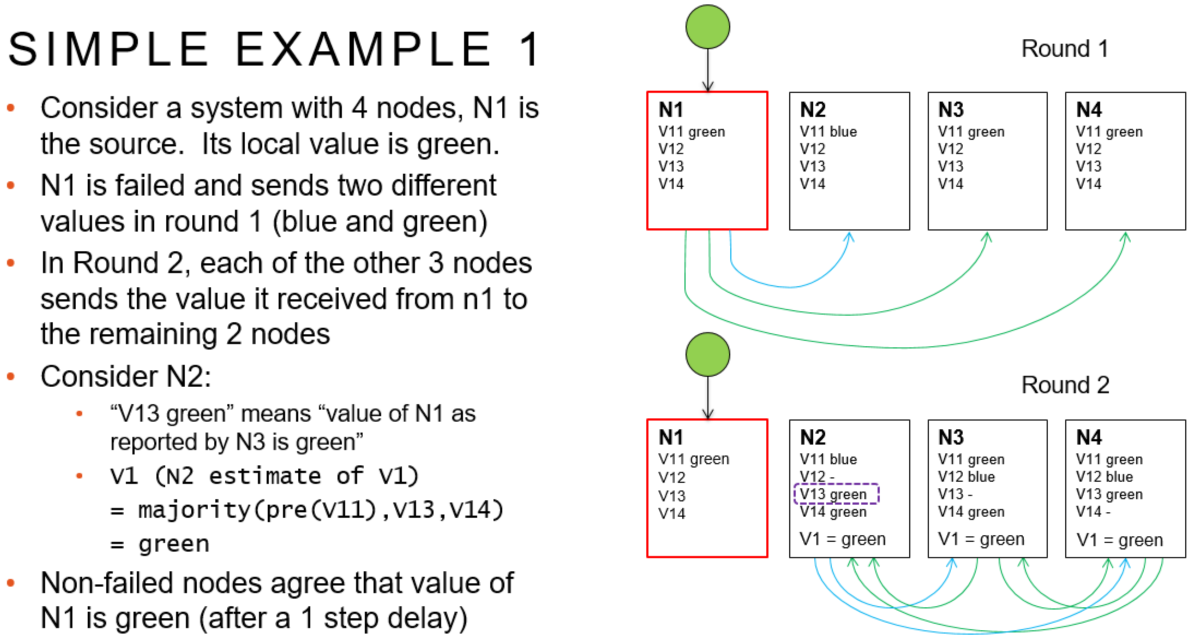


Figure 34: Color Exchange Architecture for Asymmetric Modeling and Mitigation

This model does not include any mitigation and can be explored by accessing the GitHub example repository [3].

6.2.3.2 Color Exchange Mitigation Strategy

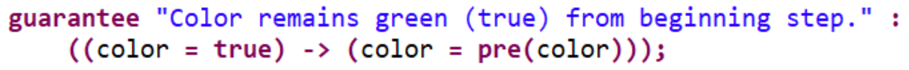
The mitigation strategy will allow a single asymmetric fault to be present (on leader node) and all receiving nodes will agree on a color value at the end of two communication rounds.



SIMPLE EXAMPLE 1

- Consider a system with 4 nodes, N1 is the source. Its local value is green.
- N1 is failed and sends two different values in round 1 (blue and green)
- In Round 2, each of the other 3 nodes sends the value it received from n1 to the remaining 2 nodes
- Consider N2:
  - “V13 green” means “value of N1 as reported by N3 is green”
  - `V1 (N2 estimate of V1)`
    `= majority(pre(V11),V13,V14)`
    `= green`
- Non-failed nodes agree that value of N1 is green (after a 1 step delay)

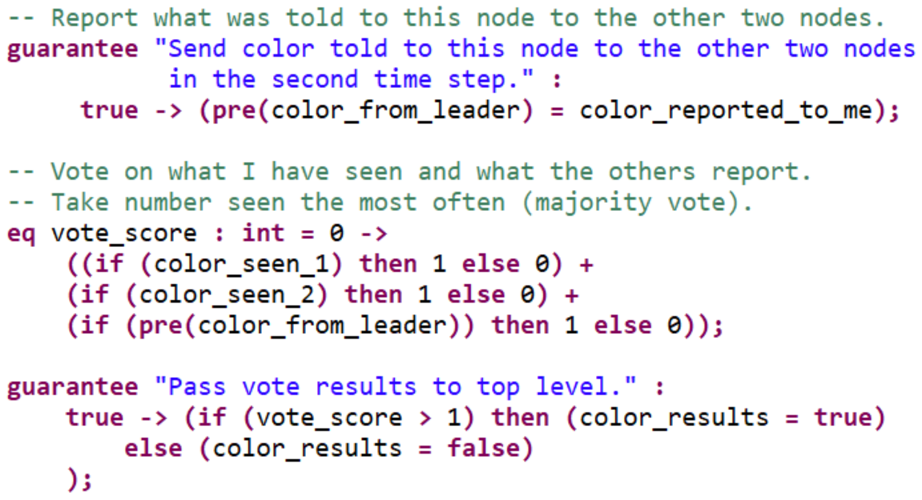
The mitigation in AGREE is shown in the following contracts. The leader node behavior is such that it's color in the nominal case is always green (true). This occurs from the first step onward.



```
guarantee "Color remains green (true) from beginning step." :
    ((color = true) -> (color = pre(color)));
```

The receiving nodes have additional ports between them in order to share what they received in the first step. These are all one to many connections from the 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.

The following contracts define the behavior over those ports. The first guarantee specifies that each node outputs what they saw from the leader in the previous step to the other receiving nodes. The second guarantee specifies that the behavior of what each node reports to the top level (majority voting on what everyone saw).

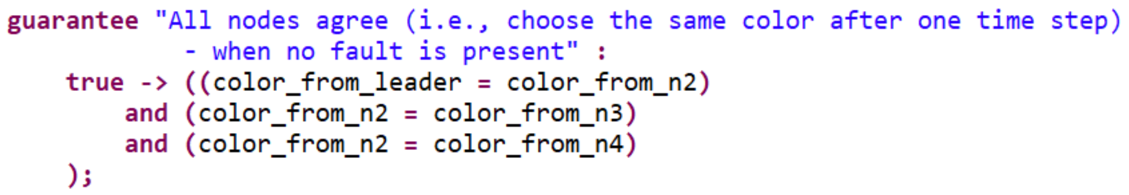


```
-- Report what was told to this node to the other two nodes.
guarantee "Send color told to this node to the other two nodes
           in the second time step." :
     true -> (pre(color_from_leader) = color_reported_to_me);

-- Vote on what I have seen and what the others report.
-- Take number seen the most often (majority vote).
eq vote_score : int = 0 ->
    ((if (color_seen_1) then 1 else 0) +
    (if (color_seen_2) then 1 else 0) +
    (if (pre(color_from_leader)) then 1 else 0));

guarantee "Pass vote results to top level." :
    true -> (if (vote_score > 1) then (color_results = true)
        else (color_results = false)
    );
```

The contracts at the top level cover both cases.

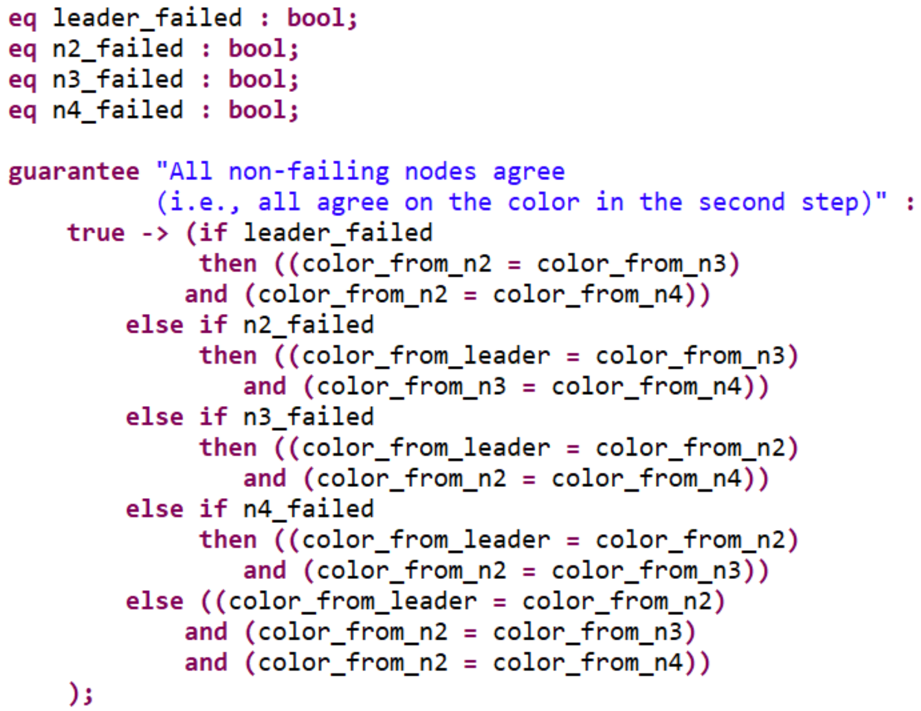
- Case 1: If the source node is a correct, all other correct nodes agree on the value that was originally sent by the source.
- Case 2: If the source node is failed, all other nodes must agree on some (somewhat arbitrary) value.

These correspond to two guarantees.



```
guarantee "All nodes agree (i.e., choose the same color after one time step)
            - when no fault is present" :
    true -> ((color_from_leader = color_from_n2)
        and (color_from_n2 = color_from_n3)
        and (color_from_n2 = color_from_n4)
    );
```

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



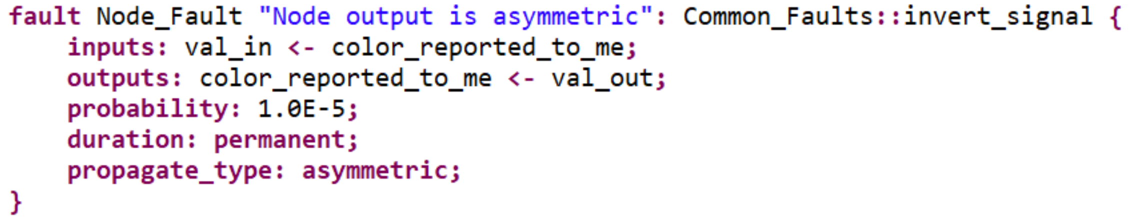
```
eq leader_failed : bool;
eq n2_failed : bool;
eq n3_failed : bool;
eq n4_failed : bool;

guarantee "All non-failing nodes agree
          (i.e., all agree on the color in the second step)" :
    true -> (if leader_failed
             then ((color_from_n2 = color_from_n3)
            and (color_from_n2 = color_from_n4))
        else if n2_failed
             then ((color_from_leader = color_from_n3)
                and (color_from_n3 = color_from_n4))
        else if n3_failed
             then ((color_from_leader = color_from_n2)
                and (color_from_n2 = color_from_n4))
        else if n4_failed
             then ((color_from_leader = color_from_n2)
                and (color_from_n2 = color_from_n3))
        else ((color_from_leader = color_from_n2)
            and (color_from_n2 = color_from_n3)
            and (color_from_n2 = color_from_n4))
    );
```

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.

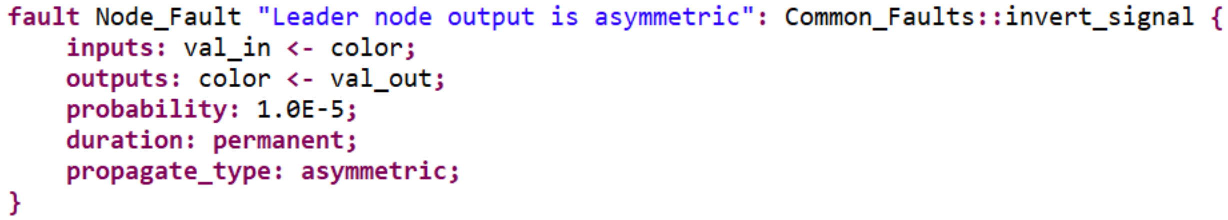
6.2.3.3 Color Exchange Fault Model and Analysis

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.



```
fault Node_Fault "Node output is asymmetric": Common_Faults::invert_signal {
    inputs: val_in <- color_reported_to_me;
    outputs: color_reported_to_me <- val_out;
    probability: 1.0E-5;
    duration: permanent;
    propagate_type: asymmetric;
}
```



```
fault Node_Fault "Leader node output is asymmetric": Common_Faults::invert_signal {
    inputs: val_in <- color;
    outputs: color <- val_out;
    probability: 1.0E-5;
    duration: permanent;
    propagate_type: asymmetric;
}
```

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* [3].

6.2.4 Process ID Example

We now extend the concept of "leader node" to all nodes. In this example, the Boolean type of the last small model is extended to integers and each node passes its own process id (PID) to the other three nodes.

Note: The example is titled *PIDByzantineAgreement* and is found in GitHub [3].

6.2.4.1 PID Example AADL Architecture

The basic architecture is shown in Figure 35. 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.

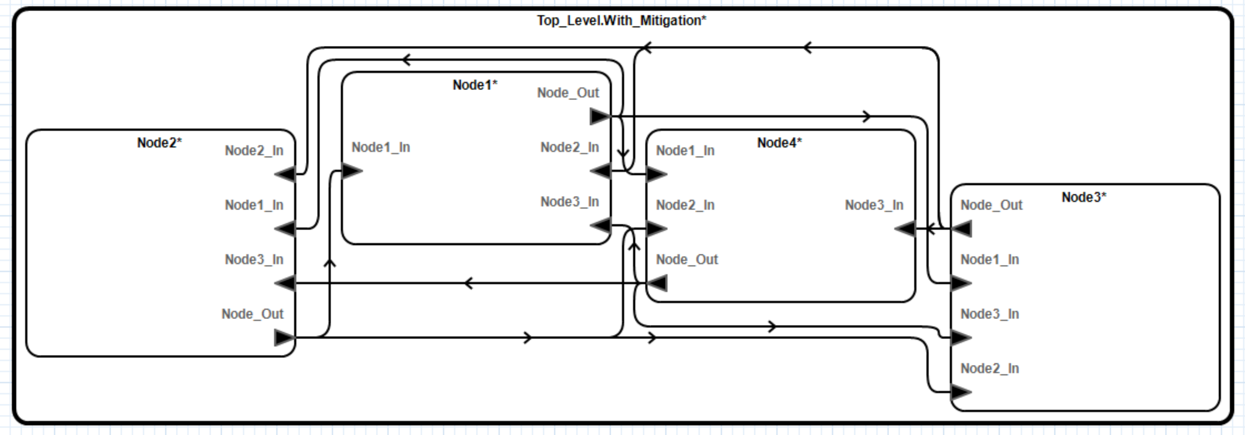
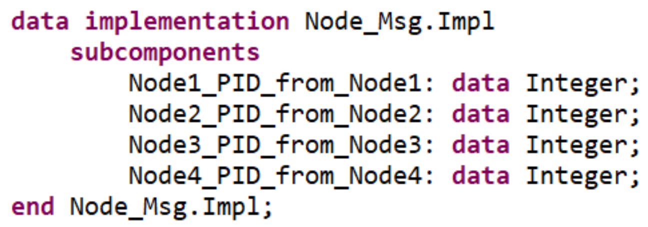


Figure 35: PID Example Architecture

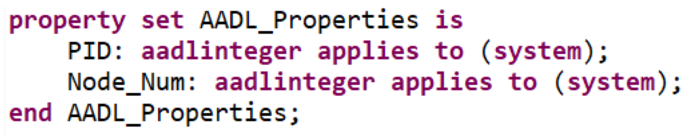
The output of the nodes (Node_Out) is a data port that passes a Node_Msg data structure. This structure has 4 fields corresponding to a specific node and its PID message.



```
data implementation Node_Msg.Impl
    subcomponents
        Node1_PID_from_Node1: data Integer;
        Node2_PID_from_Node2: data Integer;
        Node3_PID_from_Node3: data Integer;
        Node4_PID_from_Node4: data Integer;
end Node_Msg.Impl;
```

Likewise, the inputs of a node (Nodei_In) carry this same message.

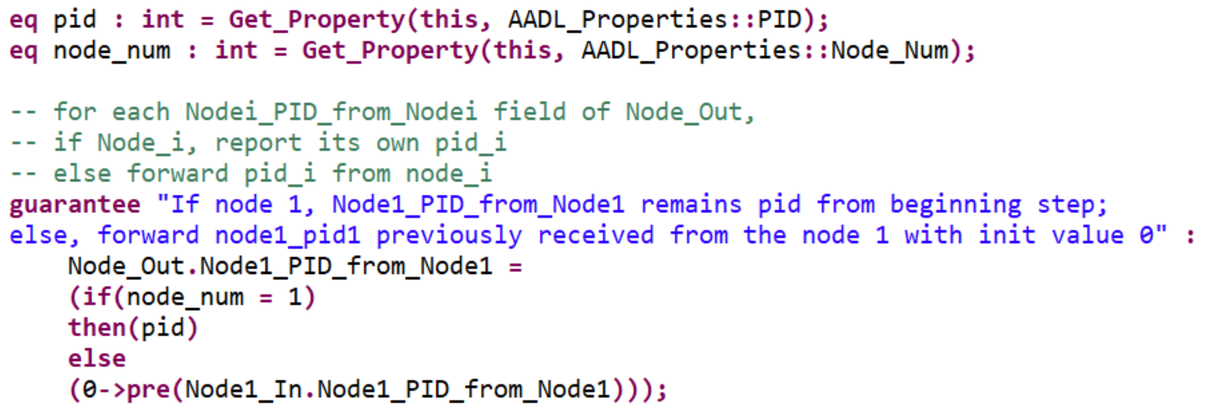
Each node has a PID and a number. These are encoded in AADL as Properties.



```
property set AADL_Properties is
    PID: aadlinteger applies to (system);
    Node_Num: aadlinteger applies to (system);
end AADL_Properties;
```

6.2.4.2 PID Example AGREE Behavioral Model with Mitigation Strategy

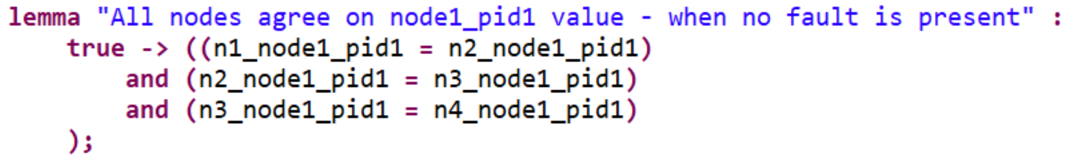
Similar to the color example, the nodes pass their PID (a unique value assigned to each node as an AADL Property) from the first step onward. The node's generic behavioral model specifies the behaviors per node number. We show one of the four contracts that support each node's behavior.



```
eq pid : int = Get_Property(this, AADL_Properties::PID);
eq node_num : int = Get_Property(this, AADL_Properties::Node_Num);

-- for each Nodei_PID_from_Nodei field of Node_Out,
-- if Node_i, report its own pid_i
-- else forward pid_i from node_i
guarantee "If node 1, Node1_PID_from_Node1 remains pid from beginning step;
else, forward node1_pid1 previously received from the node 1 with init value 0" :
    Node_Out.Node1_PID_from_Node1 =
    (if(node_num = 1)
    then(pid)
    else
    (0->pre(Node1_In.Node1_PID_from_Node1)));
```

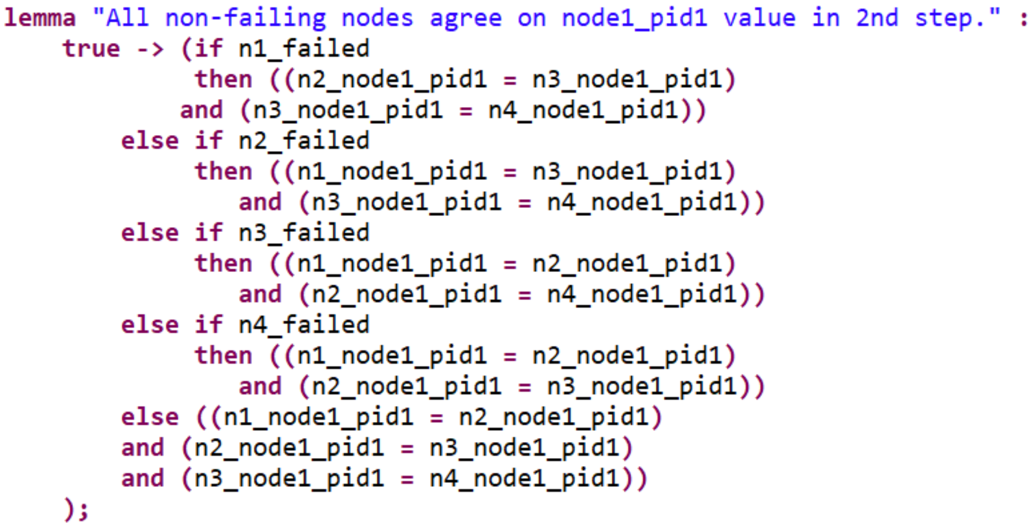
In the above contract, it is stated that any of the instantiated nodes will respond by either passing its own PID (if node 1) or passing the previously seen value *from* node 1.

The top level contracts mirror the color example. There are two types of contracts: 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 node's PID value, thus there are 4 contracts, one for each node PID.



```
lemma "All nodes agree on node1_pid1 value - when no fault is present" :
    true -> ((n1_node1_pid1 = n2_node1_pid1)
        and (n2_node1_pid1 = n3_node1_pid1)
        and (n3_node1_pid1 = n4_node1_pid1)
    );
```

The second type of contracts in the top level are:



```
lemma "All non-failing nodes agree on node1_pid1 value in 2nd step." :
    true -> (if n1_failed
                 then ((n2_node1_pid1 = n3_node1_pid1)
                and (n3_node1_pid1 = n4_node1_pid1))
        else if n2_failed
                 then ((n1_node1_pid1 = n3_node1_pid1)
                    and (n3_node1_pid1 = n4_node1_pid1))
        else if n3_failed
                 then ((n1_node1_pid1 = n2_node1_pid1)
                    and (n2_node1_pid1 = n4_node1_pid1))
        else if n4_failed
                 then ((n1_node1_pid1 = n2_node1_pid1)
                    and (n2_node1_pid1 = n3_node1_pid1))
        else ((n1_node1_pid1 = n2_node1_pid1)
        and (n2_node1_pid1 = n3_node1_pid1)
        and (n3_node1_pid1 = n4_node1_pid1))
    );
```

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 4.4.3.

6.2.4.3 PID Example Fault Model

Since each node has multiple 1-n outputs, a fault is defined for each one. To illustrate this, look at one node and its outputs below.

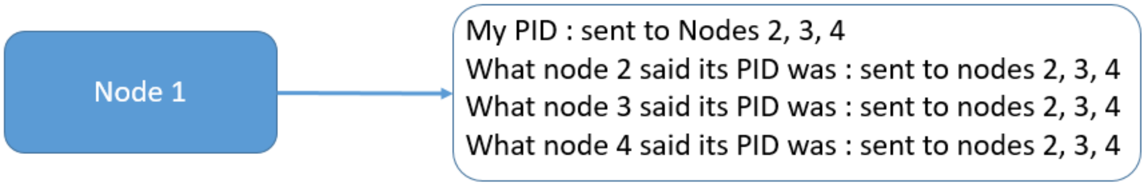
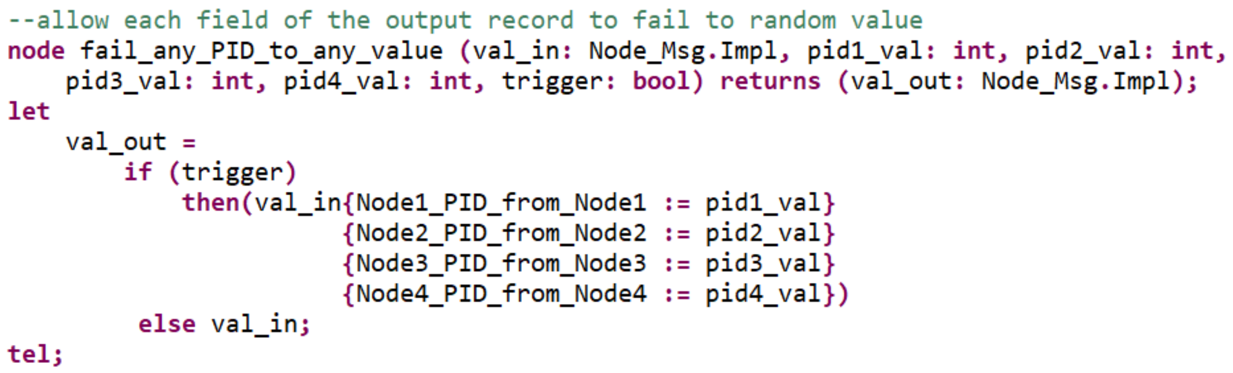

Node 1
My PID : sent to Nodes 2, 3, 4
What node 2 said its PID was : sent to nodes 2, 3, 4
What node 3 said its PID was : sent to nodes 2, 3, 4
What node 4 said its PID was : sent to nodes 2, 3, 4

Figure 36: Outputs for Node in PID Example

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

```
fault Asym_Fail_Any_PID_To_Any_Val "Node output is asymmetric" :
                          Common_Faults::fail_any_PID_to_any_value {
              eq pid1_val : real;
              eq pid2_val : real;
              eq pid3_val : real;
              eq pid4_val : real;
              inputs: val_in <- Node_Out,
                      pid1_val <- pid1_val,
                      pid2_val <- pid2_val,
                     pid3_val <- pid3_val,
                     pid4_val <- pid4_val;
              outputs: Node_Out <- val_out;
              duration: permanent;
              propagate_type: asymmetric;
}
```

And the fault node definition (fail_any_pid_to_any_value) is:



```
--allow each field of the output record to fail to random value
node fail_any_PID_to_any_value (val_in: Node_Msg.Impl, pid1_val: int, pid2_val: int,
    pid3_val: int, pid4_val: int, trigger: bool) returns (val_out: Node_Msg.Impl);
let
    val_out =
        if (trigger)
            then(val_in{Node1_PID_from_Node1 := pid1_val}
                       {Node2_PID_from_Node2 := pid2_val}
                       {Node3_PID_from_Node3 := pid3_val}
                       {Node4_PID_from_Node4 := pid4_val})
         else val_in;
tel;
```

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.

6.2.4.4 PID Example Analysis Results

Verification can be performed using max n fault analysis statement at the top level AADL system implementation and *Verify All Layers with Faults* or by *Generating Minimal Cut Sets* with cardinality less than or equal to n.

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* [3].

7 Other Information

This section collects information that may in some cases be useful to the users for their modeling and verification work with Safety Annex.

7.1 Defining Multiple Faults on a Single Output

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

- Verify X with Faults:
  All faults are considered as model elements and the model checker will present one scenario where the faults cause the violation of the properties in the counterexamples.
- Generate Minimal Cut Sets:
  All combinations that contribute to property violation will be displayed in the min cut sets.

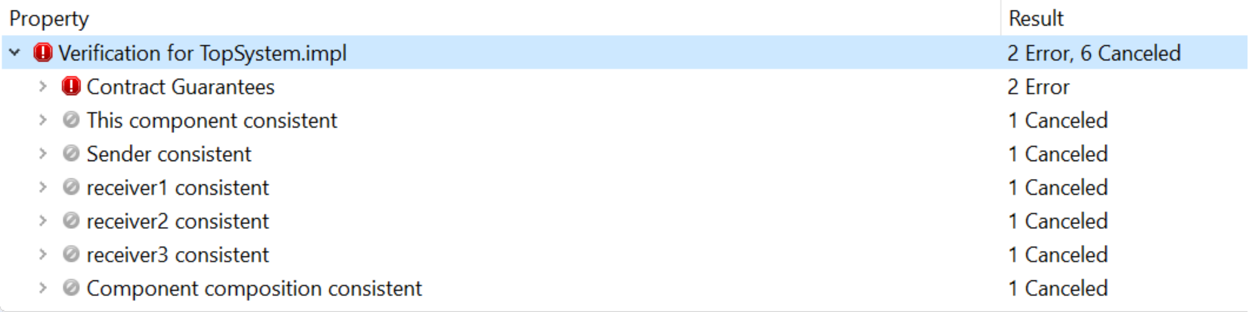
Users should be aware that multiple fault definitions on the same output should have the same return types. When the output of a component is a nested data type, the only supported way is to define faults on the whole nested data type instead of on individual fields of the data type (see Section 6.2.4.3).

```
annex safety {**
        fault Receiver_Fault_1 "Receiver output fail to zero" : Fault_Library::fail_to_real {
                eq test : real ;
                inputs: val_in <- output.NODE_VAL, alt_val <- test;
                outputs: output.NODE_VAL <- val_out;
                duration: permanent;
        }

        fault Receiver_Fault_2 "Datatype output fail to zero" : Fault_Library::nested_fault {
                inputs: val_in <- output, alt_val <- 0.0;
                outputs: output <- val_out;
                duration: permanent;
        }

**};
```

Figure 37: Different Return Types on Single Output

7.2 Lustre Error

If a Lustre error should ever appear when running fault analysis (see Figure 38), it is an indication that the model was not constructed correctly and the translated Lustre code is not supported by the backend JKind model checker. Please contact the authors of this guide and we can assist you in locating the problem.



Property	Result
Verification for TopSystem.impl	2 Error, 6 Canceled
Contract Guarantees	2 Error
This component consistent	1 Canceled
Sender consistent	1 Canceled
receiver1 consistent	1 Canceled
receiver2 consistent	1 Canceled
receiver3 consistent	1 Canceled
Component composition consistent	1 Canceled

Figure 38: Lustre Error Output

8 Appendices

8.1 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 user's convenience. Create a new AADL package within the project. Name the new package to user's desire (often it is called "faults" or "common_faults") and then copy the following code into the body of the package. Users can also add new fault node definitions or modify existing ones to create their own customized fault library.

```
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;

**};
```

9 References

[1] AGREE Users Guide
https://github.com/loonwerks/formal-methods-workbench/tree/master/documentation/agree

[2] AMASE GitHub Repository
https://github.com/loonwerks/AMASE/tree/master

[3] AMASE Examples
https://github.com/loonwerks/AMASE/tree/master/examples