Safety Annex Users Guide

Version 0.5

Danielle Stewart, University of Minnesota

Jing (Janet) Liu, Collins Aerospace

Darren Cofer, Collins Aerospace  
Mike Whalen, University of Minnesota

Mats Heimdahl, University of Minnesota

Version History

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

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

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

Statement describing asymmetric faults.

# Table of Contents

[1 Table of Contents 3](#_Toc11676039)

[2 Table of Figures 5](#_Toc11676040)

[3 Introduction 6](#_Toc11676041)

[4 Brief Overview of AADL, AGREE, and the Safety Annex 7](#_Toc11676042)

[4.1 Using the Safety Annex AADL Plugin 10](#_Toc11676043)

[5 Safety Annex Language 18](#_Toc11676044)

[5.1 Syntax Overview 18](#_Toc11676045)

[5.2 Lexical Elements and Types 19](#_Toc11676046)

[5.3 Subclauses 20](#_Toc11676047)

[5.4 Spec Statement 21](#_Toc11676048)

[5.4.1 Fault Statement 22](#_Toc11676049)

[5.4.1.1 Input Statement 22](#_Toc11676050)

[5.4.1.2 Output Statement 23](#_Toc11676051)

[5.4.1.3 Duration Statement 23](#_Toc11676052)

[5.4.1.4 Trigger Statement 24](#_Toc11676053)

[5.4.1.5 Probability Statement 24](#_Toc11676054)

[5.4.1.6 Propagate Type Statement 24](#_Toc11676055)

[5.4.1.7 Propagate From Statement 25](#_Toc11676056)

[5.4.1.8 Safety Equation Statements 25](#_Toc11676057)

[5.4.1.8.1 Eq Statements 25](#_Toc11676058)

[5.4.1.8.2 Set Statements 26](#_Toc11676059)

[5.4.1.8.3 Range Statements 26](#_Toc11676060)

[5.4.1.8.4 Interval Statements 26](#_Toc11676061)

[5.4.2 Analysis Statement 28](#_Toc11676062)

[5.4.2.1 Max N Faults Analysis 28](#_Toc11676063)

[5.4.2.2 Probabilistic Analysis 28](#_Toc11676064)

[5.4.3 Hardware Fault Statement 28](#_Toc11676065)

[5.4.3.1 Duration 29](#_Toc11676066)

[5.4.3.2 Probability 29](#_Toc11676067)

[5.4.3.3 Propagation Type 29](#_Toc11676068)

[6 The Tool Suite (Safety Annex, AGREE, AADL) 30](#_Toc11676069)

[6.1 Tool Suite Overview 30](#_Toc11676070)

[6.2 Installation 31](#_Toc11676071)

[6.2.1 Install OSATE 31](#_Toc11676072)

[6.2.2 Install Safety Annex 32](#_Toc11676073)

[6.2.3 Install SMT Solver 34](#_Toc11676074)

[6.2.4 Set AGREE Analysis Preferences 36](#_Toc11676075)

[6.3 Development Environment Installation 37](#_Toc11676076)

[6.3.1 Install OSATE Development Environment 37](#_Toc11676077)

[6.3.2 Download Safety Annex Source Code 38](#_Toc11676078)

[6.3.3 Github Branches 38](#_Toc11676079)

[6.3.4 Run OSATE 38](#_Toc11676080)

# Table of Figures

[Figure 1: Toy Example for Safety Annex and AGREE 7](#_Toc11676081)

[Figure 2: AADL Code for Toy Example with AGREE and Safety Annexes 9](#_Toc11676082)

[Figure 3: Fault Hypothesis Example 10](#_Toc11676083)

[Figure 4: Import Menu Option 11](#_Toc11676084)

[Figure 5: Importing Toy Example Project 12](#_Toc11676085)

[Figure 6: Workspace After Importing Toy Example 13](#_Toc11676086)

[Figure 7: AGREE and Safety Analysis Dropdown Menu 14](#_Toc11676087)

[Figure 8:AGREE Verification Results 14](#_Toc11676088)

[Figure 9: Counterexample from Safety Analysis 16](#_Toc11676089)

[Figure 10: Generated Excel File for Counterexample 17](#_Toc11676090)

[Figure 11: Medical Device Example 20](#_Toc11676091)

[Figure 12: Safety Annex Grammar 21](#_Toc11676092)

[Figure 13: Fault Node Definition 22](#_Toc11676093)

[Figure 14: Fault Statement With Asymmetric Propagate Type 24](#_Toc11676094)

[Figure 15: Propagation Statement Example 25](#_Toc11676095)

[Figure 16: Max One Fault Example 28](#_Toc11676096)

[Figure 17: Probability Threshold Example 28](#_Toc11676097)

[Figure 18: Hardware Fault Statement 29](#_Toc11676098)

[Figure 19: Overview of Safety Annex/AGREE/OSATE Tool Suite 31](#_Toc11676099)

[Figure 20: OSATE Loading Screen 32](#_Toc11676100)

[Figure 21: Safety Analysis Menu Item 33](#_Toc11676101)

[Figure 22: Windows 10 System Control Panel 34](#_Toc11676102)

[Figure 23: System Properties Dialog Box 35](#_Toc11676103)

[Figure 24: Environment Variables Dialog Box 35](#_Toc11676104)

[Figure 25: System Variable Text Edit Box 36](#_Toc11676105)

[Figure 26: AGREE Analysis Preferences 37](#_Toc11676106)

# Introduction

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

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

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

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

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

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

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

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



Figure 1: Toy Example for Safety Annex and AGREE

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

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

**package** Integer\_Toy

**public**

**with** Base\_Types;

**with** faults;

**system** A

**features**

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

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

**annex** agree {\*\*

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

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

\*\*};

**annex** safety {\*\*

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

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

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

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

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

**}**

\*\*};

**end** A ;

**system** B

**features**

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

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

**annex** agree {\*\*

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

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

\*\*};

**annex** safety {\*\*

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

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

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

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

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

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

**}**

\*\*};

**end** B ;

**system** C

**features**

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

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

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

**annex** agree {\*\*

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

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

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

\*\*};

**end** C ;

**system** top\_level

**features**

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

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

**annex** agree {\*\*

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

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

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

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

\*\*};

**end** top\_level;

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

**subcomponents**

A\_sub : **system** A ;

B\_sub : **system** B ;

C\_sub : **system** C ;

**connections**

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

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

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

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

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

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

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

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

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

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

**annex** agree{\*\*

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

\*\*};

**annex** safety{\*\*

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

--analyze : max 1 fault

\*\*};

**end** top\_level.Impl;

toy1

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

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

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

**Note:** Top level analysis can be performed only within a system implementation.

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.

**Important Note: At this time, fault hypotheses (see Figure 3) must be added to each layer of a system in order for analysis to proceed correctly. Also, at most one of the analysis statements must be present. In the example shown in Figure 3, the maximum *n* faults analysis statement is commented out. In this case, the probabilistic analysis will be run.**



Figure 3: Fault Hypothesis Example

## Using the Safety Annex AADL Plugin

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

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



Figure 4: Import Menu Option

Then choosing "Existing Project into Workspace."



Figure 5: Importing Toy Example Project

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



Figure 6: Workspace After Importing Toy Example

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



Figure 7: AGREE and Safety Analysis Dropdown Menu

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



Figure 8:AGREE Verification Results

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

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

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

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



Figure 9: Counterexample from Safety Analysis

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

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

The .xls file type should now be selected.

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

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



Figure 10: Generated Excel File for Counterexample

# Safety Annex Language

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

## Syntax Overview

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

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

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

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

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* as well. Safety syntax can also be placed inside of annexes in a *component implementation*. This annex links directly to the AGREE annex also associated with the component in question.

## Lexical Elements and Types

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

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

-- Here is a comment.

-- a long comment may be split onto

-- two or more consecutive lines

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

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

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

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

digit ::= (0..9)

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

Boolean and numeric literal values are defined as follows:

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

Integer\_literal ::= decimal\_integer\_literal

Real\_literal ::= decimal\_real\_literal

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

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

numeral ::= digit\*

Boolean\_literal are: true, false.

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

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

String elements are defined with the following syntax:

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

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

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

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

**data** Alarm\_Outputs

**end** Alarm\_Outputs;

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

**subcomponents**

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

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

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

**end** Alarm\_Outputs.Impl;

**placeholder**

Figure 11: Medical Device Example

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

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

## Subclauses

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

**annex** safety {\*\*

-- safety spec statements here...

\*\*};

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

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

| 'analyze' ':' AnalysisBehavior

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

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

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

AnalysisBehavior: 'max' Int\_Literal 'fault'

| 'probability' Real\_Literal

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

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

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

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

| 'enabled' ':' TriggerCondition ';'

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

| SafetyEqStatement

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

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

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

PropagationTypeConstraint: 'asymmetric'

| 'symmetric'

TemporalConstraint: 'permanent'

| 'transient'

TriggerCondition: 'must' '{' Expr ("," Expr)\* '}'

| 'enabler' '{' Expr ("," Expr)\* '}'

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

| 'interval' ID '=' Interval ';'

| 'set' ID '=' '{'INTEGER\_LIT (',' INTEGER\_LIT)\* '}' ';'

Interval: '[' Expr ',' Expr ']')

| '(' Expr ',' Expr']')

| '[' Expr ',' Expr ')')

| '(' Expr ',' Expr ')')

**placeholder**

Figure 12: Safety Annex Grammar

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

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

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

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

**system** A

**features**

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

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

**annex** agree {\*\*

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

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

\*\*};

**annex** safety {\*\*

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

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

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

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

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

**}**

\*\*};

**end** A ;

Figure 13: Toy Example System A

## Spec Statement

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

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

| 'analyze' ':' AnalysisBehavior

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

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

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

A simple example is :

**annex** safety {\*\*

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

**...**

**}**

\*\*};

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

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

### Fault Statement

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

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

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

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

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

| 'enabled' ':' TriggerCondition ';'

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

| SafetyEqStatement

Each *fault* spec statement corresponds with one fault node definition that will wrap the output of a single AADL component. Continuing with the Toy Example, we refer to the fault subcomponents located in the fault definition given in Figure 15.

**annex** safety {\*\*

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

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

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

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

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

**}**

\*\*};

Figure 14: Toy Example System A Safety Annex

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

****

Figure 15: Fault Node Definition

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

#### Input Statement

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

Notice the “**input**” keyword in Figure 15 . 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 17.



Figure 16: Fault node wrapping output of AADL component

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

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

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

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

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

#### Output Statement

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

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

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

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

#### Probability Statement

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

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

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

#### Duration Statement

A duration statement specifies whether the fault will be transient\* or 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.

#### Trigger Statement

The safety annex currently does not support trigger statements. This will be implemented in future work. The following is a description of what trigger statements will look like once implemented.

There are two types of triggers that can occur within a model. We call these *must* triggers and *enabler* triggers. The *must* triggers are of the form: if the trigger has occurred, then the fault must have been activated. The *enabler* triggers are of the form: if the trigger has occurred, then the fault may have been activated. In either of these cases, the triggers are specified using a list or a series of disjunctions. The trigger may have a probability associated with it. This probabilistic value is an optional piece of the statement. The following is a simplified grammar of the trigger statement syntax:

TriggerCondition: 'must' '{' Expr ("," Expr)\* '}'

| 'enabler' '{' Expr ("," Expr)\* '}'

#### Propagate-Type Statement

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

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



Figure 17: Sender-Receiver AADL Model

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

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

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



Figure 18: Fault Statement With Asymmetric Propagate Type

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

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

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

#### Propagate-From Statement

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

****

Figure 19: Propagation Statement Example

#### Safety Equation Statements

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

##### Eq Statements

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

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

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

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

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

**annex** safety {\*\*

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

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

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

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

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

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

**}**

\*\*};

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

##### Set Statements

Set equation statements are currently not supported in the safety annex. They will be implemented in future work.

Set equation statements can be used to specify a set of discrete values of nondeterminism.

**set** set\_values **= {**0**,**3**,**5**} ;**

The variable set\_values will hold the value 0, 3, or 5.

##### Range Statements

Range equation statements are currently not supported in the safety annex. They will be implemented in future work.

A range equation statement is used to specify a discrete range of integer values. This is the same idea as a set, but is used when the size of the set gets to be unwieldy.

**range** range\_values **= {**0**,**50**} ;**

The variable range\_values will hold some integer value from 0 to 50 inclusive.

##### Interval Statements

Interval equation statements are currently not supported in the safety annex. They will be implemented in future work.

Interval equation statements can be used to specify a real interval of nondeterminism for some variable. These intervals can be any combination of open, closed, or neither. The following is an example of a range including -1.2 up to (but not including) 30.0.

**interval** input\_values **=** [-1.2, 30.0)**;**

### Analysis Statement

An analysis statement is given at the top level of the system implementation under analysis. This is shown in Figure 20 and Figure 21. It will specify 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 below.

**Important Note: At this time, fault hypotheses (see Figure 3) must be added to each layer of a system in order for analysis to proceed correctly. Also, at most one of the analysis statements must be present.**

#### Max N Faults Analysis

As shown in Figure 15, the user can specify the maximum number of active faults in a system. In this way, it can be determined if the system is resilient to a certain number of faults.

**annex** safety{\*\*

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

\*\*};

Figure 20: Max One Fault Example

There are three options for max N fault analysis:

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

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

#### Probabilistic Analysis

In order for the probabilistic analysis to run, probabilities must be assigned to each fault definition as shown in the Toy Example of Figure 2. The syntax of probabilistic analysis is shown in Figure 16 with a top level threshold of 1.0E-7.

**annex** safety{\*\*

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

\*\*};

Figure 21: Probability Threshold Example

There are two options for probabilistic analysis.

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

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

### 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 new fault model element for HW components. In comparison to the basic fault statement, users are not specifying behavioral effects for the HW failures, nor data ports to apply the failure. An example of a model component fault declaration is shown in Figure 22. This example is taken from the Wheel Brake System model found in <https://github.com/loonwerks/AMASE/tree/develop/examples>. The grammar for a hardware fault subcomponent is shown below.

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

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

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

The example shows the failure of a hardware valve component.



Figure 22: Hardware Fault Statement

In addition, users can specify fault dependencies outside of fault statements using propagation statements. For more information on propagation type statements, see section 5.4.1.6.

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

For descriptions of these, please refer to sections 5.4.1.4, 5.4.1.3, and 5.4.1.6. The syntax and explanations are identical in the hardware case.

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

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

## Tool Suite Overview

Figure 7 shows an overview of the AGREE/OSATE tool suite. As presented in the figure, OSATE is an Eclipse plugin that serves as the IDE for creating AADL models. Both AGREE and the Safety Annex run as plugins in OSATE. OSATE provides both a language (AADL annex to annotate the models with assume-guarantee behavioral contracts in the case of AGREE and an AADL annex to annotate the model with faults in the case of the safety annex) and a tool (for compositional verification of the contracts reside in AADL models). AGREE translates an AADL model and its contract annotations into Lustre and then queries the JKind model checker to perform the verification. JKind invokes a backend Satisfiability Modulo Theories (SMT) solver (e.g., Yices or Z3) to validate if the guarantees are valid in the compositional setting. The safety annex uses an extension point in AGREE to access the AGREE program and insert the faults into the AGREE contracts. Then that program is translated into Lustre and the JKind model checker is queried to perform the verification/safety analysis.



Figure 23: Overview of Safety Annex/AGREE/OSATE Tool Suite

## Installation

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

### Install OSATE

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

After following the OSATE download instructions found on the OSATE download site (above). The splash screen shown in Figure 19 should appear, and OSATE should begin loading.



Figure 24: OSATE Loading Screen

If OSATE loads successfully, continue to the next step in the installation process. If not, and you are running Windows, the most likely culprit involves mismatches between the 32-bit and 64-bit version of OSATE and the bit-level of the Windows OS. Please check to see whether the version of OSATE matches the bit-level of your version of Windows OS. If running Windows 10, this information can be found in the System Control Panel as shown below in Figure 21. Note that this information is also required for downloading the correct version of the SMT Solver in the next installation step.

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

### Install Safety Annex

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

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

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

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

To test whether the safety annex has been correctly installed, a Safety Annex menu should appear in OSATE as shown in Figure 20.



Figure 25: Safety Analysis Menu Item



Figure 26: Windows 10 System Control Panel

### Install SMT Solver

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

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

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

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

To add directories to your system path in Windows, first navigate to the System Control Panel and choose the "Advanced system settings" button on the left side of the panel. The system properties dialog will appear. Choose the "Advanced" tab in the dialog as shown in Figure 22 then click "Environment variables".



Figure 27: System Properties Dialog Box

The environment variables dialog box is shown in Figure 23.



Figure 28: Environment Variables Dialog Box

In order to make the application available to all user accounts choose the PATH environment variable in the "System variables" section and click "Edit…". This will bring up a text edit box, as seen in Figure 12. If the existing path string in the text edit box does not end with a semicolon (‘;’), add a semicolon first, then append the path to the SMT solver’s "bin" directory, and click "OK" on the dialogs. The bin directory for the Yices tool is underneath the main Yices directory, e.g., C:\Apps\ yices-2.4.2-x86\_64-pc-mingw32-static-gmp\yices-2.4.2\bin. The bin directory for the Z3 tool is underneath the main z3 directory, e.g., C:\Apps\z3-4.4.1-x64-win\z3-4.4.1-x64-win\bin.



Figure 29: System Variable Text Edit Box

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

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

### Set AGREE Analysis Preferences

Use the SMT solver of your choice (Yices or Z3) and set the AGREE Analysis preferences as shown in Figure 25.

Window -> Preferences -> AGREE -> Analysis



Figure 30: AGREE Analysis Preferences

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

## Development Environment Installation

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

### Install OSATE Development Environment

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

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

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

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

Notes and Possible Issues:

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

### Download Safety Annex Source Code

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

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

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

### Github Branches

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

Team -> Switch To -> New Branch

Then type into the textbox the branch as required.

### Run OSATE

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

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