AGREE Users Guide

Version 0.4

Andrew Gacek, Rockwell Collins Inc.
John Backes, Rockwell Collins Inc.
Mike Whalen, University of Minnesota
Darren Cofer, Rockwell Collins, Inc.

Version History

Version	Date	Author	Information
0.4	11/20/2013	Mike Whalen	Initial version of the AGREE Users Guide.

Table of Contents

1 Introduction

The Assume Guarantee REasoning Environment (AGREE) is a *compositional, assume-guarantee-style* model checker for AADL models. It is *compositional* in that it attempts to prove properties about one layer of the architecture using properties allocated to subcomponents. The composition is performed in terms of *assumptions* and *guarantees* that are provided for each component. *Assumptions* describe the expectations the component has on the environment, while *guarantees* describe bounds on the behavior of the component. AGREE uses *k-induction* as the underlying algorithm for the model checking.

The main idea is that complex systems are likely to be designed as a hierarchical federation of systems. As we descend the hierarchy, design information at some level turns into requirements for subsystems at the next lower level of abstraction. These hierarchical levels can be straightforwardly expressed in AADL. What we would like to support, therefore, is:

- an approach to requirements validation, architectural design, and architectural verification that uses the requirements to drive the architectural decomposition and the architecture to iteratively validate the requirements, and
- an approach to verify and validate components prior to building code-level implementations.

AGREE is a first step towards realizing this vision. Components and their connections are specified using AADL and annotated with *assumptions* that components make about the environment and *guarantees* that the components will make about their outputs if the assumptions are met. Each layer of the system hierarchy is verified individually; AGREE attempts to prove the system-level guarantees in terms of the guarantees of its components. This guide explains the syntax of AGREE and how to use the AGREE plugin for OSATE/Eclipse.

This document is organized as follows: Chapter 2 describes a small example. Chapter 3 describes the syntax of the AGREE language. Chapter 4 describes the AGREE tool suite. Chapter 5 (TBD) describes the architecture of AGREE.

2 Brief Overview of AADL and AGREE

AGREE is meant to be used in the context of an AADL model. AGREE models the components and their connections as they are described in AADL. This section provides a very brief introduction to AADL and AGREE 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.

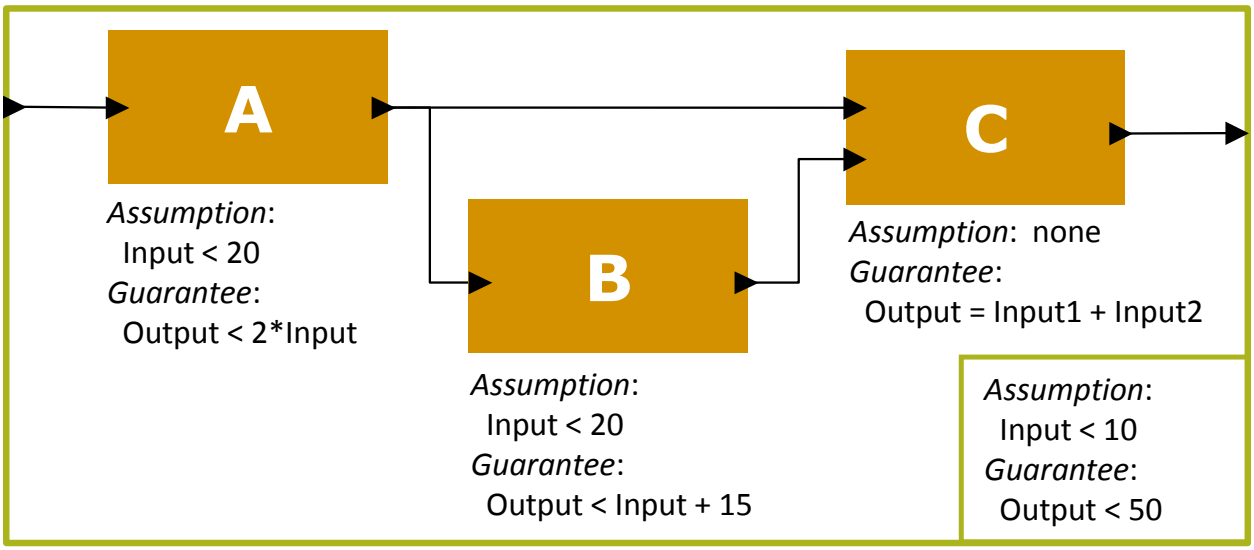


Figure 1: Toy Compositional Proof Example

In the model in Figure 1, we want to show that the system level property (Output < 50) holds, given the guarantees provided by the components and the system assumption (Input < 10). This toy example has one interesting feature: the property is *true* if all of the signals have type integer and it is *false* if they have floating point types (can you see why?).

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;

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;
```

```
   **};
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;
   **};
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 {**
      guarantee "C output range" : Output = Input1 + Input2;
   **};
end C ;

system top_level
   features
      Input: in data port Base_Types::Integer;
      Output: out data port Base_Types::Integer;
   annex agree {**
      assume "System input range " : Input < 10;
      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;};
end top_level.Impl;

end Integer_Toy;
```

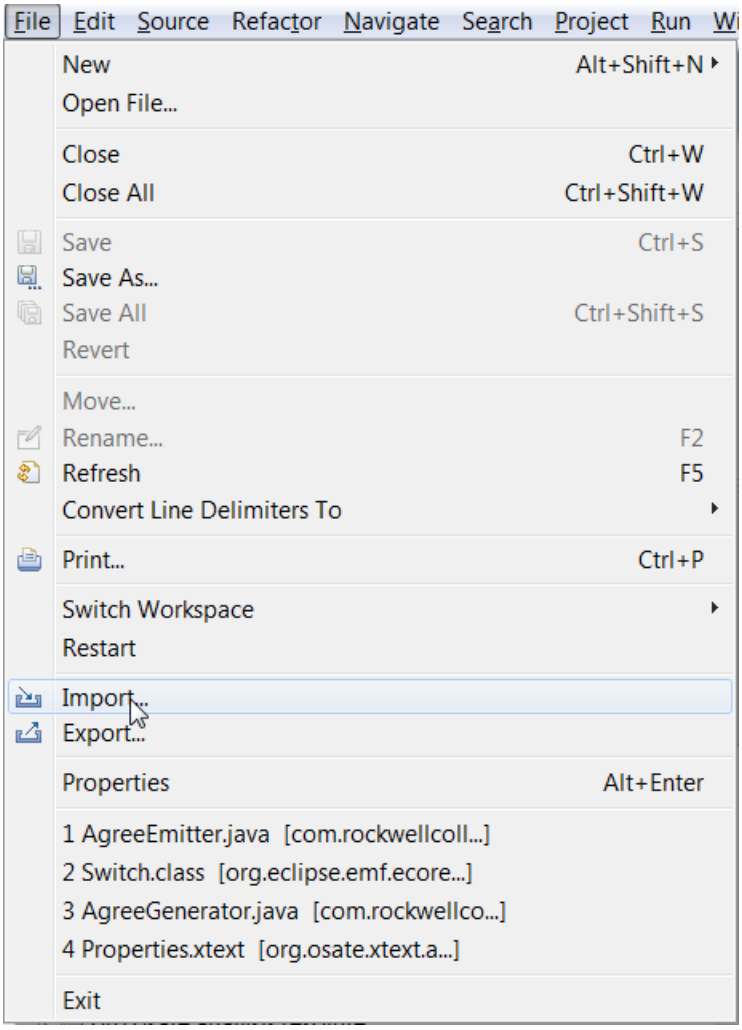
Figure 2: AADL Code for Integer Model

In Figure 2, **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.* We will explain more about timing and connection delays in Chapter 3: AGREE Language. 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).

2.1 Using the AGREE AADL Plug-in

From the class materials, I have provided a .zip file containing the Toy_Verification project that contains the example from Figure 2. After unzipping the model, it can be imported by choosing File > Import:



Then choosing “Existing Project into Workspace”

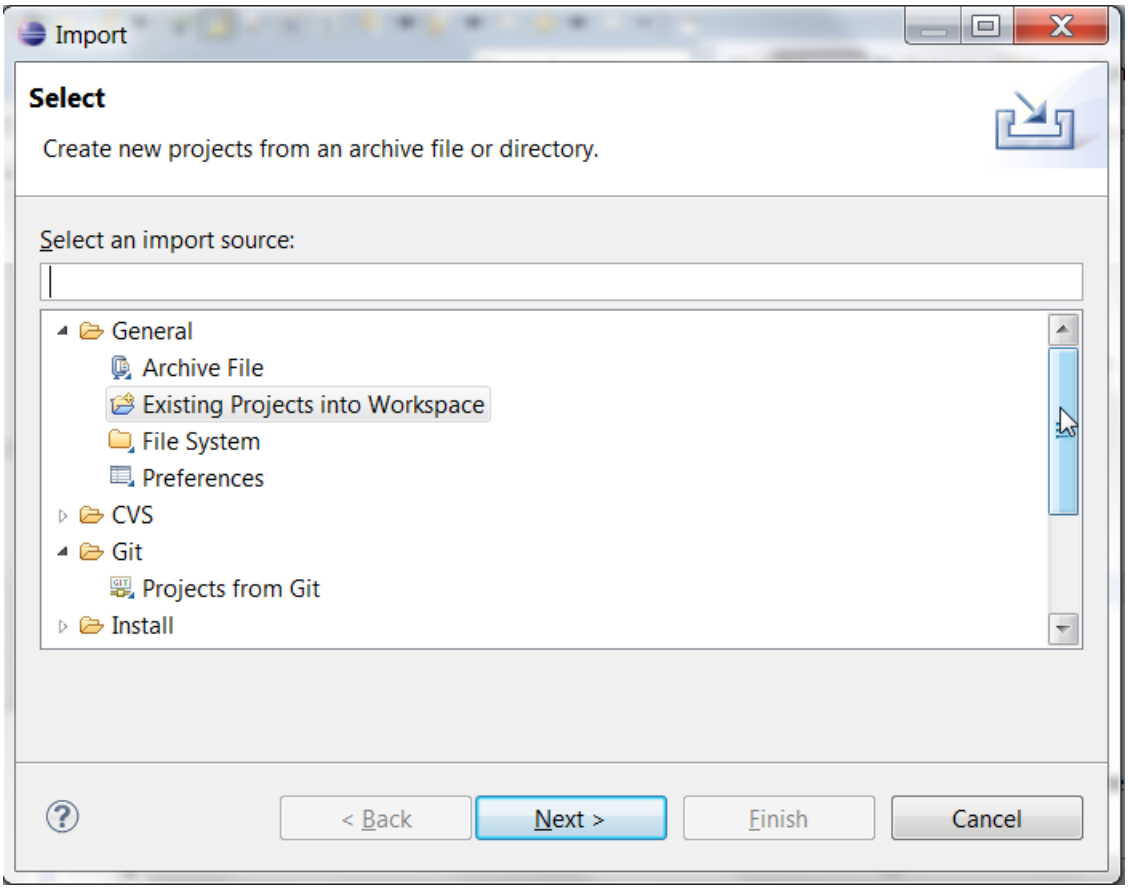


Figure 3: Importing Toy_Verification Project

and navigating to the unzipped directory after pressing the Next button. Figure 4 shows what the model looks like when loaded in the AGREE/OSATE tool.

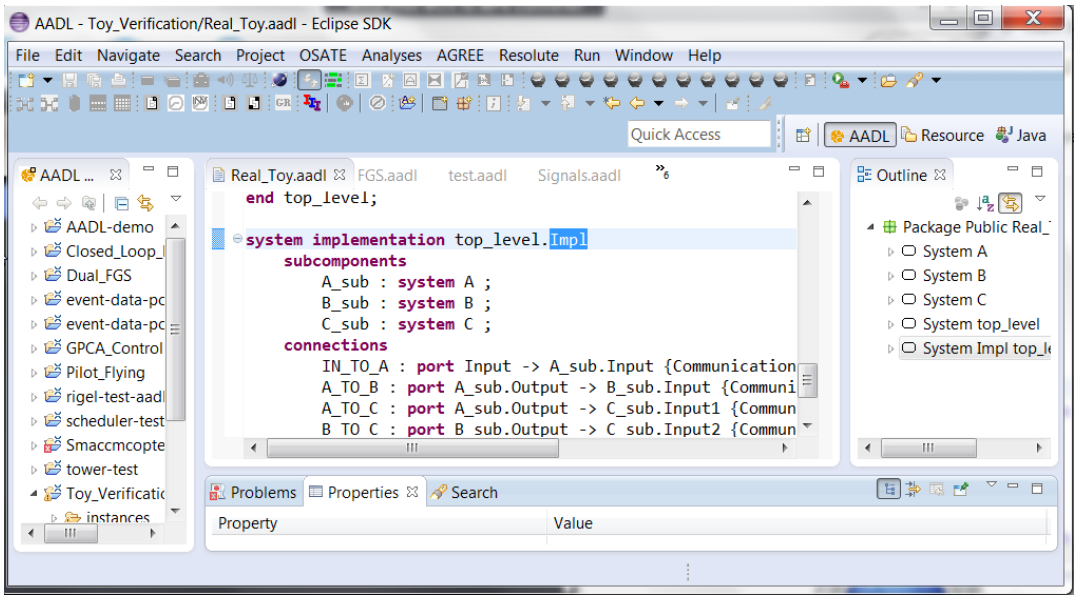
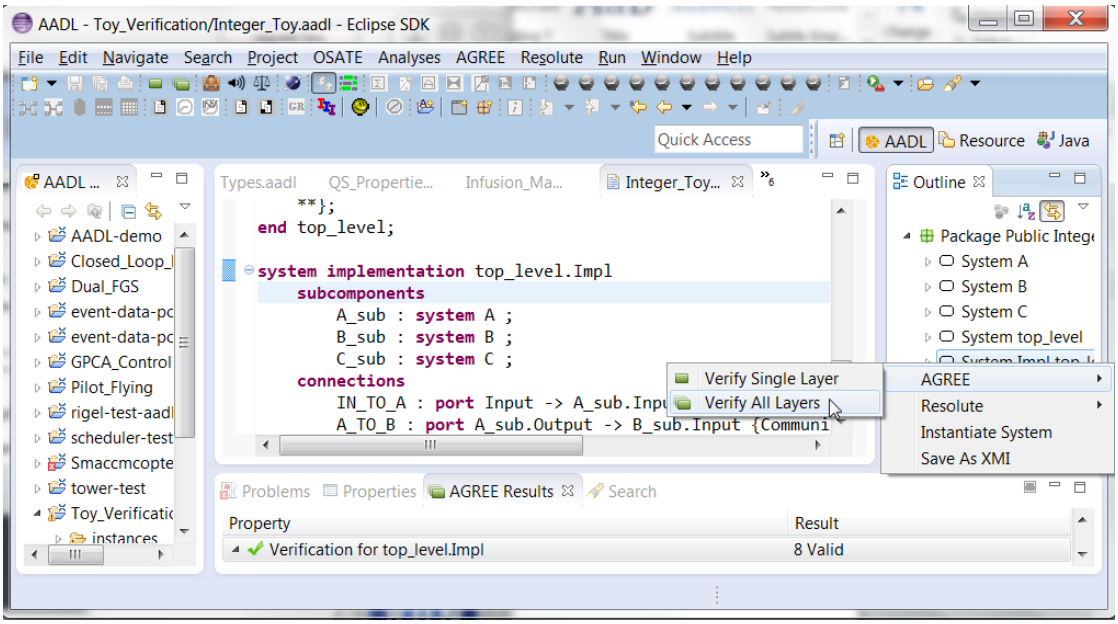


Figure 4: AGREE/OSATE Environment

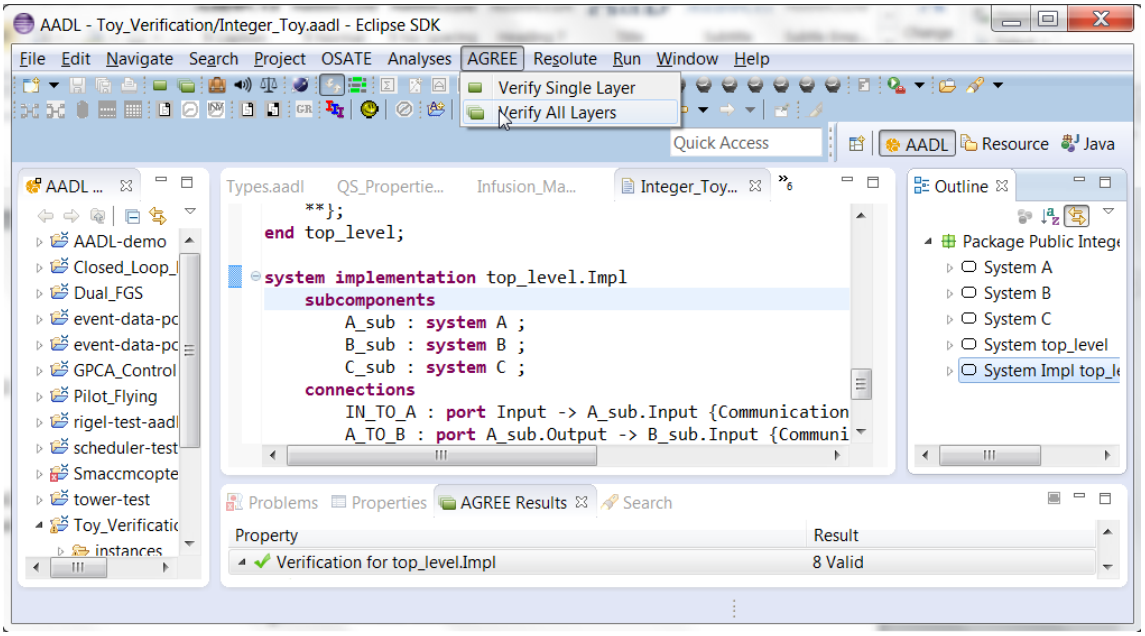
Note that in the workspace in Figure 4, there are several projects, so your workspace will probably look slightly different. The project that we are working with is called Toy_Verification.

Open the Integer_Toy.aadl model by double-clicking on the file in the AADL Navigator pane. To invoke AGREE, we select the Top_Level.Impl system implementation in the outline pane on the right. We can then either

1. right-click on the Top_Level.Impl element on the outline pane and choose "AGREE > Verify All Layers":



2. Or, Choose the “Verify All Layers" item from the AGREE menu:



As AGREE runs, you should see checks for “Contract Guarantees”, “Contract Assumptions” and “Contract Consistency” as shown in Figure 5.

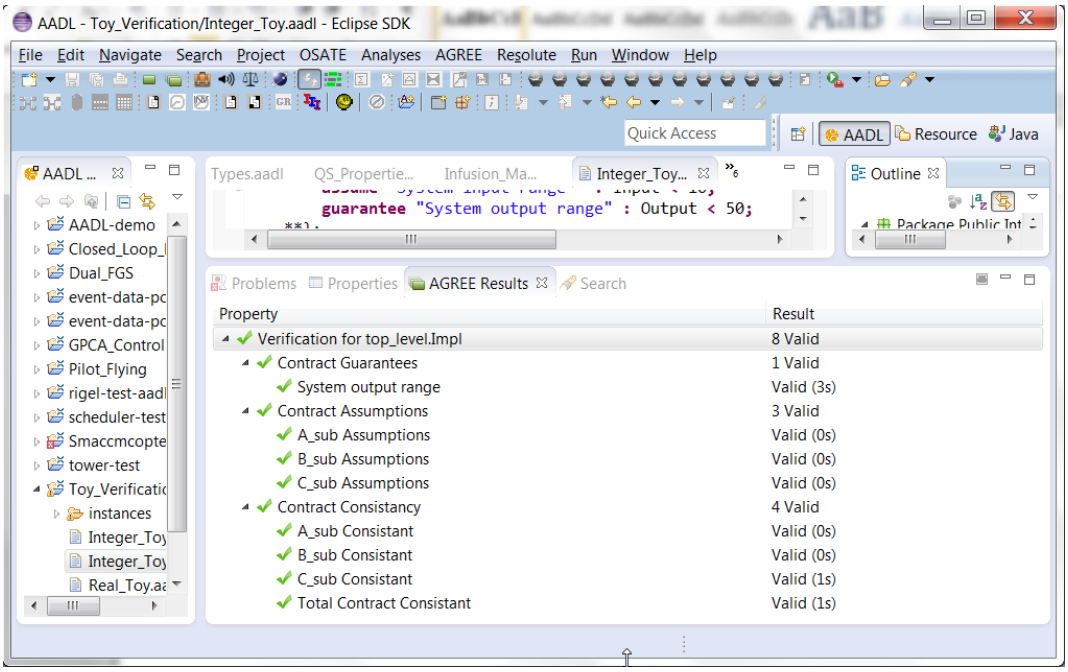


Figure 5: AGREE Results

Now, let's analyze the same model but with the ports instantiated to floating point numbers. Open the Real_Toy.aadl model by double clicking on the file in the AADL Navigator panel. Again select the top_level.Impl System Implementation in the outline panel on the right of OSATE, and either right-click and choose the "AGREE" menu or choose the "AGREE" menu in Eclipse.

Now the top-level property fails, as shown in Figure 6.

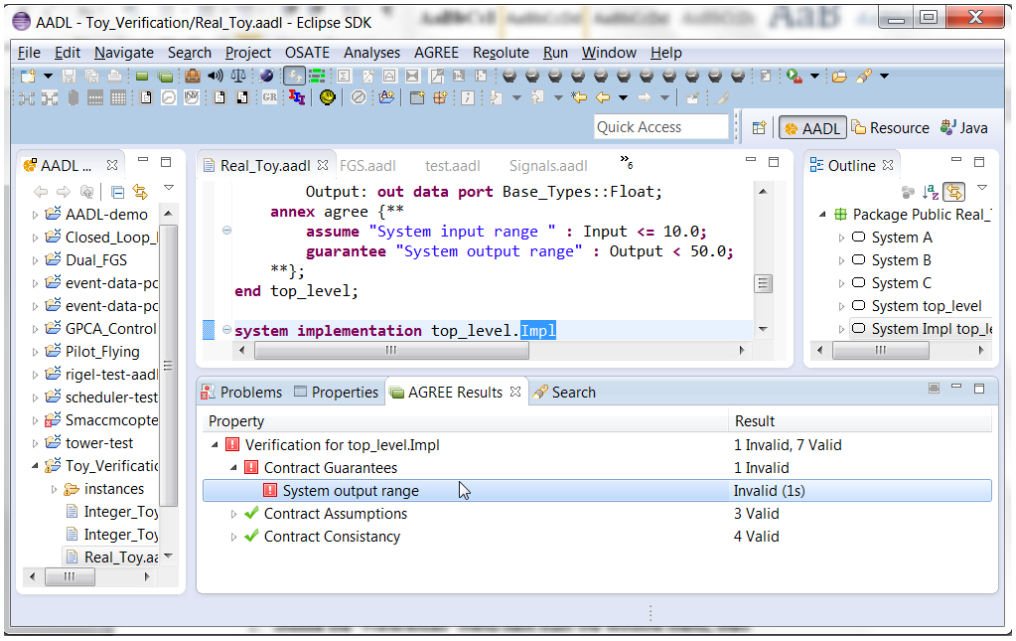


Figure 6: Failed Property

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 7 shows the counterexample that is generated by this failure in the console window.

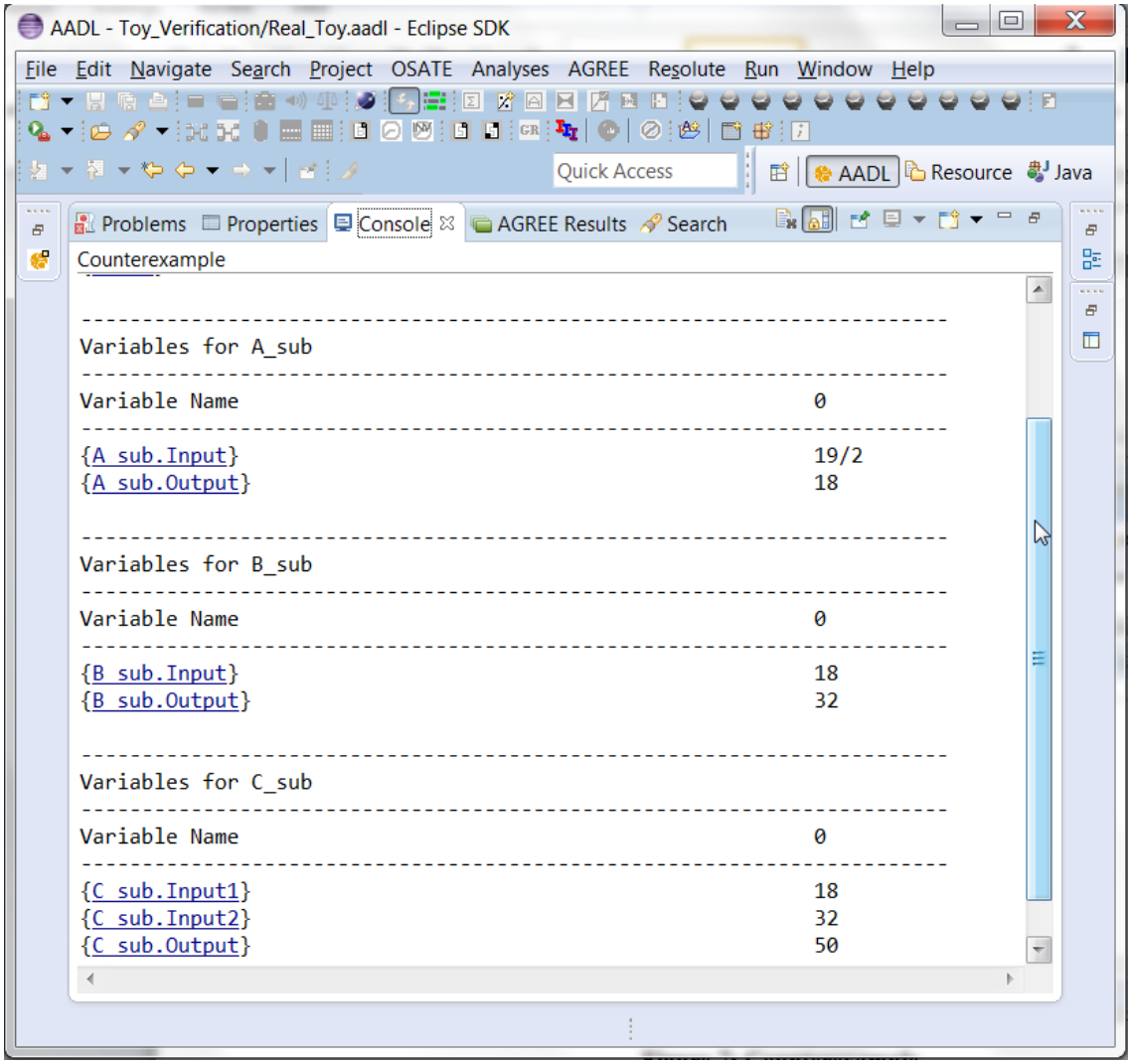


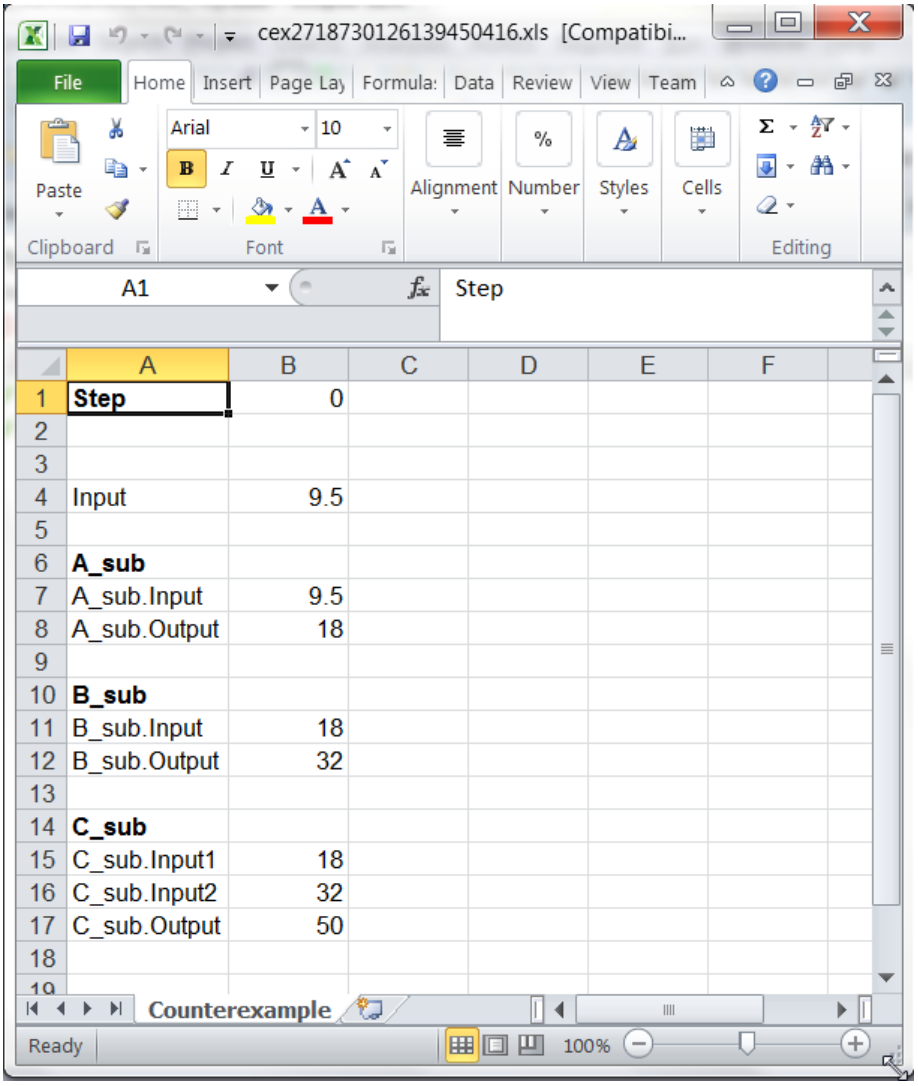
Figure 7: Counterexample

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". **NB: 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,**

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 8.



	A	B
1	**Step**	0
2		
3		
4	Input	9.5
5		
6	**A_sub**	
7	A_sub.Input	9.5
8	A_sub.Output	18
9		
10	**B_sub**	
11	B_sub.Input	18
12	B_sub.Output	32
13		
14	**C_sub**	
15	C_sub.Input1	18
16	C_sub.Input2	32
17	C_sub.Output	50
18		

Figure 8: Excel Counterexample File

Note that this counterexample is only one step long. If it were multiple steps, these would be displayed in consecutive columns from left to right.

When executed with real-valued inputs and outputs, it is possible to find a counterexample to the system-level property. In this counterexample, the system input is 9.5, so it is less than 10, but the system output is equal to 50, violating the system guarantee. Can you find the reason for the counterexample?

One possible reason, in this case, is that since we are not using integer inequalities on the various components, the assumptions and guarantees are too “loose”. There are several ways that this can be fixed (try some out yourself before reading ahead).

One possible fix is to change the system assumption to ensure that the input value is small enough (Input < 8.0 is sufficient). What is the largest range for the input that can ensure the property? Can you determine it exactly?

3 Chapter 3: AGREE Language

In this chapter we present the syntax and semantics of the input language of AGREE. We first present an overview of the computational model of the language, then present the syntax of the language.

The AGREE language is derived from the *synchronous dataflow language* Lustre. Let us expand on this definition somewhat. A *dataflow* language consists of a set of *equations* that assign *variables* in which a variable can be computed as soon as its *data dependencies* have been computed. As an example, consider a system that computes the values of two variables, X and Y, based on four inputs: a, b, c, and d:

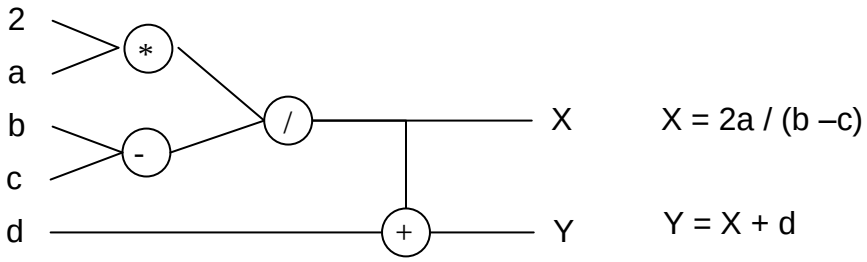


Figure 9: A dataflow model and its associated set of equations

This diagram is to be read left-to-right, with the inputs "flowing" through the system of operators to create the outputs at the right side. The diagram can be represented more concisely as a set of equations, as shown at right. We name the inputs to the dataflow model *input variables* and all variables that are computed by the model *state variables.*

As the basis of a high-level programming language, the dataflow model has several merits:

- It is a completely functional model without side effects. This feature makes the model well-suited to formal verification and program transformation. It also facilitates reuse, as a module will behave the same way in any context into which it is embedded.

- It is a naturally parallel model, in which the only constraints on parallelism are enforced by the data-dependencies between variables. This allows for parallel implementations to be realized, either in software, or directly in hardware.

Dataflow models can be either *synchronous* or *asynchronous*. In an asynchronous dataflow model, the outputs of the system are continually recomputed depending on the inputs to the system. In the synchronous model, however, real-time is broken into a sequence of instants in which the model is recomputed. The synchronous model is better suited to translation into a programming language, as it more naturally matches the behavior of a computer program. Therefore, all of the dataflow-style languages adopt some form of this approach.

The variables in a dataflow model are used to label a particular computation graph; they are not used as constraints. Therefore, it is incorrect to view the equations as a set of constraints on the model: a set of equations such as {X = 2a/Y, Y = X + d} does not correspond to an operator network because X and Y mutually refer to one another. Put another way, there is no way to arrange the variables from left to right such that each can be computed. This is shown in Figure 10, where the bold red-lines indicate the cyclic dependencies. Such a system may have no solution or infinitely many solutions, so cannot be directly used as a deterministic program. If viewed as a graph, these sets of equations have *data dependency cycles*, and are considered incorrect.

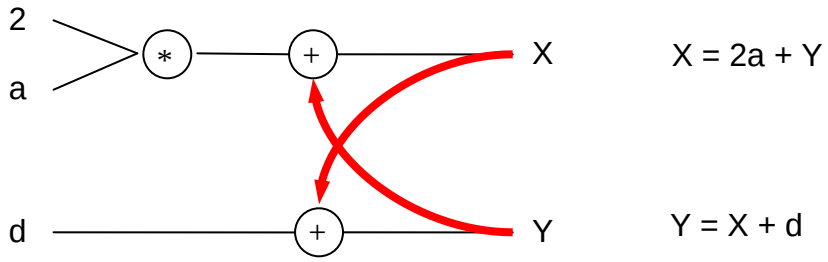


Figure 10: A Dataflow Model with Cyclic Dependencies

However, in order for the language to be useful, we must be able to have mutual reference between variables. To allow benign cyclic dependencies, a *delay operator* (prev) is added. The operator returns the value of an expression, delayed one instant. For example: {X = 2a + Y; Y = (prev(X, 1)) + d} defines a system where X is equal to 2a plus the current value of Y, while Y is equal to the *previous* value of X (with value in the initial instant set to 1) plus the current value of d. Systems of equations of this form always have a single solution. The delay operator is also the mechanism for recording state about the model. For example, we can construct a counter over the natural numbers by simply defining the equation: x = prev(x+1, 1).

Finally, some notion of selection is added to assignment expressions. In Lustre, this is simply an if/then/else statement. From these elements, at its core, a dataflow program can be viewed as simply a set of input variables and assignment equations of the form $\{X_0 = E_0, X_1 = E_1, ..., X_n = E_n\}$ that must be acyclic in terms of data dependencies.

3.1 Syntax Overview

Before describing the details of the language, we provide a few general notes about the syntax. In the syntax notations used below, syntactic categories are indicated by `Consolas monospace` font. Grammar productions enclosed in parenthesis (‘( )’) indicate a set of choices in which a vertical bar (‘|’) is used to separate alternatives in the syntax rules or ‘..’ is used to describe a range (e.g. ('A'..'Z')). Sometimes one of is used at the beginning of a rule as a shorthand for choosing among several alternatives. The * character indicates repetition (zero or more occurrences) and + indicates required repetition (1 or more occurrences). A ? character indicates that the preceding token is optional. Any characters in single quotes describe concrete syntax: (e.g.: '+', '-', '=>', '"'). Note that the last example is

the concrete syntax for a single quote. Examples of grammar fragments are written in the `Courier monospace` font.

AGREE is built on top of the AADL 2.0 architecture description language. The AGREE formulas are found in an AADL *annex*, which extends the grammar of AADL. Generally, the annex follows the conventions of AADL in terms of lexical elements and types with some small deviations (which are noted). AGREE operates over a relatively small fragment of the AADL syntax. Thus familiarity with the entire AADL language is not required. We will build up the language starting from the smallest fragments.

3.2 Lexical Elements

Comments always start with two adjacent hyphens and span to the end of a 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')
```

Some example identifiers are: `Count`, `X`, `Get_Symbol`, `Ethelyn`, `Snobol_4`, `X1`, `Page_Count Store_Next_Item`. **NB: Identifiers are case insensitive**! Thus `Hello`, `HeLlO`, and `HELLO` all refer to the same entity in AADL.

Literal numeric values are defined as follows:

```
numeric_literal ::= integer_literal | real_literal

integer_literal ::= decimal_integer_literal | based_integer_literal

real_literal ::= decimal_real_literal

decimal_integer_literal ::= numeral ( positive_exponent )?

decimal_real_literal ::= numeral . numeral ( exponent )?

numeral ::= digit ( (underline)? Digit )*

exponent ::= E ('+')? numeral | E '-' numeral

positive_exponent ::= E ('+')? numeral
```

Some examples are: `12`, `0`, `1E6`, `123_456`, `12.0`, `0.0`, `0.456`.

String elements are defined with the following syntax:

```
string_literal ::= "(string_element)*"

string_element ::= "" | non_quotation_mark_graphic_character
```

3.3 Types

The types that are supported again come from the AADL Data Modeling Annex, which is presented in full in Appendix B. The following is an excerpt from the AADL Data Modeling Annex describing the AADL types. The Data Model annex document provides guidelines for data modeling in the following way. User-defined AADL data component types represent application data types. These data component types are then annotated with properties to indicate relevant details of the data type in a data modeling language or source language. The property `Data_Representation` specifies the realization of the data type in terms of basic data types built into the language, such as real, integer, string, and Boolean, and in terms of composite data types, such as *array* (ordered sets accessible by index), *struct* (collection of named elements), and *union* (collection of named alternatives).

In the case of array or string, additional properties characterize the size of the array in terms of multiple dimensions (`Dimension`) and the data type of the array elements (`Base_Type`). Note that the type of the array elements is characterized in terms of an AADL defined data component type. This allows arrays of user-defined types as well as base types to be modeled. The Data Modeling Annex includes a package (`Base_Types`) of predeclared AADL data component types (and implementations) for a set of data types that represent built-in data types. Those data component types are themselves characterized by the `Data_Representation` property as representing a built-in data types such as integer, fixed, float, string, character, enum, Boolean.

Because in AADL the types are defined as instances of AADL data objects, they are not presented as a grammar. Instead, the Base_Types package containing the AADL data objects is presented, followed by examples. Informally, the basic types are as follows.

The `Base_Types::Boolean` type comprises symbolic values false and true.

The `Base_Types::Integer` type describes "unbound" integers. To describe integers of various bit sizes, `Base_Types::Integer_8`, `Base_Types::Integer_16`, `Base_Types::Integer_32`, and `Base_Types::Integer_64` represent 8, 16, 32, and 64 bit signed integers and `Base_Types::Unsigned_8`, `Base_Types::Unsigned_16`, `Base_Types::Unsigned_32`, and `Base_Types::Unsigned_64` describe 8, 16, 32, and 64 bit unsigned types.

The Base_Types::Float_32 and Base_Types::Float_64 types define single and double precision floating point numbers.

3.3.1 Base_Types Package

```
package Base_Types
public
  data Boolean
```

```
properties
  Data_Model::Data_Representation => Boolean;
end Boolean;

data Integer
properties
  Data_Model::Data_Representation => Integer;
end Integer;

-- Signed integer of various byte sizes

data Integer_8 extends Integer
properties
  Data_Model::Number_Representation => Signed;
  Source_Data_Size => 1 Bytes;
end Integer_8;

data Integer_16 extends Integer
properties
  Data_Model::Number_Representation => Signed;
  Source_Data_Size => 2 Bytes;
end Integer_16;

data Integer_32 extends Integer
properties
  Data_Model::Number_Representation => Signed;
  Source_Data_Size => 4 Bytes;
end Integer_32;

data Integer_64 extends Integer
properties
  Data_Model::Number_Representation => Signed;
  Source_Data_Size => 8 Bytes;
end Integer_64;

-- Unsigned integer of various byte sizes
```

```
data Unsigned_8 extends Integer
properties
  Data_Model::Number_Representation => Unsigned;
  Source_Data_Size => 1 Bytes;
end Unsigned_8;

data Unsigned_16 extends Integer
properties
  Data_Model::Number_Representation => Unsigned;
  Source_Data_Size => 2 Bytes;
end Unsigned_16;

data Unsigned_32 extends Integer
properties
  Data_Model::Number_Representation => Unsigned;
  Source_Data_Size => 4 Bytes;
end Unsigned_32;

data Unsigned_64 extends Integer
properties
  Data_Model::Number_Representation => Unsigned;
  Source_Data_Size => 8 Bytes;
end Unsigned_64;

data Natural extends Integer
properties
  Data_Model::Integer_Range => 0 .. Max_Target_Integer;
end Natural;

data Float
properties
  Data_Model::Data_Representation => Float;
end Float;

data Float_32 extends Float
properties
  Data_Model::IEEE754_Precision => Simple;
  Source_Data_Size => 4 Bytes;
```

```
  end Float_32;

  data Float_64 extends Float
  properties
    Data_Model::IEEE754_Precision => Double;
    Source_Data_Size => 8 Bytes;
  end Float_64;


  data Character
  properties
    Data_Model::Data_Representation => Character;
  end Character;


  data String
  properties
    Data_Model::Data_Representation => String;
  end String;
end Base_Types;
```

3.3.2 Examples

```
package Base_Types::Example_Types
public
  with Data_Model, Base_Types;

  data One_Dimension_Array
  properties
    Data_Model::Data_Representation => Array;
    Data_Model::Base_Type => (classifier (Base_Types::Unsigned_32));
    Data_Model::Dimension => (42);
  end One_Dimension_Array;

  data Two_Dimensions_Array
  properties
    Data_Model::Data_Representation => Array;
    Data_Model::Base_Type => (classifier (Base_Types::Integer_32));
    Data_Model::Dimension => (74, 75);
  end Two_Dimensions_Array;
```

```
data A_Struct2
  properties
    Data_Model::Data_Representation => Struct;
  end A_Struct2;

  data implementation A_Struct2.impl
  subcomponents
    f1 : data Base_Types::Float;
    c2 : data Base_Types::Character;
  end A_Struct2.impl;

data A_Union2
  properties
    Data_Model::Data_Representation => Union;
  end A_Union2;

  data implementation A_Union2.impl
  subcomponents
    f1 : data Base_Types::Float;
    c2 : data Base_Types::Character;
  end A_Union2.impl;

data An_Enum
  properties
    Data_Model::Data_Representation => Enum;
    Data_Model::Enumerators => ("foo", "bar");
    Data_Model::Representation => ("00", "11");
  end An_Enum;
end Base_Types::Example_Types;
```

3.3.3 Important Note About Types in AGREE

Currently all bit-sized integer and unsigned types are approximated by unbound integers in AGREE. Similarly, all floating point numbers are approximated by rational numbers. This is due to the representation used by the underlying *jkind* solver. **This means that AGREE results are not guaranteed to be sound with respect to system implementations that use bit-level representations.** We expect that future versions of *jkind* will support bit-level integers, as these are widely supported by solvers. On the other hand, floating point solvers are currently immature, so it is likely that reals will be used for the

forseeable future. If exact floating point behavior (including rounding and truncation) are important to your verification problem, AGREE may provide incorrect answers.

3.4 Expressions

The set of expressions for AGREE is described in the grammar below:

```
RelateOp::=
       '<' | '<=' | '>' | '>=' | '=' | '<>' | '!=';

QID::= ID '::' ID ;

NestedDotID ::= ID ('.' NestedDotId)? ;

FunctionCall ::= NestedDotId '(' Expr (',' Expr)* ')' ;

Expr ::=
         Expr '->' Expr
       | Expr '=>' Expr
       | Expr '<=>' Expr
       | Expr 'or' Expr
       | Expr 'and' Expr
       | Expr RelateOp Expr
       | Expr ('+' | '-') Expr
       | Expr ('*' | '/' | 'div') Expr
       | ('-' | 'not') Expr
       | 'if' Expr 'then' Expr 'else' Expr
       | 'prev' '(' Expr ',' Expr ')'
       | 'next' '(' Expr ')'
       | 'Get_Property' '(' Expr ',' Expr ')'
       | ID
       | QID
       | numeric_literal
       | Boolean_literal
       | NestedDotId
       | FunctionCall
       | 'pre' '(' Expr ')'
       | 'this' ('.' NestedDotId)?
       | '(' Expr ')'
;
```

The order of precedence (from lowest to highest) is as follows:

```
->
=>
<=>
or
and
< | <= | > | >= | = | <> |!=
+ | -
* | / | div
unary minus | not
```

if then else
prev | next | Get_Property
ID | QID | NestedDotID | numeric_literal | Boolean_literal | FnCallExpr | pre | this | ()

Thus, `x + if y then a else b * prev(z.f - 1, 0)` would be parsed as follows:

```
x + (if y then a else (b * (prev((z.f) - 1, 0))))
```

The meaning of the arithmetic, relational, and Boolean operators is straightforward. If/then/else is an *expression,* not a *statement*; it behaves like the ?: operator in Java. So, you can write:

```
x = if (b) then y else z ;
```

3.4.1 Id and Field Expressions

Identifier expressions support reference to different AADL objects as well as AGREE variables and constants. Constants or variables must be defined locally (in the AGREE annex block or the enclosing definition), and they can be referred to by a single identifier ID. It is possible to refer to the input and output ports of subcomponents using the `x.y.z` notation. The same notation is used for inputs and outputs that are of record type: if x is a record type containing field y, then the notation `x.y` is used.

3.4.2 Stream Expressions

The `prev` expression defines an initialized stream. So, if we write:

```
x = prev(y + 1, 0);
```

In the initial instant, x is equal to 0. In all subsequent instants, x is equal to the previous value of y +1. If we examine the evolution of x and y over a time window of ten steps, it is relatively straightforward to see.

Time Instant	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**
y	4	5	8	7	3	12	6	9	1	3
y+1	5	6	9	8	4	13	7	10	2	4
x	0	5	6	9	8	4	13	7	10	2

The arrow (->) operator is the stream initialization operator. Given an expression x -> y, in the initial instant in time, the value is equal to x. In all subsequent instants, it is equal to y. So, suppose we have:

```
x = false -> a
```

Then, in the first instant in time, x will be assigned ‘false’ and in every other instant in time, it will be assigned ‘a’.

NB: A common mistake is to mis-type '->' for '=>' (and vice-versa). This will often cause your model to return incorrect results. Please check for this error. The '=>' operator is the implication operator: if you write:

```
a => b,
```

instead, then `a` and `b` are expected to be Boolean expressions and the meaning of the operator is equivalent to `(not a) or b`. So, writing:

```
x = false => a
```

Will assign x to true in all time instants:

```
x = false => a ⇔
x = (not false) or a ⇔
x = true or a ⇔
x = true
```

The `pre` expression is an *uninitialized* pre expression. Its value is *undefined* in the initial instant. This expression is expected to be used in combination with the arrow expression; this can yield expressions that are, on occasional, more terse than using the `prev` expression. However, the following equivalence always holds for arbitrary expressions x and y:

```
prev(x, y)   ⇔   y -> pre(x)
```

For novice users, we recommend using the initialized `prev` expression as it is less error prone than the `-> pre` combination.

3.4.3 Function Call Expression

The AGREE language supports both functions and *nodes*, which are like functions, but can have state (nodes will be explained in the next section). Unlike other expression types, it is possible for function call expressions to return multiple values, so one can write:

```
eq x: int, y: int = foo(a, b);
```

For a function 'foo' that takes two arguments and returns two values.

3.5 Declarations

There are two kinds of declarations that are of interest for AGREE. First, there are the AADL components that define the architecture that is reasoned about in AGREE. Second, there are local declarations within AGREE annex blocks. In this section, we will only provide a cursory overview of the AADL declarations; for a complete overview, we recommend the standard reference *SAE Aerospace Standard AS5506B: Architecture Analysis and Design Language* and the Addison Wesley book: *System Modeling and Analysis with AADL*.

3.5.1 AADL Declarations

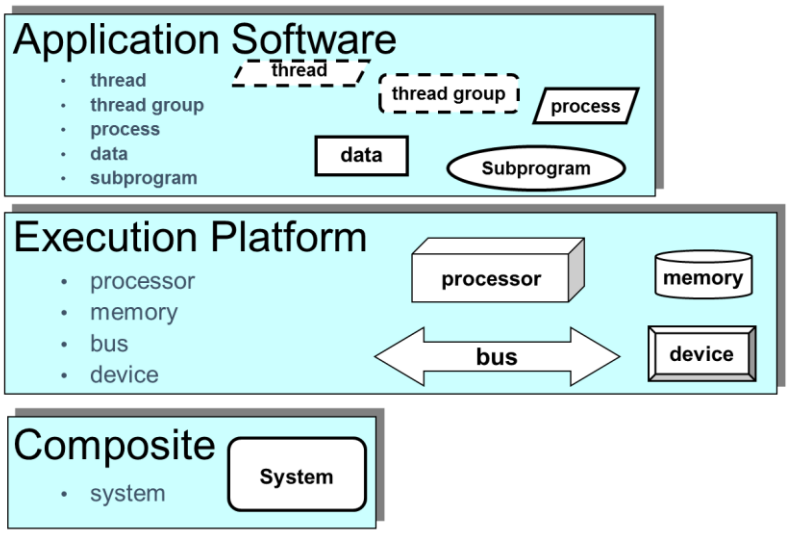


Figure courtesy of Peter Feiler: SAE AADL V2: An Overview

Figure 11: Overview of AADL Components

AADL can be used to describe both software and the physical platform on which it executes, as shown in Figure 11. In the current version of AGREE, only the application software is directly annotated for analysis; information about the physical platform is used to structure the analysis[1], but currently is not annotated. Therefore, it is possible to create AGREE annexes in *thread, thread group, process,* and *system* components.

For each component type, AADL distinguishes between *types, implementations,* and *instances.* In AGREE, we are primarily concerned with *types* and *implementations*, which are shown in Figure 12. The component *type* defines the publicly visible interface to the component: the inputs and outputs to the components (defined by *ports*) as well as input *parameters,* shared memory *access*, and publicly callable *subprograms*. For Java programmers, this is roughly analogous to an *interface*.

[1] In the current version of AGREE, the platform is assumed to be synchronous, so this isn't really true; platforms all behave equivalently. In future releases, we will account for the system architecture in terms of timing and accounting for physical failures.

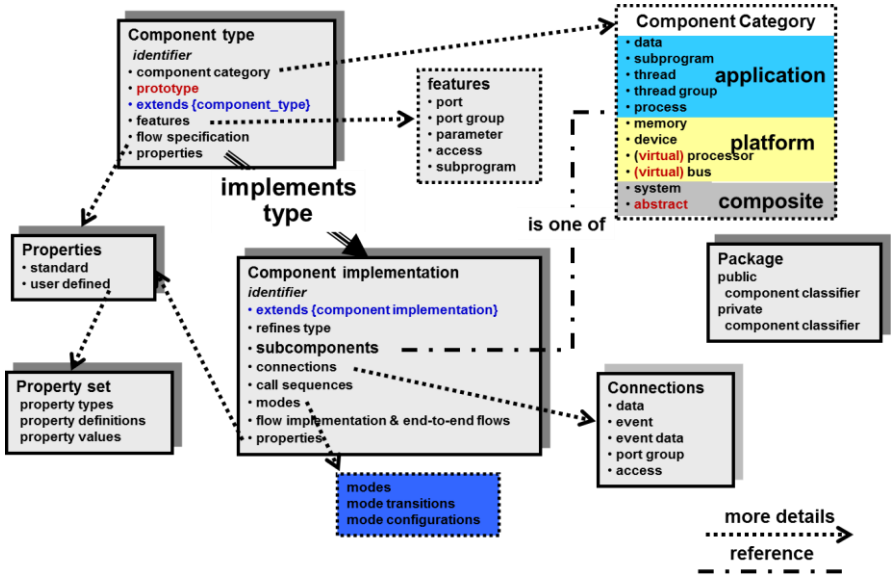


Figure courtesy of Peter Feiler: SAE AADL V2: An Overview

Figure 12: Component Types and Implementations in AADL

The type does not contain any of the internal structure of the component, however. Instead, *Implementations* of a type describe the internal structure of a component. To make this concrete, we examine a portion of our toy model from Chapter 2 in Figure 13. The top_level *system* defines two *ports:* Input, an **`in data port`** of type Integer, and Output, an **`out data port`** of type Integer. AADL defines three different kinds of ports: **`data port`**s, **`event port`**s, and **`event data port`**s. These ports have different semantics within AADL; data ports describe data that is periodically updated by a source process and sampled by a destination process. Event and event data ports cause events to be dispatched to a receiver process, which (usually) then executes to process the event.

For AGREE, since we abstract the timing model of the architecture, all of these port types are currently equivalent and all ports behave (roughly) as **`data port`**s. In future versions of AGREE, these ports will be distinguished and an accurate representation of the different behaviors will be supported.

```
system top_level
   features
      Input: in data port Base_Types::Integer;
      Output: out data port Base_Types::Integer;
   annex agree {**
      assume "System input range " : Input < 10;
      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;};
end top_level.Impl;
```

Figure 13: Integer_Toy model fragment.

In the system implementation, we see the decomposition of the top_level system into subsystems A, B, and C, and the connections between subcomponents and the top-level system interface. When connecting ports, AADL supports *properties* that allow aspects of the communication over the port to be further explained. In this model, each of the connections are *immediate* (that is, the data transfer occurs within the same frame); it is also possible to create a *delayed* connection, in which the output of the sender is buffered until the next frame.

NB: By default, AGREE assumes that connections are *delayed.* This default may be switched in the near future, so that the default is *immediate.* For now, the best practice is to explicitly state whether each connection is *immediate* or *delayed.*

NB: Currently in AGREE, the initial value of *delayed connections* is set to the "zero value" for the type: this is 0 for integers, 0.0 for reals, and false for Booleans. An option to change this value will be added to future versions of the tool.

From a synchronous dataflow perspective, an immediate connection occurs in the same time step and induces a dataflow relationship between the sender and the receiver. For example, since A_sub has an immediate connection to B_sub, B_sub must be evaluated "after" A_sub within the time step. The immediate connections have to form a *partial order*; that is, if *X* sends to *Y* through an immediate connection, then if *Y* also sends to *X*, it cannot do so through an immediate connection. Intuitively, if there were immediate connections in both directions, *X* would have to be scheduled before *Y* within the frame and vice versa.

Currently AGREE only supports port-based communications. In particular, it does not support remote-procedure-call (RPC-style) communication. This will be revisited in the future, but for the moment, the procedure call semantics require additional work to translate into our composition framework.

3.5.2 AGREE Subclauses and Declarations

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

```
annex agree {**
  -- agree declarations here…
**};
```

From within the subclause, it is possible to refer to the ports and properties of the enclosing component as well as the inputs and outputs of subcomponents. The top-level grammar for AGREE annex is shown in Figure 14Figure 14.

```
AgreeSubclause ::= (SpecStatement)+ ;

SpecStatement ::=
        'assume' STRING ':' Expr ';'
      | 'guarantee' STRING ':' Expr ';'
      | EqStatement
      | PropertyStatement
      | ConstStatement
      | FnDefExpr
      | NodeDefExpr
      | 'assert' Expr ';'
      | 'lift' NestedDotId ';'
      | LemmaStatement
      ;

LemmaStatement ::= 'lemma' STRING ':' Expr ';' ;
PropertyStatement ::= 'property' ID '=' Expr ';' ;
ConstStatement ::= 'const' ID ':' Type '=' Expr ';' ;
EqStatement ::= 'eq' Arg (',' Arg)* '=' Expr ';' ;
FnDefExpr ::= 'fun' ID '(' Arg (',' Arg)* ':' Type '=' Expr ';' ;
NodeDefExpr ::= 'node' ID '(' Arg (',' Arg)* ')' ':' 'returns'
                          '(' Arg (',' Arg)* ')' ';'
                NodeBodyExpr ;

Arg ::= ID ':' Type ;

NodeBodyExpr ::= ('var' (Arg ';')+ )?
                 'let' (NodeStmt)+ 'tel' ';' ;
NodeStmt ::=
        Arg (',' Arg)* '=' Expr ';'
      | LemmaStatement
```

Figure 14: AGREE Declaration Grammar

An AGREE subclause consists of a sequence of statements. The different kinds of statements and their use are described below.

3.5.3 Assume Statements:

Assume statements allow specification of environmental assumptions for a component. An example of an assume statement is:

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

The string `"System input range "` is used to identify the assumption when performing verification and the expression `Input < 10` defines the assumption itself.

3.5.4 Guarantee Statements:

Guarantee statements allow specification of the expected behavior of the component, if the component's assumptions are met. An example of a guarantee is:

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

The string `"System output range"` is used to identify the guarantee when performing verification. The expression `Output < 50` defines the guarantee expected of the component.

3.5.5 Eq Statements:

Equation statements can be used to create local variable declarations within the body of an AGREE subclause. An example of an Eq statement is:

```
eq ctr: int = prev(0, ctr + 1);
```

In this example, we create a variable that counts up from zero. **NB: for equations and nodes, currently the type names are different than the rest of the AADL model (this is an oversight that will be corrected shortly, but not in time for Seng5861 Fall 2013).** The currently supported types for equations and nodes are: int, real, and bool.

3.5.6 Property Statements:

Property statements allow specification of named Boolean expressions. An example property statement is:

```
property not_system_start_implies_mode_0 =
  not(OP_CMD_IN.System_Start) => (GPCA_SW_OUT.Current_System_Mode = 0);
```

Property statements are syntactic sugar (they are equivalent to defining an equation of type `bool`).

3.5.7 Const statements:

Const statements allow definition of named constants. An example const statement is:

```
const ADS_MAX_PITCH_DELTA: real = 1.0 ;
```

Currently, it is possible to specify constants for base types, but not for composite types (records and arrays).

3.5.8 Function Definitions:

Function definitions in AGREE allow specification of *non-recursive, pure* functions, that is, functions with no side effects. An example is shown below:

```
fun abs(x: real) : real = if (x > 0.0) then x else -x ;
```

This example defines the absolute value function. Functions in AGREE are very simple; recursive and iterative functions are not supported. For example the standard definition for Fibonacci numbers:

```
fun fib(x: int) : int =
   if (x <= 0) then -1
   else if (x = 1 or x = 2) then 1
   else fib(x-1) + fib(x-2);
```

would yield an error in AGREE.

3.5.9 Node Definitions:

Node definitions in AGREE allow specification of *stateful* definitions; that is, definitions that can maintain internal state. Nodes are a generalization of functions. It is probably most straightforward to describe via example. An example node for maintaining a generalized counter would be:

```
node Counter(init:int, incr: int, reset: bool)
      returns(count: int);
let
      count = if reset then init
                  else prev(count, init)+incr;
tel;
```

In this example, if reset is true, the counter is reset back to the `init` value. Otherwise, it increments by `incr`. The node maintains state (the value of `count` changes from time step to time step). It is then possible to instantiate this node in other expressions. For example:

```
eq x1 : int = Counter(0, 1, prev(x1 = 9, false));
eq x2 : int = Counter(1, prev(x2, 0), false);
```

Given these equations, x1 is a counter that repeatedly counts up to 9 then resets to zero, and x2 computes the Fibonacci series.

An example of a more complex node with multiple nodes, multiple outputs and local variables would be a 4-bit adder:

```
node ADD1(a: bool, b: bool, carry_in: bool) returns
         (out: bool, carry_out: bool);
let
  out = (a <> b) <> carry_in;
  carry_out = (a and b) or (a and carry_in) or (b and carry_in);
tel;

node ADD4 (a0: bool, a1: bool, a2: bool, a3: bool,
           b0 : bool, b1: bool, b2: bool, b3: bool) returns
          (s0 : bool, s1: bool, s2: bool, s3:bool, carry_out: bool);
var c0: bool;
    c1: bool;
    c2: bool;
    c3: bool;
let
  s0,c0 = ADD1(a0,b0,false);
  s1,c1 = ADD1(a1,b1,c0);
  s2,c2 = ADD1(a2,b2,c1);
  s3,c3 = ADD1(a3,b3,c2);
  carry_out = c3;
tel;
```

The ADD1 node takes two single bit inputs and a carry input bit and computes an output and a carry bit. We can use this to create a four bit adder ADD4 by "stringing together" four of the 1 bit adders. Note that all local variables (defined with **var**) and all output variables (defined in the **returns** section) must be assigned exactly one time within the **let** block.

The Recursive nodes, like recursive functions, are not supported.

3.5.10 Advanced Topic: Assert statements

Assert statements allow definition of axioms within the model. Axioms are "facts" about the behavior of the system or the environment that are added to the model to support proofs. An example assertion is of the form:

```
assert (FGS_L.LSO.Valid and FGS_R.LSO.Valid) =>
   FGS_L.LSO.Leader = FGS_R.LSO.Leader;
```

Assertions are sometimes used for *architectural patterns* whose correctness is established in a separate phase of analysis. The assertion above is from a pattern called *leader selection* that ensures that one of a set of redundant components is the leader.

NB: Assert statements are assumed to be true and are not validated in any way by AGREE. The use of such statements is extremely dangerous, and all assert statements should be examined by a domain expert and formal verification expert.

3.5.11 Advanced Topic: Lift statements

Guarantee statements in an AGREE annex are proven by using only direct evidence from some implementation's subcomponents. To help clarify this, consider the picture in Figure 15. In this Figure, the blocks represent different AADL system implementations. If AGREE is used to prove the Assumptions and Guarantees of system 1 (A_1 and G_1), AGREE will only use the assumptions and guarantees of system 2 and 3 as evidence (A_2, G_2, A_3, and G_4). The tool will ignore any of the constraints posed by systems 4 through 6. The philosophy of AGREE is that component contracts should yield the minimum constraints necessary in order to prove the guarantees of the direct parents. This aids the model checker by explicitly hiding information that may be unnecessary to prove top level claims.

However, in some cases proof of top level claims may require constraints imposed by components lower in the hierarchy. A user can manually add the assumptions and guarantees of leaf level components to their parents in the hierarchy to solve this problem, but this may require a lot of typing of similar guarantees. To make re-specification of guarantees less tedious, `lift` statements in the AGREE grammar.

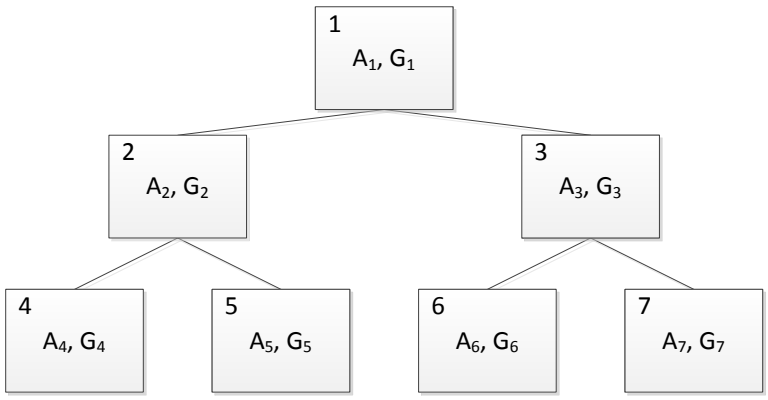


Figure 15: A Hierarchical Model

Consider the AADL model for the `External_Interface` shown in Figure 16. In this example a guarantee in the top level component (the `External_Interface`) requires constraints from a leaf level component (the `Sub_State_Machine`). In order to use the guarantee provided by the `Sub_State_Machine` a `lift` statement is used in the annex of the `State_Machine.Impl` component to effectively copy paste this guarantee in the body of the AGREE annex in the `State_Machine.Impl` component. Since the `State_Machine.Impl` component is a subcomponent of the `External_Interface.Impl` the contracts lifted into the `State_Machine.Impl` system implementation are used in the proof of the guarantees of the `External_Interface`. Lift statements can be chained through the hierarchy to pass constraints provided by contracts all the way from leaf level components to components in the top level of the heirarchy.

```
system External_Interface
```

```
	features
		State_Signal: out data port Types::state_sig.impl;

	annex agree {**
		guarantee "behavior" :
			State_Signal.val = prev(State_Signal.val,0);
	**};

end External_Interface;

system implementation External_Interface.Impl
	subcomponents
		SM: system Transmission::State_Machine.Impl;

	connections
		SSToSM: port SM.State_Out -> State_Signal
		{Communication_Properties::Timing => immediate;};
end External_Interface.Impl;



system State_Machine
	features
		State_Out: out data port Types::state_sig.impl;
end State_Machine;

system implementation State_Machine.Impl
	subcomponents
		SSM: system Sub_State_Machine.Impl;

	connections
		SMToSSM: port SSM.State_Out -> State_Out
		{Communication_Properties::Timing => immediate;};

	annex agree {**
		lift SSM;
	**};

end State_Machine.impl;

system Sub_State_Machine
	features
		State_Out: out data port Types::state_sig.impl;

	annex agree {**
		guarantee "sub behavior" :
			State_Out.val = prev(State_Out.val, 0);
	**};
end Sub_State_Machine;

system implementation Sub_State_Machine.Impl

	annex agree {**
		assert State_Out.val = prev(State_Out.val,0);
	**};
```

```
end State_Machine.impl;
```

Figure 16: An AADL Model Using a "Lift" Statement.

3.5.12 Advanced Topic: Lemma Statements

Assert statements are used to introduce lemmas to assist the model checker when performing verification. AGREE uses *k-induction over the transition relation* to try to prove properties – see Appendix A for a high-level description of the procedure. For many systems and properties, this works very well and is able to prove interesting properties about the system without assistance. However, sometimes a property is *true* but not *provable* using this technique. The reason that this happens is the property to be proved is too weak to be inductively provable. Lemma statements are additional properties that are added to an AGREE model in order to *strengthen* the property to be proved.

An example lemma would be:

```
lemma "drug flow lemma" :
      (not drug_flow_stopped) => spo2_never_below_thresh ;
```

From the perspective of proof, lemmas behave the same as guarantees; they must be proven by AGREE. However, unlike guarantees, lemmas are not made visible when trying to prove properties at the next level of abstraction.

3.6 AGREE Package Subclauses

AGREE subclauses can occur either within AADL components or at the top-level of a package. The component-level subclauses have been well explained in previous sections of the document.

Package-level subclauses are designed to provide reusable libraries of definitions for AGREE. Nodes, Functions, and constants in these subclauses can be referenced by component-level subclauses by using the dot notation: <Package_Name>.<definition name>. So, for example, the equation:

```
eq x1 : int = Agree_Common.Counter(0, 1, prev(x1 = 9, false));
```

Uses the Counter node defined in the Agree_Common package.

4 Using the AGREE/OSATE Tool Suite

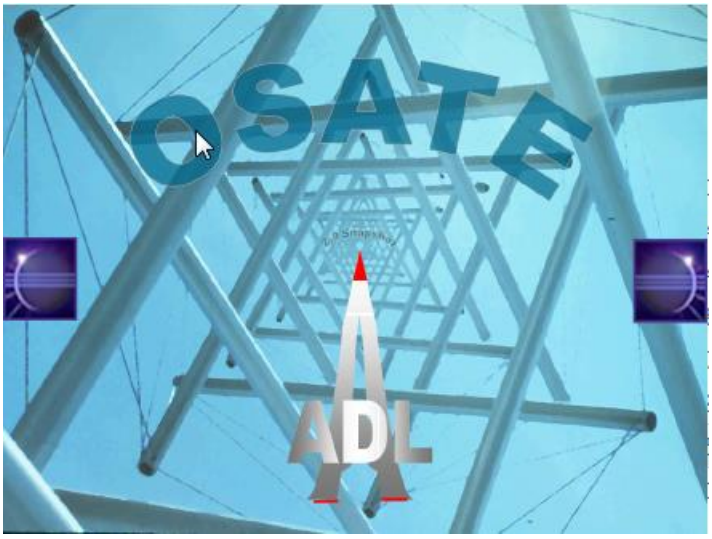
4.1 Installation

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

4.1.1 Install OSATE

Binary releases of the OSATE tool suite for different platforms are available at: http://www.aadl.info/aadl/osate/stable/. At the time of writing this document, the most current release of OSATE is 2.0.4; the plugins have been tested against the 2.0.4 and 2.0.3 releases of OSATE. To download version 2.0.4, use the URL: http://www.aadl.info/aadl/osate/stable/2.0.4/products/. Choose the version of OSATE that is appropriate for your platform. There are two binary versions for Windows: osate2-2.0.4-win32.win32.x86.zip and osate2-2.0.4-win32.win32.x86_64.zip; the first is for 32-bit Windows and the second is for 64-bit Windows.

Once the .zip file is downloaded, all that is required is to unzip it into a location in the file system. One candidate location for Windows is C:\apps\osate, but any location in the file system that is write-accessible is fine. After expanding the .zip file, navigate to the osate.exe file and double-click it. The following splash screen should appear, and OSATE should begin loading:



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 7, this information can be found in the System Control Panel as shown below in Figure 17. Note that this information is also required for downloading the correct version of the Yices tool in the next installation step.

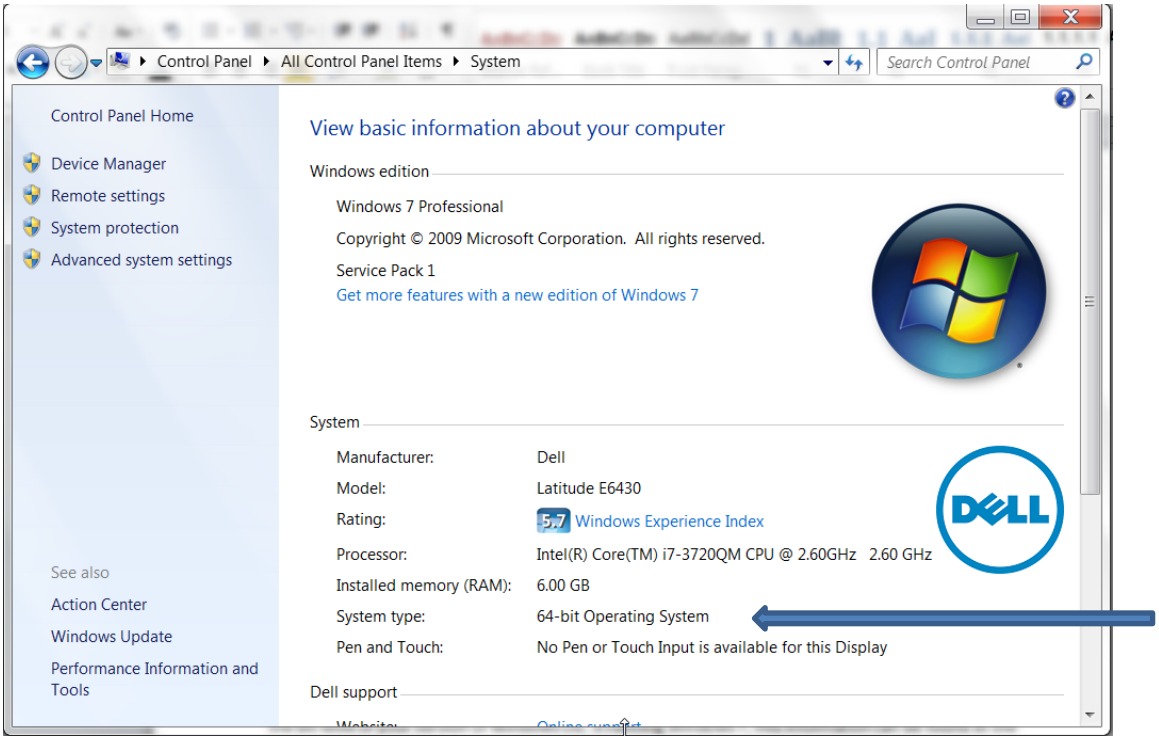


Figure 17: Windows OS Version and Bit size information

4.1.2 Install Yices 1

Yices from SRI, Inc. is the underlying symbolic solver that is used by AGREE and the jkind model checker. Due to licensing restrictions on yices 1, it is not possible to include the solver with our distribution and it must be downloaded separately.

Navigate to the Yices install page at: http://yices.csl.sri.com/download.shtml and download the version of yices appropriate for your platform.

Yices must be unzipped and placed in a directory somewhere in the file system. Then this directory must be added to the system path. In Linux, you must add the path to your config file, usually .bashrc. If you are running linux, I will assume that you are savvy enough to do this ☺.

In Windows, this requires navigating to the System Control Panel shown in Figure 17 and choosing the "Advanced system settings" button on the left side of the panel. The system properties dialog will appear. Choose the "Advanced" tab as shown in Figure 18, then click "Environment variables".

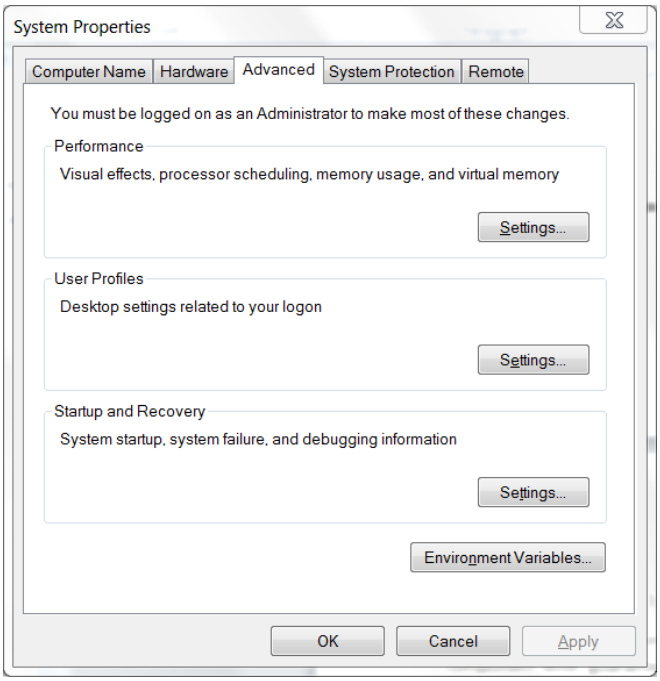


Figure 18: System Properties Dialog Box

The environment variables dialog box is shown in Figure 19. 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. If the path string does not end with a semicolon (';'), add one, then add the path to the 'bin' directory underneath the main yices directory.

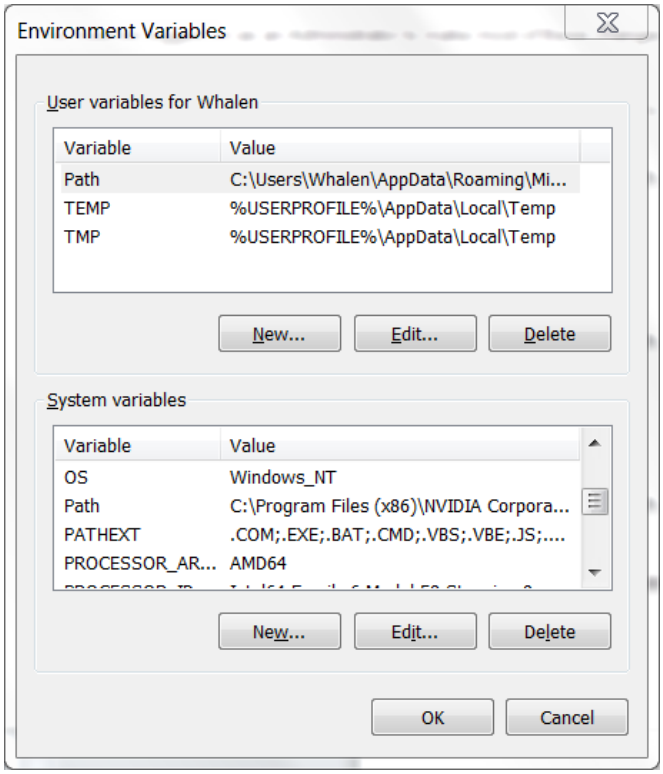
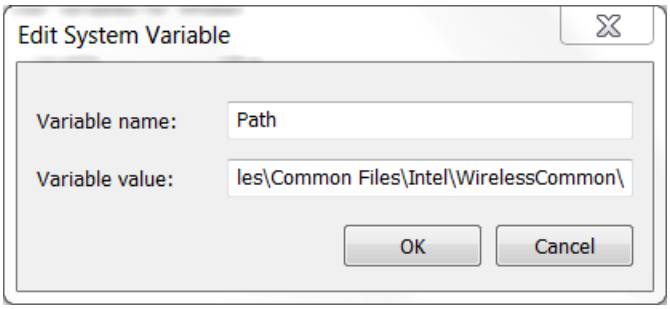
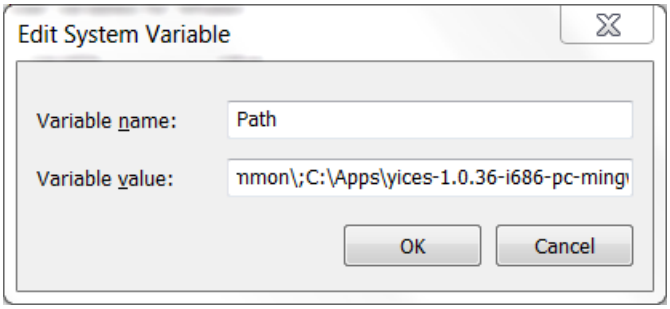


Figure 19: Environment Variables Dialog Box

For example, I installed yices into C:\apps. After navigating the directory structure generated by the unzipping process, the bin directory is: C:\Apps\yices-1.0.36-i686-pc-mingw32\yices-1.0.36\bin. I started with path:



I then added the path to the yices bin directory as follows:



I then clicked 'OK' on this dialog and the windows in Figure 19 and Figure 18.

To test whether yices has been correctly installed on either Windows or Linux, open up a command prompt window and type: `yices --version`. A number of the form 1.0.xx (where xx is something greater than or equal to 29) should be displayed, matching the installed version of yices. At the time this document was written, the current version of yices is 1.0.39.

4.1.3 Install jkind

Download the latest release of jkind at: https://github.com/agacek/jkind/releases and unzip it into a location in the file system. Place the directory containing jkind.exe on your path using the same technique that was described for installing yices.

To test whether jkind has been successfully installed, open a new command window and type 'jkind'. You should see something like the following:

```
usage: jkind [options] <input>
 -blame            generalize counter examples
 -bmc              bounded model checking only (implies -no_inv_gen)
 -excel            generate results in Excel format
 -help             print this message
 -induct_cex       generate inductive counterexamples
 -n <arg>          number of iterations (default 200)
 -no_inv_gen       disable invariant generation
 -reduce_inv       reduce and display invariants used
 -scratch          produce files for debugging purposes
 -smooth           smooth counterexamples (minimal changes in input values)
 -solver <arg>     SMT solver (default: yices, alternatives: cvc4, z3)
 -timeout <arg>    maximum runtime in seconds (default 100)
 -version          display version information
 -xml              generate results in XML format

C:\apps >
```

4.1.4 Install AGREE

Download the latest release of AGREE at: https://github.com/smaccm/smaccm/releases and unzip it into a location in the file system. Unzipping the file should create a directory called 'dropins' containing

a set of .jar files. Copy this directory to the osate2 directory; it should be a sibling of the 'plugins' directory and the osate.exe file as shown in Figure 20.

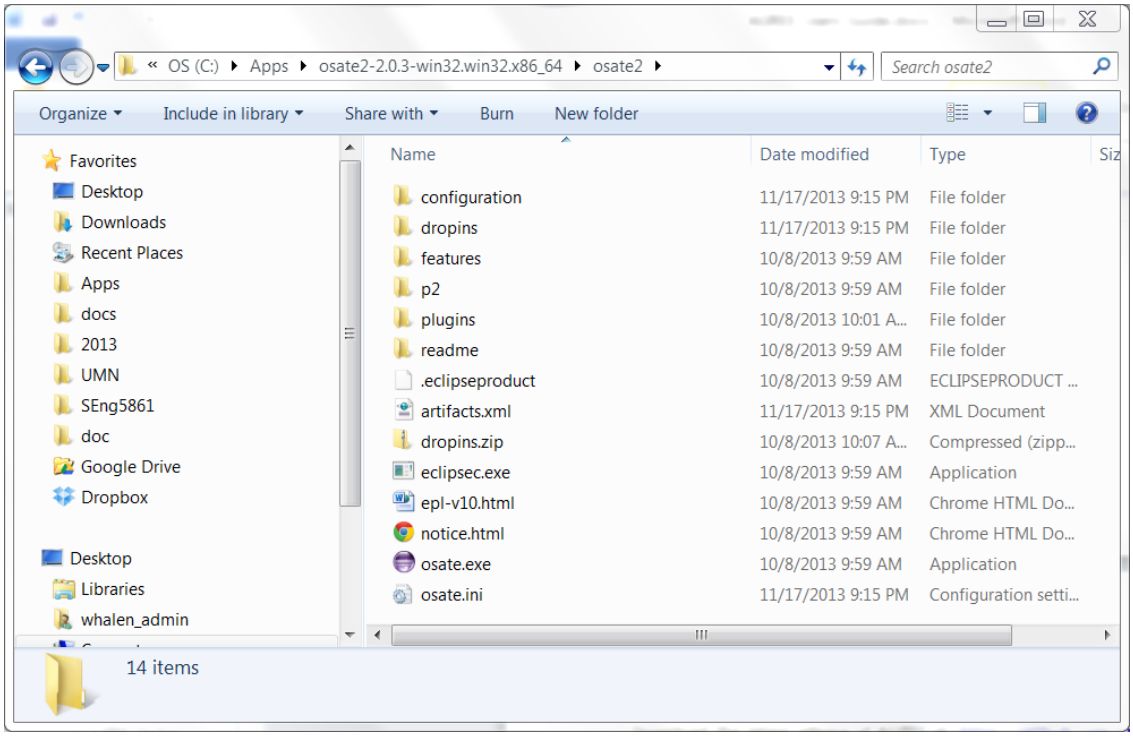


Figure 20: OSATE Directory with dropins directory added.

To test whether AGREE has been correctly installed, start OSATE. If it has been correctly installed, an AGREE menu should appear in OSATE, as shown in Figure 21. At this point, you should be ready to go!

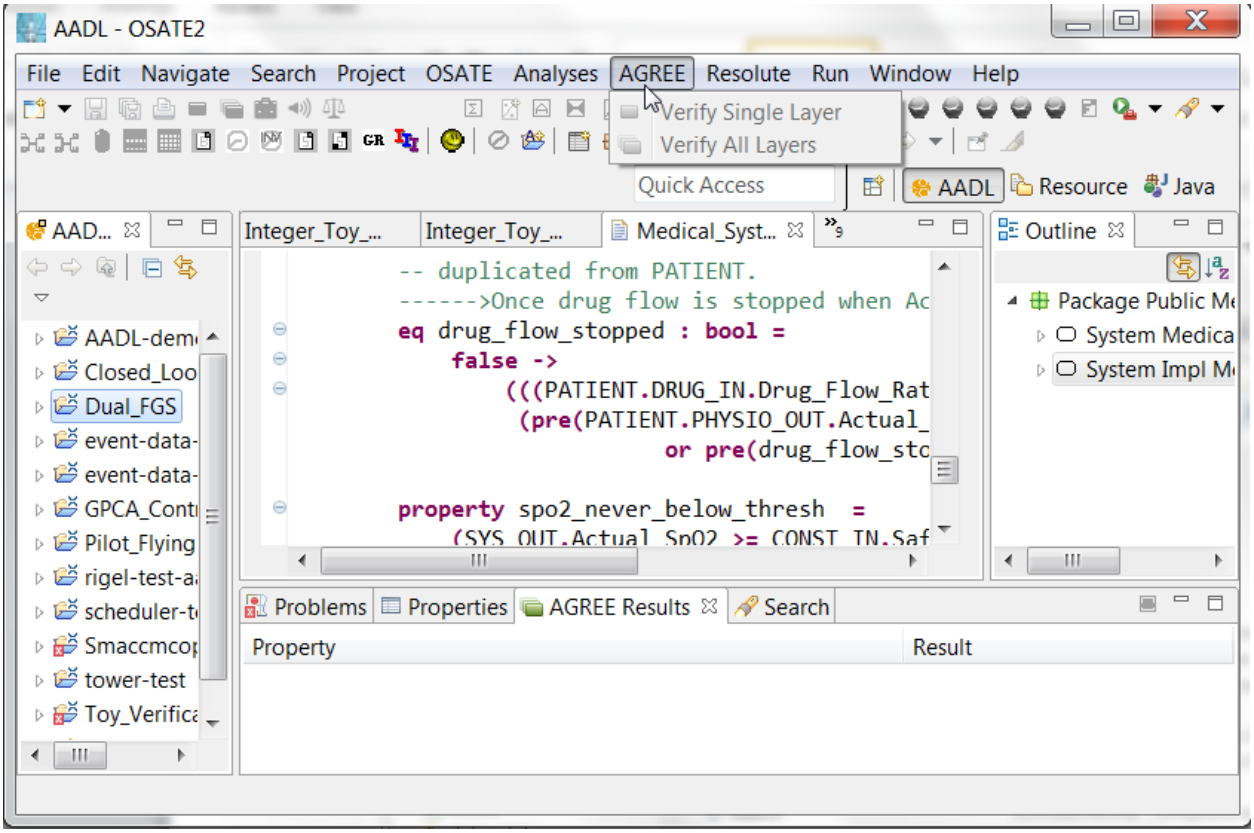
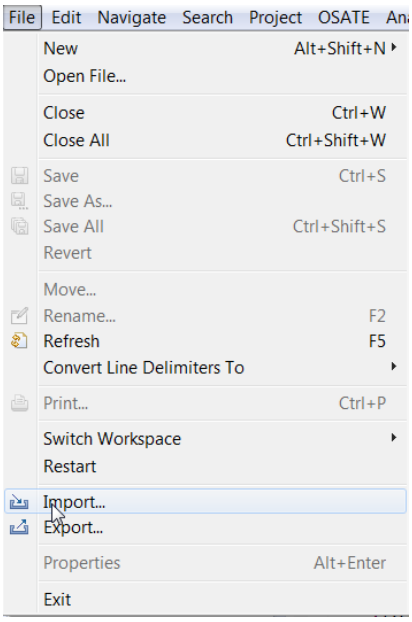


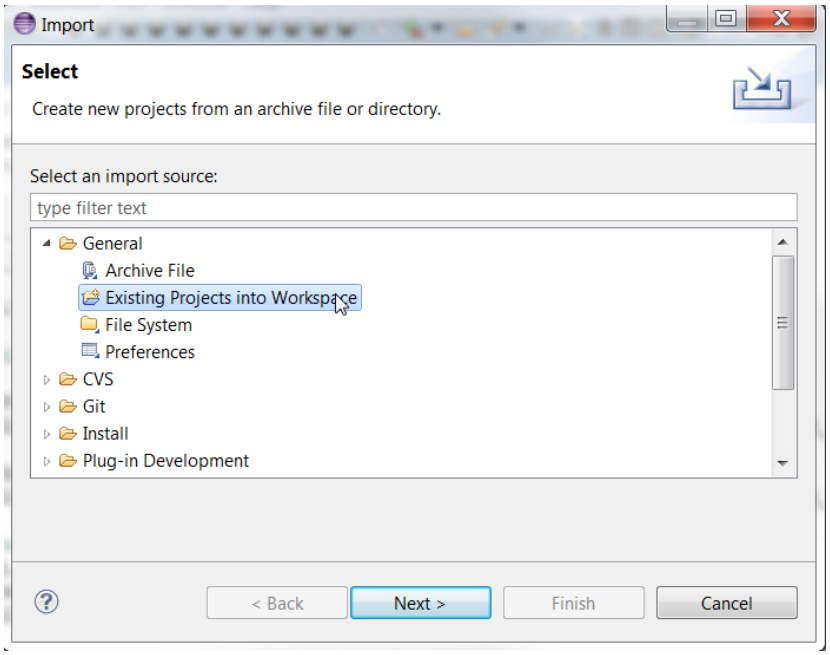
Figure 21: AGREE Install Test

4.2 Importing Archived Projects into AGREE

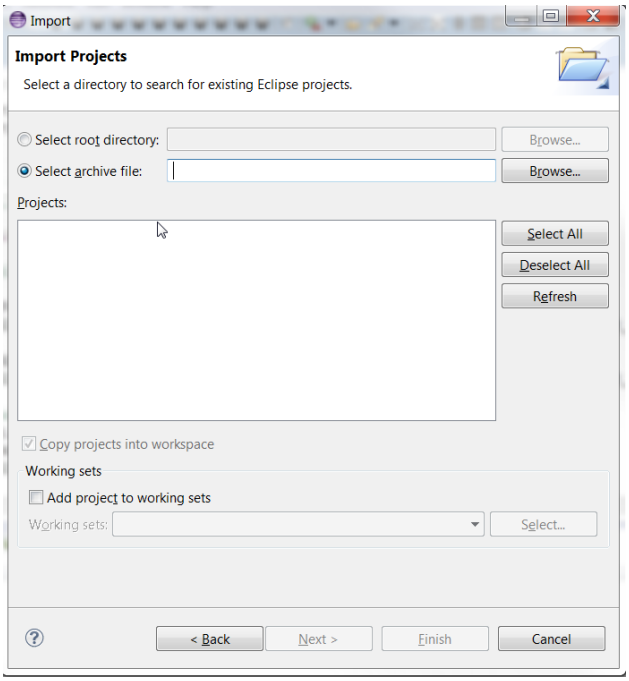
To import an archived project into AGREE, all that is required is to “import” it using the File menu:



Then, from the Import dialog box, choose “Existing Projects into Workspace” and click Next



Choose “Select archive file” and navigate to the location containing the .zip file:



Then navigate to the directory containing the .zip file and choose “Finish”. You should now see the project in the AADL Navigator in the left-hand-side pane in Eclipse.

4.3 Using AGREE

This section is TBD. For the purposes of the exercises in Seng5861, the tool guidance in Chapter 2 should be sufficient to use the tool.

Appendix A On K-Induction

The AGREE tool framework uses *induction* to try to prove the system level-guarantees from the component-level guarantees. But what does this mean? To explain, we first refresh the user's understanding of mathematical induction performed over natural numbers. Often, one wishes to prove a mathematical fact of the following sort:

$\sum_{x=1}^{n} x = \frac{(n+1)*n}{2}$

We can prove this by *weak induction.* This involves two steps: first, a base case, where we show that the property holds for the initial value (in this case, the value 1), and an inductive case, where if we assume the property is true of *n*, we prove that it is true of *n+1.* For this example, the base case is

$\sum_{x=1}^{1} x = \frac{(1+1)*1}{2}$

Since $1 = \frac{2}{2}$, we satisfy the base case. If we assume that the property is true of *n,* we can prove the inductive case over (n+1) as follows:

$$\sum_{x=1}^{(n+1)} x = \frac{((n+1)+1)*(n+1)}{2}$$

= def. of summation

$$\sum_{x=1}^{(n)} x + (n+1) = \frac{((n+1)+1)*(n+1)}{2}$$

= induction hypothesis

$$\frac{(n+1)*n}{2} + (n+1) = \frac{((n+1)+1)*(n+1)}{2}$$

= arithmetic expansion

$$\frac{n^2+n}{2} + \frac{2n+2}{2} = \frac{n^2+3n+2}{2}$$

= arithmetic equalities

$$\frac{n^2+3n+2}{2} = \frac{n^2+3n+2}{2}$$

QED.

The induction principle used by AGREE is similar. However, instead of performing induction over natural numbers, it performs induction over the *transition system* that defines the properties. Through a

compilation step, any AGREE model can be turned into a complex first-order logical formula that defines how the system can evolve from one time instant to the next time instant, denoted *T*; the formula T is defined over a set of *pre-state* variables and a set of *post-state* variables, that describe the values of the variables in the model in the state before and after the transition. This idea is not entirely straightforward, but a full explanation is outside the scope of this User's Guide. For a complete explanation please see *Model Checking* by Ed Clarke et. al or *Logic in Computer Science* by Huth and Ryan.

Using this notation, and a formula *I* that defines the set of allowed initial values for variables, you can describe the evolution of the system as follows:

$$I(s_0)\ \&\ T(s_0, s_1)\ \&\ T(s_1, s_2)\&\ T(s_2, s_3)\&\ \ldots$$

Where I defines the initial constraint on the variables and the T's define the step-to-step evaluation of the system. This provides a structure from which you can perform induction. Suppose you define a property that you want to hold over a system state as *P(s).* Then it is possible to talk about performing induction over this structure.

[more here]

Appendix B Annex B: Data Modeling

Normative

Annex B.1Scope

(1) The Data Modeling Annex document provides guidelines to model data types as part of an AADL architectural model and guidelines to map data models expressed in other modeling notations into an architecture model expressed in AADL.

(2) The purpose of AADL is to model the computer based embedded system, including the runtime architecture of embedded systems. In that context AADL provides the component category data, whose role is to represent data components with shared access, the data type of such data components, as well as the data type of data communicated through data ports, event data ports, and subprogram parameters. AADL supports implementation declarations for data components, which can represent the elements of a data type as subcomponents. Its intent is to represent the composition of a data component out of elements when relevant to the architecture, such as the aggregation of output from several sources to be communicated as a single aggregate, or concurrent access to shared data managed at the level of elements of a data component

(3) AADL allows one to capture some abstractions of the data model. The core language standard specifies that by default the data type name in the source text is assumed to be the same as the identifier of the AADL data component. If necessary, the Source_Name property can be used to define a mapping from the AADL identifier to a name in the source text. Other information about the data types and data components in the source text can be captured through other properties, such as Source_Data_Size to indicate the amount of memory an instance of the data type takes. It is expected that there is an equivalent data type in the source text expressed in a source language

such as C or Ada, in an equivalent notation for modeling frameworks like SCADE, SDL or Simulink, or data modeling notations (e.g. OMG IDL or ITU- T ASN.1).

(4) In some occasions, users may want to record data modeling related information that is relevant for the runtime architecture as part of an AADL model to support checking of architectural consistency, consistency between architecture information and the source text, and generation of a runtime system from AADL models. The Data Modeling annex document is intended to support these objectives together with the Code Generation annex document.

Annex B.2Modeling Data Types in AADL

(5) The Data Model annex document provides guidelines for data modeling in the following way. User-defined AADL data component types represent application data types. These data component types are then annotated with properties to indicate relevant details of the data type in a data modeling language or source language. The property `Data_Representation` specifies the realization of the data type in terms of basic data types built into the language, such as real, integer, string, and Boolean, and in terms of composite data types, such as *array* (ordered sets accessible by index), *struct* (collection of named elements), and *union* (collection of named alternatives).

(6) In the case of *array* or *string*, additional properties characterize the size of the array in terms of multiple dimensions (`Dimension`) and the data type of the array elements (`Base_Type`). Note that the type of the array elements is characterized in terms of an AADL defined data component type. This allows arrays of user-defined types as well as base types to be modeled. The Data Modeling Annex includes a package (`Base_Types`) of predeclared AADL data component types (and implementations) for a set of data types that represent built-in data types. Those data component types are themselves characterized by the `Data_Representation` property as representing a built-in data types such as *integer*, *fixed*, *float*, *string*, *character*, *enum*, *Boolean*.

(7) In the case of *struct* or *union*, the designer has two options. First option is to specify the data types of the elements by an ordered list of classifiers as the `Base_Type` property value. The order of the list determines the order of the data elements. If the names of the data elements are relevant, then they can be specified by the `Element_Names` property as an ordered list of names. In this case, the designer only specifies which data are available; there is no mechanism to reference or interact with entities inside this data component at model-level.

(8) Second option is to use AADL mechanism to provide access to data type members or access methods to the data type represented by the data model. Data type members are expressed as subcomponents and access wethods are expressed as provides subprogram access features in the AADL data type and implementation declaration.. Explicitly declaring members as data subcomponents gives more control on the architectural mapping of the members, such as memory layout, or access policies.

Note: these two options are notionally equivalent. No matter which option is chosen in the declarative AADL model, the second option (use of subcomponents) is the one used internally in the AADL processing tools to represent and manipulate arrays and structures. This allows treating both options in a uniform way.

(9) The following properties further characterize representation of numeric values: `Data_Digits`, `Data_Scale` for specifying the precision of fixed-point types, `IEEE754_Precision` for float types and `Number_Representation` for integer types. Three other properties are intended to specify constraints on the numerical data, namely range constraints (`Real_Range`, `Integer_Range`) and the measurement unit in which the data values are intended to be interpreted (`Measurement_Unit`).

(10) The `Code_Set` property allows for the specification of the character set to be used in string and character data types.

(11) The `Enumerators` property allows for the specification of the literal values to be used in an enumeration data type.

(12) The low-level representation of Enumerators can be defined using the `Representation` property.

(13) The `Initial_Value` property allows for specification of the initial value of a data component.

Note: AADLV2 proposes standard properties to define data types. For instance, the fact that a data component in the source text represents a constant is recorded by the `Access_Right` property with the value *Read_Only*. Other properties exist to specify the memory requirements of a data type, the concurrency protocol to access it, etc.

(14) An implementation method may restrict the combinations of properties from this property set, and the property from the AADL core.

Note: the rationale is to ensure there is one single definition of a data type, either as defined using this annex, or using core mechanisms with `Source_Language`, `Type_Source_Name`.

Annex B.3Data Modeling Property Set

property set Data_Model **is**

`Base_Type` **: list of classifier (** `data` **)** **applies to** (`data`); The `Base_Type` property specifies the base type of a data component type. The classifiers being referenced are those defined in the `Base_Types` package or from user defined packages.
`Code_Set` **: aadlinteger** **applies to** (`data`); The `Code_Set` property is used to specify the code set used to represent a character or a string. The value applied is the registered value affected and defined in the *"OSF character and Code Set Registry"* by the OSF. This document is available at http://www.opengroup.org/dce/info/.
`Data_Digits` **: aadlinteger** **applies to** (`data`); The `Data_Digits` property specifies the total number of digits of a fixed-point type.
`Data_Scale` **: aadlinteger** **applies to** (`data`); The `Data_Scale` property defines the scale of the fixed-point types (10**(-scale) is the precision).
`Data_Representation` **: enumeration** (`Array, Boolean, Character, Enum, Float, Fixed, Integer, String, Struct, Union`) **applies to** (`data`); The `Data_Representation` property may be used to specify the representation of simple or composite data types within the programming language source code.

Note: An implementation is allowed to support only a subset of these types.

`Dimension` : **list of aadlinteger**

applies to (data);

The `Dimension` property is used to specify the dimensions of a multi-dimensional array, the i^{th} value in the list representing the i^{th} dimension of the array. This property shall be used in conjunction with the `Data_Representation` property.

`Element_Names` : **list of aadlstring**

applies to (data);

The `Element_Names` provides the names of a struct or union members in order of appearance as defined by the `Base_Type` property.

`Enumerators` : **list of aadlstring**

applies to (data);

The `Enumerators` provides the list of enumeration literals attached to an enumeration data component.

`IEEE754_Precision` : **enumeration** (Simple, Double)

applies to (data);

The `IEEE754_Precision` property indicates, for a float data component type, the precision used. This property is derived from the notion of precision per the 754-1985 IEEE Standard for Binary Floating-Point Arithmetic.

`Initial_Value` : **list of aadlstring**

applies to (data, port, parameter);

`Initial_Value` specifies a list of initial values for a data component or port in string form. For a subprogram parameter, it defines a default value.

It can be used to represent initial values other than strings as string. This (list of) string is interpreted by the source language processor. In this case, the core AADL language processor does not check consistency of the initial values with the data type of the data component.

`Integer_Range` : **range of aadlinteger**

applies to (data, port, parameter);

`Integer_Range` specifies a range of integer values that apply to the data component. This property is used to represent integer range constraints on data that is of some integer type.

`Measurement_Unit` : **aadlstring**

applies to (data, port, parameter);

The `Measurement_Unit` property specifies the measurement unit of the data being communicated. A full list of recommended name for units, as part of the SI standard is proposed by the International Bureau of Weight and Measures: http://www.bipm.org/en/si/.

Note: this should not be mixed with AADL units, which specify units to be applied to properties, e.g. thread's period, bandwidth usage that belong solely to the architecture modeling

`Number_Representation` : **enumeration** (Signed, Unsigned)

applies to (data);

`Number_Representation` specifies whether an integer data component is signed or unsigned.

Real_Range: **range of aadlreal**

applies to (data, port, parameter);

Real_Range specifies a range of real values that apply to the data component. This property is used to represent real range constraints on data that is of some real type.

Representation : **list of aadlstring**

applies to (data);

Representation specified the actual representation of enumerator's value.

end Data_Model;

Annex B.4Predeclared AADL Package for Basic Data Types

(15) This AADL package defines a set of basic types for use as data component types in AADL models. Typically, they can be used to represent the base type of a data component or port.

Note: These data types are different from the built in property types such as **aadlinteger**, **aadlreal**, **aadlboolean**, or **aadlstring**. The latter denote types manipulated to represent entities of the AADL model, whereas the types defined below represent types manipulated by the system being modeled.

(16) This package belongs to the set of models defined by the AADL V2. Implementations of AADL tools are allowed to add additional properties to clarify some elements of a data component (e.g. project specific limits) or to help model processing.

(17) Implementations are not allowed to modify the values of the properties defined for the types defined below.

(18) A method of implementation may restrict the types from this package that are supported, e.g. to generate code, to reflect implementation limits.

```
package Base_Types
public
  data Boolean
  properties
    Data_Model::Data_Representation => Boolean;
  end Boolean;

  data Integer
  properties
    Data_Model::Data_Representation => Integer;
  end Integer;

  -- Signed integer of various byte sizes

  data Integer_8 extends Integer
  properties
    Data_Model::Number_Representation => Signed;
```

```
  Source_Data_Size => 1 Bytes;
end Integer_8;

data Integer_16 extends Integer
properties
  Data_Model::Number_Representation => Signed;
  Source_Data_Size => 2 Bytes;
end Integer_16;

data Integer_32 extends Integer
properties
  Data_Model::Number_Representation => Signed;
  Source_Data_Size => 4 Bytes;
end Integer_32;

data Integer_64 extends Integer
properties
  Data_Model::Number_Representation => Signed;
  Source_Data_Size => 8 Bytes;
end Integer_64;

-- Unsigned integer of various byte sizes

data Unsigned_8 extends Integer
properties
  Data_Model::Number_Representation => Unsigned;
  Source_Data_Size => 1 Bytes;
end Unsigned_8;

data Unsigned_16 extends Integer
properties
  Data_Model::Number_Representation => Unsigned;
  Source_Data_Size => 2 Bytes;
end Unsigned_16;

data Unsigned_32 extends Integer
properties
```

```
  Data_Model::Number_Representation => Unsigned;
  Source_Data_Size => 4 Bytes;
end Unsigned_32;

data Unsigned_64 extends Integer
properties
  Data_Model::Number_Representation => Unsigned;
  Source_Data_Size => 8 Bytes;
end Unsigned_64;

data Natural extends Integer
properties
  Data_Model::Integer_Range => 0 .. Max_Target_Integer;
end Natural;

data Float
properties
  Data_Model::Data_Representation => Float;
end Float;

data Float_32 extends Float
properties
  Data_Model::IEEE754_Precision => Simple;
  Source_Data_Size => 4 Bytes;
end Float_32;

data Float_64 extends Float
properties
  Data_Model::IEEE754_Precision => Double;
  Source_Data_Size => 8 Bytes;
end Float_64;

data Character
properties
  Data_Model::Data_Representation => Character;
end Character;

data String
properties
```

```
    Data_Model::Data_Representation => String;
  end String;
end Base_Types;
```

Annex B.5Examples

(19) This AADL package lists some examples of the AADL modeling property sets.

```
package Base_Types::Example_Types
public
  with Data_Model, Base_Types;

  data One_Dimension_Array
  properties
    Data_Model::Data_Representation => Array;
    Data_Model::Base_Type => (classifier (Base_Types::Integer));
    Data_Model::Dimension => (42);
  end One_Dimension_Array;

  data                                                Two_Dimensions_Array
  properties
    Data_Model::Data_Representation                 =>                  Array;
    Data_Model::Base_Type      =>      (classifier      (Base_Types::Integer));
    Data_Model::Dimension              =>              (74,               75);
  end Two_Dimensions_Array;

  -- Two equivalent ways to define a structure with named
  -- elements

  data A_Struct1
  properties
    Data_Model::Data_Representation => Struct;
    Data_Model::Base_Type => (classifier (Base_Types::Float),
                              classifier (Base_Types::Character));
    Data_Model::Element_Names => ("f1", "c2");
  end A_Struct1;

  data A_Struct2
  properties
    Data_Model::Data_Representation => Struct;
  end A_Struct2;
```

```
data implementation A_Struct2.impl
subcomponents
  f1 : data Base_Types::Float;
  c2 : data Base_Types::Character;
end A_Struct2.impl;

--  Two equivalent ways to define a union with named elements

data A_Union1
properties
  Data_Model::Data_Representation => Union;
  Data_Model::Base_Type => (classifier (Base_Types::Float),
                            classifier (Base_Types::Character));
  Data_Model::Element_Names => ("f1", "f2");
end A_Union1;

data A_Union2
properties
  Data_Model::Data_Representation => Union;
end A_Union2;

data implementation A_Union2.impl
subcomponents
  f1 : data Base_Types::Float;
  c2 : data Base_Types::Character;
end A_Union2.impl;

data An_Enum
properties
  Data_Model::Data_Representation => Enum;
  Data_Model::Enumerators => ("foo", "bar");
  Data_Model::Representation => ("00", "11");
end An_Enum;

data A_Fixed_Point
properties
```

```
  Data_Model::Data_Representation => Fixed;
  Data_Model::Data_Digits => 18;
  Data_Model::Data_Scale => 8;
end A_Fixed_Point;

data A_Kanji extends Base_types::Character
properties
  Data_Model::Code_Set => 16#30001#;
  -- ISO code for Japanese characters
end A_Kanji;

data Russian_Strings extends Base_Types::String
properties
  Data_Model::Code_Set => 16#15#;
  -- ISO code for Cyrillic characters
end Russian_Strings;

data Bounded_Strings extends Base_Types::String
properties
  Data_Model::Dimension => (10);
end Bounded_Strings;

-- Representation of a type defined in another modeling
-- notation, or programming language

data ASN1_Type
  properties
    Source_Language => ASN1;
    Source_Text => ("ASN1_types.asn");
    Type_Source_Name => "the_type";
end ASN1_Type;

data C_Type
properties
  Source_Language => C;
  Source_Text => ("types.h");
  Type_Source_Name => "the_type";
```

```
  end C_Type;

end Base_Types::Example_Types;
```