Safety Annex Users Guide

Version

Danielle Stewart, University of Minnesota

Darren Coffer, Rockwell Collins  
Mike Whalen, University of Minnesota

Version History

|  |  |  |  |
| --- | --- | --- | --- |
| Version | Date | Author | Information |
| 0.4 | 9/1/2017 | Danielle Stewart | Initial version of the Safety Annex Users Guide. |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |

[1 Introduction 1](#_Toc472506736)

[2 Brief Overview of AADL and AGREE 2](#_Toc472506737)

[2.1 Using the AGREE AADL Plug-in 4](#_Toc472506738)

[3 AGREE Language 14](#_Toc472506739)

[3.1 Dataflow Language 14](#_Toc472506740)

[3.2 Syntax Overview 15](#_Toc472506741)

[3.3 Lexical Elements 16](#_Toc472506742)

[3.4 Types 17](#_Toc472506743)

[3.5 Subclauses 18](#_Toc472506744)

[3.6 Statements 20](#_Toc472506745)

[3.6.1 Assume Statements 20](#_Toc472506746)

[3.6.2 Guarantee Statements 20](#_Toc472506747)

[3.6.3 Equation Statements 21](#_Toc472506748)

[3.6.4 Property Statements 21](#_Toc472506749)

[3.6.5 Constant Statements 21](#_Toc472506750)

[3.6.6 Node Definitions 21](#_Toc472506751)

[3.6.7 Record Definitions 23](#_Toc472506752)

[3.6.8 Real-time Patterns 23](#_Toc472506753)

[3.6.9 Advanced Topic: Assert statements 24](#_Toc472506754)

[3.6.10 Advanced Topic: Lemma Statements 24](#_Toc472506755)

[3.6.11 Advanced Topic: Linearization Definitions 25](#_Toc472506756)

[3.7 Expressions 26](#_Toc472506757)

[3.7.1 ID Expressions 27](#_Toc472506758)

[3.7.2 NestedDotID (Field) Expressions 27](#_Toc472506759)

[3.7.3 Node Call Expressions 28](#_Toc472506760)

[3.7.4 Linearization Call Expressions 28](#_Toc472506761)

[3.7.5 Stream (Previous Value and Arrow) Expressions 28](#_Toc472506762)

[3.7.6 Event Expressions 29](#_Toc472506763)

[3.7.7 Floor and Real Expressions 30](#_Toc472506764)

[3.7.8 Get Property Expressions 30](#_Toc472506765)

[3.7.9 Unary Minus and Not Expressions 30](#_Toc472506766)

[3.7.10 Record Update Expressions 30](#_Toc472506767)

[3.7.11 Arithmetic Operations 30](#_Toc472506768)

[3.7.12 Relation Expressions 30](#_Toc472506769)

[3.7.13 Boolean Expressions 30](#_Toc472506770)

[4 AGREE/OSATE Tool Suite 31](#_Toc472506771)

[4.1 Tool Suite Overview 31](#_Toc472506772)

[4.2 Installation 32](#_Toc472506773)

[4.2.1 Install OSATE 32](#_Toc472506774)

[4.2.2 Install the SMT Solver 33](#_Toc472506775)

[4.2.3 Install the JKind Model Checker 36](#_Toc472506776)

[4.2.4 Install AGREE 37](#_Toc472506777)

[4.3 Main Features 39](#_Toc472506778)

[4.3.1 Import Existing Projects 39](#_Toc472506779)

[4.3.2 Create New Projects 42](#_Toc472506780)

[4.3.3 Verify Contracts 42](#_Toc472506781)

[4.3.4 Check Realizability 44](#_Toc472506782)

[4.3.5 Export AGREE Contracts 44](#_Toc472506783)

[Appendix A Introduction On K-Induction 46](#_Toc472506784)

[Appendix B AADL Declarations 48](#_Toc472506785)

1. Table of Figures

# Introduction

# Brief Overview of AADL, AGREE, and Safety Annex

## Using the Safety AADL Plug-in

# 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.

## 3.1 Syntax Overview

Before describing the details of the language, we provide some general notes about the syntax. In the syntax notations used below, syntactic categories are indicated by Courier font. Grammar 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 build 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. A cursory overview of AADL declarations and AGREE contracts is provided in Appendix XXX.

AADL describes the interface of a component in a *component type*. A *component type* contains a list of *features* which 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.

## 3.2 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;

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.

## 3.3 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, it is possible to refer to the features and properties of the enclosing component as well as the inputs and outputs of subcomponents (if the subclause is a component implementation). A simplified description of the top-level grammar for Safety annex is shown in Figure XX.

SpecStatement: 'fault' (STRING)? ':' faultDefName=NestedDotID

'{' (FaultSubcomponent)\* '}'

;

FaultSubcomponent:

'inputs' ':' NamedID '<-' Expr

(','NamedID '<-' Expr)\*';'

| 'outputs' ':' NestedDotID '<-' NamedID

(','NestedDotID '<-' NamedID)\*';'

| 'duration' ':' TemporalConstraint Interval ';'

| 'trigger' ':' TriggerCondition ('['probability=REAL\_LIT ']')? ';'

| SafetyEqStatement

;

TemporalConstraint:

'permanent'

| 'transient'

;

TriggerCondition:

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

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

;

SafetyEqStatement:

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

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

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

;

Interval:

=>({*ClosedInterval*} '[' Expr ',' Expr ']')

| =>({*OpenLeftInterval*} '(' Expr ',' Expr']')

| =>({*OpenRightInterval*} '[' Expr ',' Expr ')')

| =>({*OpenInterval*} '(' Expr ',' Expr ')')

;

A Safety subclause consists of a spec statement which consists of a sequence of statements. These different kinds of statements and their uses are described in section 3.4.

Safety subclauses can occur either within an AADL component or component implementation.

## 3.4 Spec Statement

The Safety annex subclause can contain one or more spec statements. The following shows the syntax of a spec statement:

SpecStatement: 'fault' (STRING)? ':' faultDefName=NestedDotID

'{' (FaultSubcomponent)\* '}' ;

Each spec statement corresponds with one fault definition that will wrap a single component. In the case of multiple fault types on a component with multiple outputs, the subclause will contain more than one spec statement; one for each of the fault definitions.

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 the component implementation 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).

An example of a fault node is provided:

**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;**

The input and output statements (section 3.5.1, 3.5.2) will refer directly to the inputs and return values of the fault node. Every fault node definition contains an input parameter called trigger. This parameter is collected from the TriggerStatement (see section 3.5.4). All other input parameters are linked in the Inputs statement (section 3.5.1) and the return values are linked with AADL component in the Outputs statement (section 3.5.2).

The fault spec statement will contain zero or more Fault Subcomponent statements. In the case of zero, no faults wrap the AADL component and hence no fault analysis is performed.

## 3.5 Fault Subcomponent Statements

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)? ';'

| 'trigger' ':' TriggerCondition ('['probability=REAL\_LIT ']')? ';'

| SafetyEqStatement

;

## 3.5.1 Input Statements

Input statements are where the parameters of the fault node definition are linked to expressions which assign the node parameters a value. Each fault node has a trigger parameter. This is the only input parameter that is not accounted for in the input statement.

As an example, we look at the fail\_to\_int fault node definition and provide an example of the input statement associated with this node.

**node** fail\_to\_int**(**val\_in**: int,** alt\_val**:** **int,** trigger**:** **bool)**

**returns(**val\_out**:** **int);**

**let**

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

**tel;**

The inputs that must be explicitly stated are: val\_in and alt\_val. The left side of the input statement must use these identifiers. The right side of the input statement consists of AGREE or AADL expressions (see AGREE Users Guide, section 3.7). Examples of this include boolean or arithmetic expressions as well as AADL Data Implementation variables. The following is an example of an input statement using the fail\_to\_int node and the medical device model from Figure XXX:

**inputs:** val\_in **←** Alarm.Log\_Message\_ID**,**

alt\_val **←** 2\*Alarm.Log\_Message\_ID**;**

This input statement will ensure that the value associated with Alarm.Log\_Message\_ID is passed in as the val\_in parameter and likewise the value associated with 2\*Alarm.Log\_Message\_ID is the failure value if the fault is triggered. **Note: The trigger value is not specified within the input statement.** See section 3.5.4 on Trigger Statements.

## 3.5.2 Output Statements

Output statements will specify which component output will be affected by the fault node output. Since nodes may have more than one output, each must be linked to a component. Using the same example in 3.5.1 ( fail\_to\_int node and the medical device model from Figure XXX), we describe the associated output statement:

**outputs:** val\_out **←** Alarm.Log\_Message\_ID**;**

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.

## 3.5.3 Duration Statements

A duration statement specifies whether the fault will be transient or permanent. A transient fault has associated with it a time interval of discrete time steps. A permanent fault will remain indefinitely and has no such interval in the statement.

An example of a transient fault lasting two time steps is as follows:

**duration: transient** [0,1]**;**

An example of a transient fault lasting 7 time steps is as follows:

**duration: transient** [0,6]**;**

An example of a permanent fault is as follows:

**duration: permanent;**

## 3.5.4 Trigger Statements

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:

'trigger' ':' TriggerCondition ('['probability=REAL\_LIT ']')? ';'

TriggerCondition:

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

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

;

As an example, consider a fault that has two enabler triggers, i.e. if either of the triggers occur, then the fault may occur. We will call these triggers Expr1 and Expr2 in the example below. This particular trigger also has a 0.1% chance of occurring.

**trigger: enabler {** Expr1**,** Expr2 **}** [0.001]**;**

An equivalent way of writing this statement is using an explicit disjunctive form instead of the implicit disjunctive form of a list:

**trigger: enabler {** Expr1 **or** Expr2 **}** [0.001]**;**

**Note: Any of the boolean expressions used in AGREE or AADL can be used for expressions (i.e. and, or, if..then..else, true, false).**

## 3.5.5 Safety Equation Statements

The AGREE language makes use of Equation Statements (AGREE Users Guide, section 3.6.3) . The Safety annex has extended this definition to be able to account for various kinds of deterministic and nondeterministic behavior. There are three kinds of Safety Equation Statements.

## 3.5.5.1 Equation Statement

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** count**:** **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** var**:** **real;**

To use a nondeterministic value within a fault statement, the equation statement would be defined as above and then used in the input statement to link with the fault node.

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.

## 3.5.5.2 Interval Equation Statements

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)**;**

## 3.5.5.3 Set Equation Statements

Set equation statements can be used to specify a set of discrete intervals of nondeterminism. The set is interpreted as follows: FINISH ME.

# Safety Annex/AGREE/OSATE Tool Suite

## 4.1 Tool Suite Overview

## 4.2 Installation

## 4.2.1 Install OSATE

## 4.2.2 Install the SMT Solver

## 4.2.3 Install the JKind Model Checker

## 4.2.4 Install AGREE

## 4.2.5 Install Safety Annex