AGREE User Guide

Contents

List of Tables

List of Figures

Chapter 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: Section 2 provides a brief overview of

AADL and AGREE through a small example. Section 3 describes the syntax of the AGREE language. Section 4 describes the installation information and main features of the AGREE tool suite.

Chapter 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 2.1.

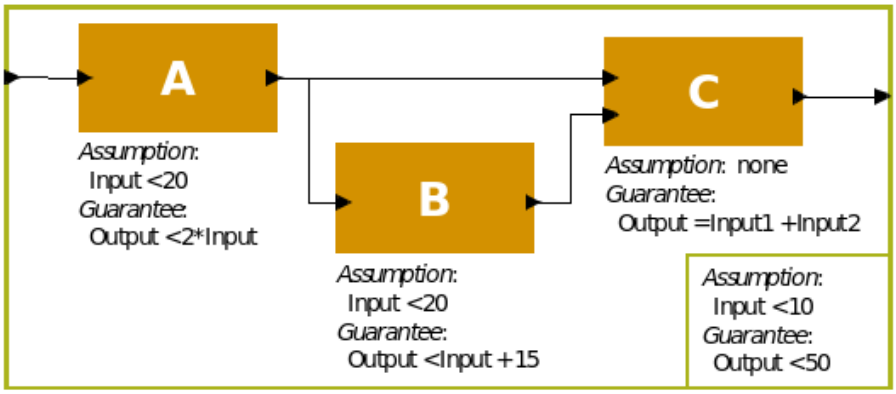


Figure 2.1: Toy Compositional Proof Example

In the model in Figure 2.1, 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 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 domain" : 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 domain" : 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 domain" : 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;
```

In the code above, **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 Section 3. 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

The example project used in the rest of this section can be retrieved from "AGREE Toy Example" (https://github.com/smaccm/smaccm/blob/master/models/Toy_AGREE_Models/T

After unzipping the model, it can be imported by choosing File > Import:

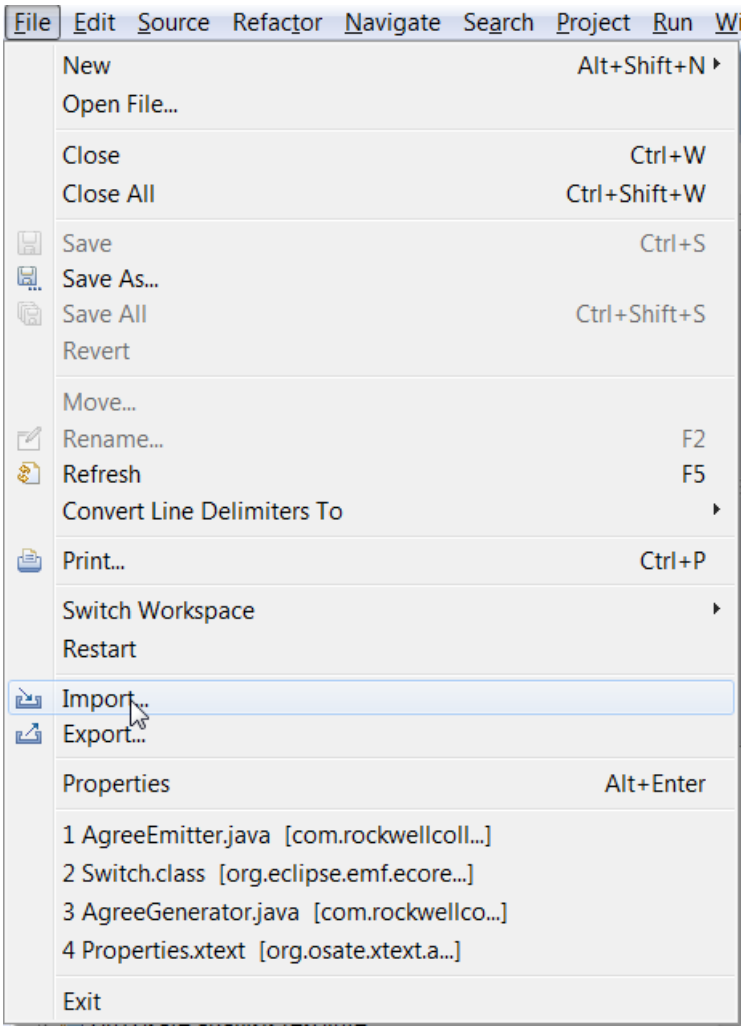


Figure 2.2: Import Menu Option

Then choosing "Existing Project into Workspace"

and navigating to the unzipped directory after pressing the Next button. Figure 2.4 shows what the model looks like when loaded in the AGREE/OSATE tool.

Note that in the workspace in Figure 2.4, there are several projects, so your workspace will probably look slightly different. The project that we are working with is called Toy_Example.

Open the Integer_Toy.aadl model by double-clicking on the file in the AADL

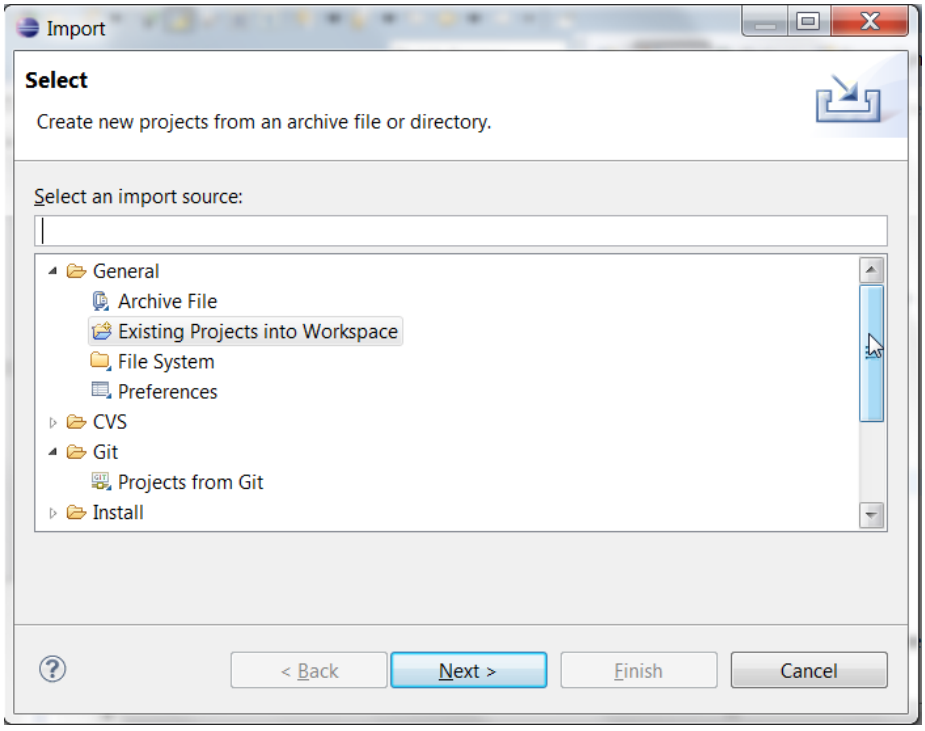


Figure 2.3: Importing Toy_Verification Project

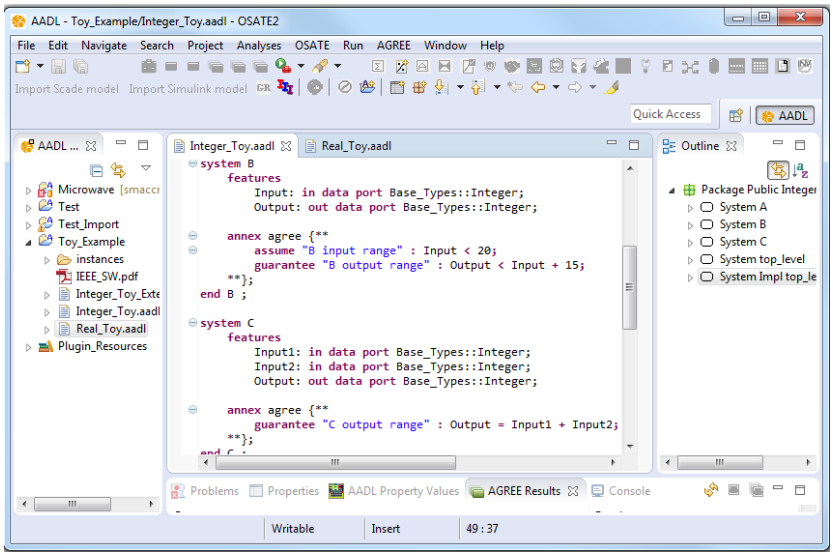


Figure 2.4: AGREE/OSATE Environment with toy example

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":

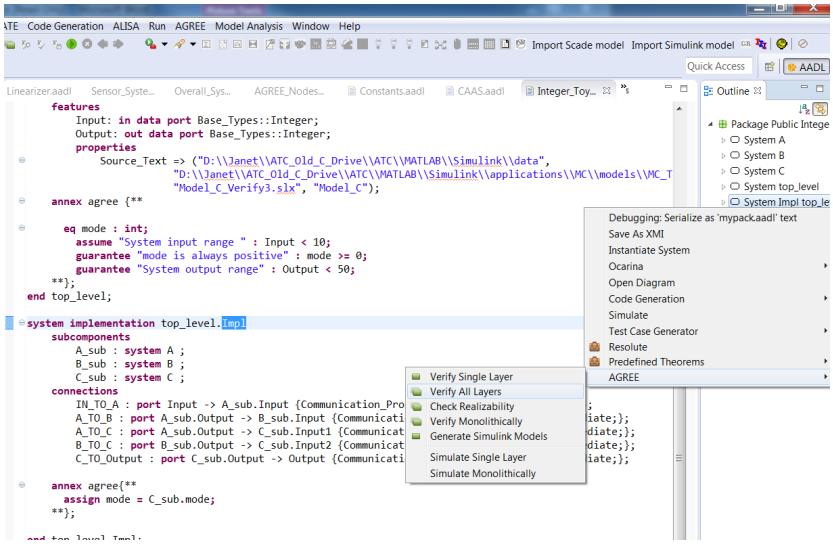


Figure 2.5: Verify All Layers Option from Right Click Menu

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 2.7.

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 2.8.

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

For working with complex counterexamples, it is often necessary to have a richer interface. It is also possible to export the counterexample to Excel by right-clicking the failing property and choosing "View Counterexample in Excel". **Note: In order to use this capability, you must have Excel installed**

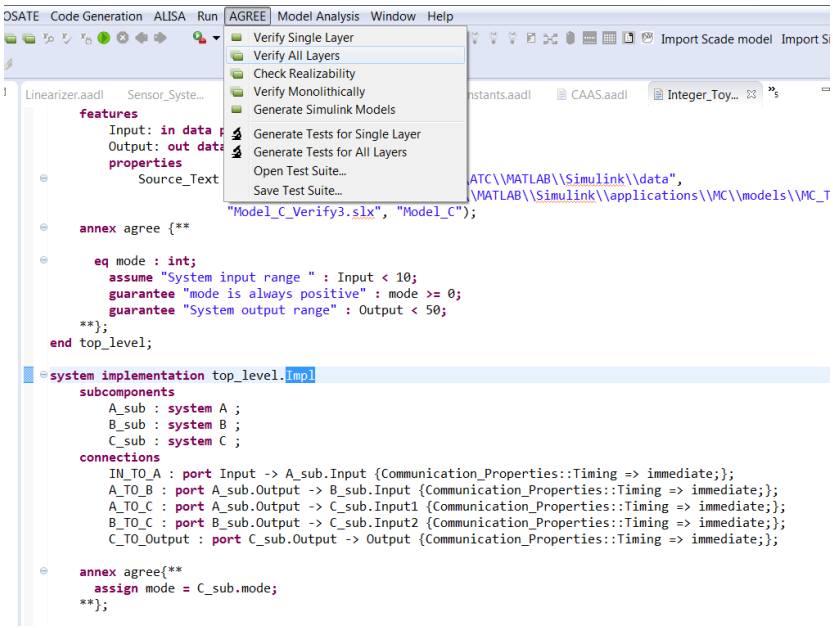


Figure 2.6: Verify All Layers Option from AGREE Menu

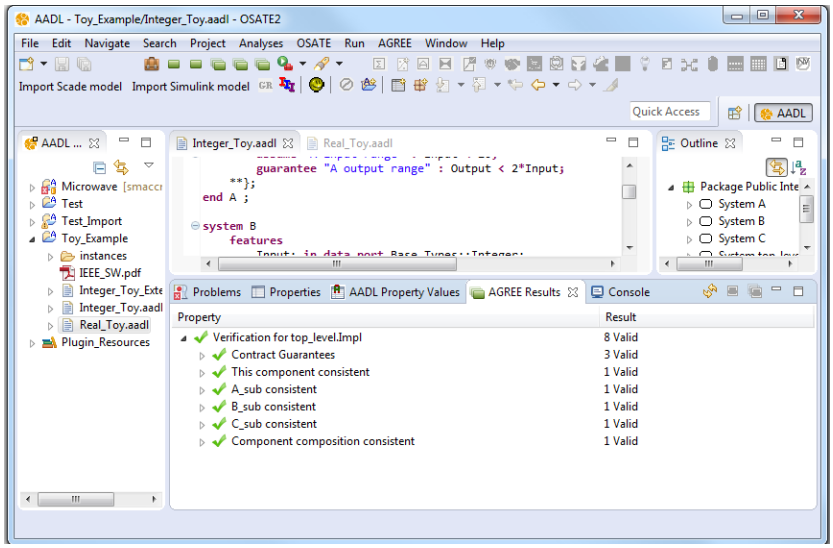


Figure 2.7: Example of AGREE Results

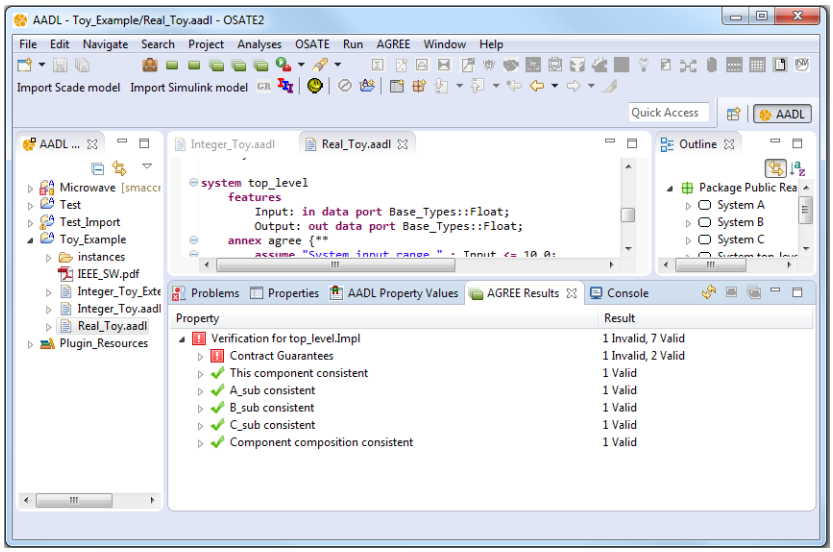


Figure 2.8: Example of Failed Property Result

on your computer. Also, you must associate .xls files in Eclipse with Excel. To do so, the following steps can be taken:

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

   The .xls file type should now be selected.
5. Now choose the "Add..." button next to "Associated Editors"
6. Choose the "External Programs" radio button
7. Select "Microsoft Excel Worksheet" and click OK.

The generated Excel file for the example is shown in Figure 2.10.

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?

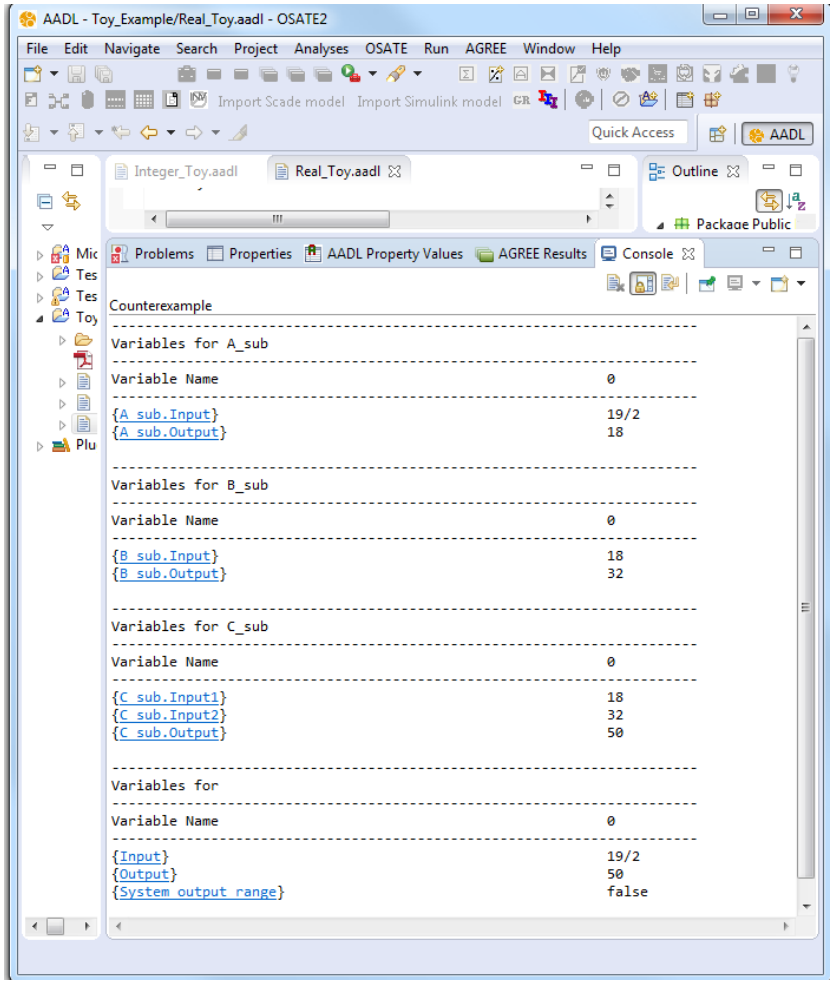
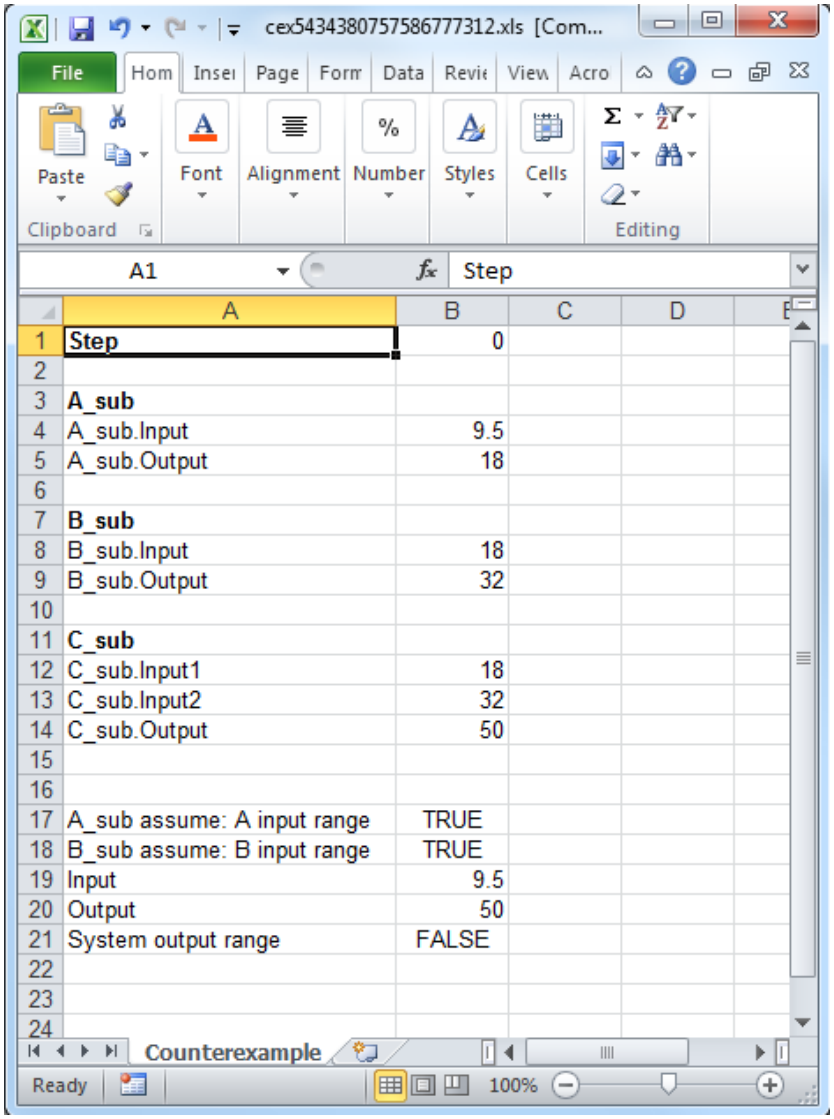


Figure 2.9: Counterexample View in Console



	A	B
1	**Step**	0
2		
3	**A_sub**	
4	A_sub.Input	9.5
5	A_sub.Output	18
6		
7	**B_sub**	
8	B_sub.Input	18
9	B_sub.Output	32
10		
11	**C_sub**	
12	C_sub.Input1	18
13	C_sub.Input2	32
14	C_sub.Output	50
15		
16		
17	A_sub assume: A input range	TRUE
18	B_sub assume: B input range	TRUE
19	Input	9.5
20	Output	50
21	System output range	FALSE
22		
23		
24		

Figure 2.10: Excel Counterexample File

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?

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.

- Section 3.1
- Section 3.2
- Section 3.3
- Section 3.4
- Section 3.5
- Section 3.6
- Section 3.7

3.1 Dataflow 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`:

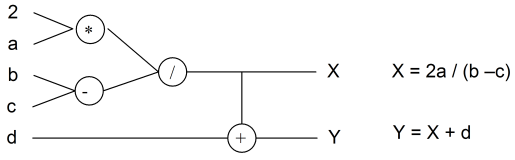


Figure 3.1: A dataflow model and its associated set of equations

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 3.2, 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.

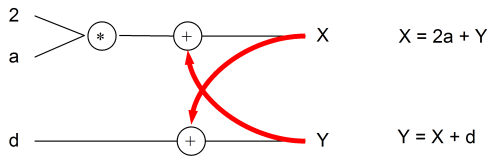


Figure 3.2: A Dataflow Model with Cyclic Dependencies

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 oper-*

ator (`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 {$\mathtt{X}_0 = \mathtt{E}_0$, $\mathtt{X}_1 = \mathtt{E}_1$, ..., $\mathtt{X}_n = \mathtt{E}_n$} that must be acyclic in terms of data dependencies.

3.2 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 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’)). 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 also written in the monospace 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 ‘+’ indicates required repetition (1 or more occurrences). 2) A ‘?’ character indicates that the preceding token is optional.

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. A cursory overview of the AADL declarations is provided in Appendix B.

AADL describes the interface of a component in a *component type*. A *component type* contains a list of *features*, which are the 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 list of connections that occur between its subcomponents and features.

For example, one may decide to create a component type for a car which contains features describing its throttle, speed, and direction. A car component may have many implementations (like a 2006 Toyota Camry, or a McLaren 650s). Different implementations may contain different electronic components, actuators, etc.

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.

3.3 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')

digit ::= (0..9)
```

Some example identifiers include `Count`, `X`, `Get_Symbol`, `Ethelyn`, `Snobol_4`, `X1`, `Page_Count`, and `Store_Next_Item`. **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` and `false`.

Examples of Integer_literals include `1`, `42`, and `-1337`.

Examples of Real_literals include `3.1415`, `1.6180`, and `0.001`.

String elements are defined with the following syntax:

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

string_element ::= "" | non_quotation_mark_graphic_character
```

3.4 Types

The following data types have been built into the AGREE language:

Primitive Types: real, bool, and int.

Composite Types: Record Types (as described in Section 3.7).

In addition, AGREE reasons about AADL features of the following types from the "Base_Types" package included in the AADL Plugin Resources library:

1. Base_Types::Boolean
2. Base_Types::Integer
    a. Base_Types::Integer_8
    b. Base_Types::Integer_16
    c. Base_Types::Integer_32
    d. Base_Types::Integer_64
    e. Base_Types::Unsigned_8
    f. Base_Types::Unsigned_16
    g. Base_Types::Unsigned_32
    h. Base_Types::Unsigned_64
3. Base_Types::Float
    a. Base_Types::Float_32
    b. Base_Types::Float_64

Note: 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. More precisely, Base_Types::Boolean is mapped

to the AGREE primitive type 'bool,' the various sizes of integer and unsigned types in Base_Types are mapped to the AGREE primitive type 'int,' and the floating point types in Base_Types are mapped to the AGREE primitive type 'real.' **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 foreseeable future. If exact floating point behavior (including rounding and truncation) are important to your verification problem, AGREE may provide incorrect answers.

AGREE reasons about AADL Data Implementations like record types. 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 of type Alarm_Outputs.Impl by placing a dot after the variable:

```
Alarm.Is_Audio_Disabled => Alarm.Log_Message_ID > 3;
```

3.5 Subclauses

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 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 AGREE annex is shown below.

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

```
SpecStatement ::= 'assume' STRING ':' Expr | PatternStatement ';'
| 'guarantee' STRING ':' Expr | PatternStatement ';'
| 'assert' (STRING ':')? Expr | PatternStatement ';'
| EqStatement
| PropertyStatement
| ConstStatement
| NodeDefExpr
| RecordDefExpr
| LemmaStatement;

LemmaStatement ::= 'lemma' STRING ':' Expr ';'

EqStatement ::= 'eq' Arg (',' Arg)* '=' Expr ';' ;

PropertyStatement ::= 'property' ID '=' Expr ';' ;

ConstStatement ::= 'const' ID ':' Type '=' Expr ';' ;

NodeDefExpr ::= 'node' ID '(' (Arg (',' Arg)*)? ')' 'returns'
    '(' (Arg (',' Arg)*)? ')' ';'
    NodeBodyExpr ;

RecordDefExpr ::= 'type' ID '=' 'struct' '{'
    (Arg (',' Arg)\*)
    '}' ';' ;

Arg ::= ID ':' Type ;

NodeBodyExpr ::= ('var' (Arg ';')+ )?
    'let' (NodeStmt)+ 'tel' ';' ;

NodeStmt ::= Arg (',' Arg)\* '=' Expr ';'

LinearizationDefExpr ::=
    ‘linearization’ name=ID ‘(‘ (args+=Arg)* ’)’
    ‘over’ ‘[‘intervals+=LinearizationInterval
        (‘,’ intervals+=LinearizationInterval)* ’]’
    (‘within’ precision=Expr)? ’.’ exprBody=Expr ’;’

LinearizationInterval: start=Expr ‘..’ end=Expr;
```

An AGREE subclause consists of a sequence of statements. The different kinds of statements and their uses are described in Section 3.6.

AGREE subclauses can occur either within AADL components or at the top-level of a package. Package-level subclauses are designed to provide reusable libraries of definitions for AGREE. Nodes (as described in Section 3.6.6) and con-

stants in these subclauses can be referenced by component-level subclauses by using the dot notation: <Package_Name>.<definition name>. So, for example, the following equation uses the Counter node defined in the Agree_Common package:

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

3.6 Statements

In this section, we present the various types of AGREE statements and their uses. **Note: Assume Statements (Section 3.6.1) and Guarantee Statements (Section 3.6.2) exclusively live in *component type*s; while Assert Statements (Section 3.6.9) and Lemma Statements (Section 3.6.10) exclusively live in *component implementation*s.**

3.6.1 Assume Statements

Assume statements specify constraints about a component that are assumed to be true. An example of an assume statement is:

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

The string "System input domain" is used to identify the assumption when performing verification. The expression `Input < 10` expresses the condition that is assumed to hold. When verifying a component implementation, the component's assumptions are assumed to be true. However, the assumptions of the component implementation's subcomponents must be proved to hold based on the assumptions of the component and the guarantees of other subcomponents.

3.6.2 Guarantee Statements

Guarantee statements specify constraints that the component maintains as long as the assumptions have always held. An example of a guarantee statement 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` expresses the condition that is guaranteed to hold.

When verifying a component implementation, guarantee statements are proven by the component assumptions and the guarantees present in subcomponent contracts.

3.6.3 Equation Statements

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

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

In this example, we create a variable that counts up from zero. 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, but constraining them with assumption, assertion, or guarantee statements. 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.

3.6.4 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**).

3.6.5 Constant Statements

Constant statements allow definition of named constants. An example constant statement is:

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

Identifiers defined by constant statements are used just like equation variables.

3.6.6 Node Definitions

Node statements are used to define stateful ''functions" that might be used frequently in a component type or implementation. Nodes can have multiple return values. If this is the cause, they must be referenced by an equation statement that has multiple arguments. Nodes can also be defined in an AADL Package. If so, they can be referenced in any expression anywhere in the model.

This way one can make a library of certain types of nodes that are useful for different tasks.

Node definitions in AGREE allow specification of *stateful* definitions; that is, definitions that can maintain internal state. 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 one-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.

Note: Nodes cannot be recursive or mutually recursive.

3.6.7 Record Definitions

Record definitions are used to define record types. Like a struct type in C programming language, a record is a collection of fields, each of its own data type. An example record definition is as follows:

```
type foo = struct {a : bool, b : int};
```

3.6.8 Real-time Patterns

AGREE also supports the specifications of real-time patterns. These patterns were adopted from the Requirements Specification Language adopted under the CESAR project. Patterns can be used instead of expressions in Assume, Guarantee, or Assert statements. The grammar for the patterns that we support is shown below:

```
PatternStatement:
  WheneverStatement
  | WhenStatement
  | RealTimeStatement
  ;

WhenStatement:
  'when' Expr 'holds' 'during' TimeInterval Expr
  ('exclusively')? 'occurs' 'during' TimeInterval
  ;

WheneverStatement:
  'whenever' Expr 'occurs' Expr
    ('exclusively')? ('occur' | 'occurs')) 'during' TimeInterval
  | 'whenever' Expr 'occurs' Expr
    ('exclusively')? ('holds') 'during' TimeInterval
  | 'whenever' Expr 'occurs' Expr 'implies' Expr
    ('exclusively')? 'during' TimeInterval
  ;
```

```
RealTimeStatement:
  'condition' Expr 'occurs' 'each' Expr ('with' 'jitter' jitter=Expr)?
  | 'condition' Expr 'occurs' 'sporadic' 'with' 'IAT' Expr
    ('with' '*jitter*' jitter=Expr)?
  ;

TimeInterval:
  '\[ | (' Expr ',' Expr '\] | )')
  ;
```

The expressions in each of the patterns must be an IDExpr (a variable without any dots) of Boolean type. Details about the semantics of these patterns, how they are implemented, and how they may be used can be found in the Final Documentation and Technical Report for the Requirements Patterns for Formal Contracts in AADL program.

The time intervals specified by these patterns reference values for a reserved variable named "time". This variable is present in any counterexamples or inductive counterexamples that the tool produces. Additionally, AGREE contains three special functions that take a single IDExpr as an argument and produce a value of type real. These functions are:

1. timeof(id) – returns the the last value of the variable time in which id was true. If id has never been true then the function returns the value -1.0.
2. timerise(id) – returns the last value of the variable time in which id transitioned to true. If id has never been true then the function returns the value -1.0.
3. timefall(id) – returns the last value of the variable time in which id transitioned to false. If id has never been false then the function returns the value -1.0.

3.6.9 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.

Assert statements make unchecked statements about how the component behaves. These are also used to reference variables from a subcomponent in the component contract. For the purpose of analysis assertions are treated just like

the system assumptions. However, AGREE never verifies that the assertions actually hold. That is to say, the assertions of a subcomponent are never proven to hold like subcomponent assumptions. Assert statements can refer to equations or features defined in the component type. They are often used to refer to subcomponent variables in contracts higher up in the model hierarchy.

Note: Assert statements are assumed to be true and are not validated in any way by AGREE. Any use of this statement should be exercised with great caution. All assert statements should be examined by a domain expert and formal verification expert.

3.6.10 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. These are used to help the model checker learn facts to improve its ability to prove other properties. However, unlike guarantees, lemmas are not made visible when trying to prove properties at the next level of abstraction. Subcomponent lemmas are not used to prove other subcomponent guarantees or system guarantees.

3.6.11 Advanced Topic: Linearization Definitions

The linearization definition provides the declaration of a linear approximation of a non-linear expression over segment(s) of its input domain, resulting in a new expression that bounds the non-linear expression with piecewise linear segment(s). The non-linear expression supports a small core of mathematical functions found on a calculator (i.e., '+', '-', '*', '/', '^') as seen in Figure 14. **Note**: The non-linear expression can contain references to only the input variable (no other AGREE identifiers). Figure 15 demonstrates a linear approximation bounding the output values within the upper and lower bounds.

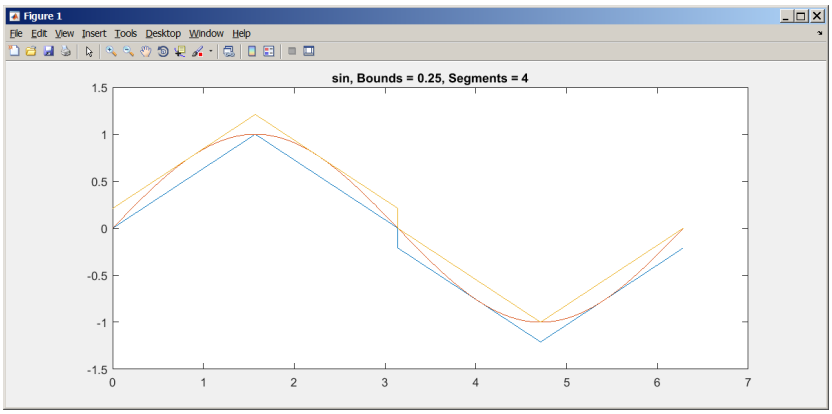


Figure 3.3: Bound Non-linear Expression with Piecewise Linear Segments

The following linearization definition example provides the linear approximation for the square operation with a real type input over input interval between -1.0 and 1.0, and input precision of 0.01.

```
linearization sq (y : real) over [-1.0 .. 1.0] within 0.01 : y^2.0;
```

Note: Currently linearization definition with single input of real type is supported.

The following function calls are allowed within the linearization body expression:

> abs, acos, asin, atan, cbrt, cos, cosh, exp, expml, log, log10, log1p, signum, sin, sinh, sqrt, tan, and tanh.

3.7 Expressions

A simplified description of the set of expressions for AGREE is presented below.

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

QID::= ID '::' ID ;

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

Literal :: = Boolean_literal | Integer_literal | Real_literal ;

Expr ::= Literal
    | ID
    | QID
```

```
    | NestedDotId
    | ID '(' Expr_List ')'
    | 'pre' '(' Expr ')'
    | 'prev' '(' Expr ',' Expr ')'
    | 'event' '(' NestedDotID ')'
    | 'floor' '(' Expr ')'
    | 'real' '(' Expr ')'
    | 'Get_Property' '(' Expr ',' AADL_Property ')'
    | 'this' ('.' NestedDotId)?
    | '(' Expr ')'
    | RecordUpdateExpr
    | ('-' | 'not') Expr
    | Expr ('+' | '-' | '\*' | '/' | 'div'| 'mod') Expr
    | Expr RelateOp Expr
    | Expr ('and' | 'or' ) Expr
    | Expr ('->' | '=>' | '<=>' ) Expr
    | 'if' Expr 'then' Expr 'else' Expr ;

Expr_List ::= Expr ',' Expr_List | Expr ;
```

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

->
=>
<=>
or
and
< | <= | > | >= | = | <> |!=
+ | -
* | / | div | mod
unary minus | not
if then else
prev | next | Get_Property
ID | QID | NestedDotID | Literal | pre | this | ()

Therefore, `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, users can write:

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

Expressions reason about the current state or past states of variables and can reference variables defined in equation statements or other identifiers in the AADL model. In the rest of this section, we describe different types of expressions in

the order they are listed as alternatives to the grammar rule for Expr.

3.7.1 ID Expressions

ID expressions are used to reference 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.

3.7.2 NestedDotID (Field) Expressions

A NestedDotID expression can have dots in between ID expressions, e.g., `food.bar.biz`. It can be used to refer to record types or variables of a subcomponent. For example, one could use the NestedDotID expression foo.bar to reference the input, output, or equation variable `bar` of subcomponent `foo` within the implementation of some AADL component. A NestedDotID expression can also be 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.

3.7.3 Node Call Expressions

A node call expression is an ID of a defined node followed by parenthesis. If the node is defined in an AADL Package, then the ID should be the AADL Package name followed by a dot (.) and then the node name.

3.7.4 Linearization Call Expressions

A Linearization Call Expression is an ID of a defined linearization expression followed by parenthesis. If the linearization expression is defined in an AADL Package, then the ID should be the AADL Package name followed by a dot (.) and then the ID for the expression. For the example linearization expression provided in Section 3.6.11, its ID "sq" can be used in other expressions like a non-recursive, pure function call. The following lemma statement provides one such example.

```
eq y : real;

lemma "sq() range positive" : sq(y) <= -0.10;
```

3.7.5 Stream (Previous Value and Arrow) Expressions

Arrow Expression. The arrow expression evaluates to the value of the expression of the left hand side of the arrow on the initial step. Otherwise it evaluates to the value of the expression on the right hand side of the arrow. The arrow expression is used with the pre expression to reason about past values of variables. For example, we can define a variable in an AGREE contract that starts at zero and increments by one each step in time using an equation statement:

```
eq count: int = 0 -> pre(count) + 1;
```

Previous Value Expression. A previous value expression evaluates to the value of its argument on the previous time frame. It should that it be guarded by an arrow expressions as its value is undefined on the initial step.

The previous value expression defines an initialized stream. So, if we write:

```
eq x : int = 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:

```
eq x : bool = (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".

Note: 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
```

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.

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.

3.7.6 Event Expressions

An event expression is a special predicate that is used to reason about AADL event data ports. For an input event data port, its semantics are such that it evaluates to true if a value is *present* on the event port and false otherwise. For an output event data port, its semantics are such that it evaluates true if data is being sent on the port and false otherwise.

3.7.7 Floor and Real Expressions

A floor expression takes an expression of type *real* as an argument and returns an *int* equal to the floor of the number.

A real expression takes an expression of type *int* as argument and returns a *real* equal to its value.

3.7.8 Get Property Expressions

A get property expression allows a user to reason about values of AADL properties in the model. The first argument is the relative path to an AADL component in the instance model or 'this' if the property exists in the component in which get property statement lives. The second argument is the name of the AADL property.

3.7.9 Unary Minus and Not Expressions

An Unary Minus expression is used to negate integer or real valued expressions.

A Not Expression is used to negate boolean valued expressions

3.7.10 Record Update Expressions

Record Update Expression are assignments to all or a specific field of a record type variable. For example, `foo {a := true; b := 1}` and `bar {a := true}` are valid Record Update Expressions, given definition for type `foo` and variable `bar` as follows:

```
type foo = struct {a : bool, b : int};

eq bar : foo;
```

The record update expression expects an expression of record type on the left hand side of the curly braces. It returns the same record as the left hand side expression except with its member IDs set to the value of the expression on the right hand side of the `:=`;

3.7.11 Arithmetic Operations

Arithmetic operations must be performed on expressions of the same type. They follow the standard order of precedence. Note that AGREE will give a warning if you write an expression that is not linear. Some theorem provers do not reason about non-linear expressions. Non-linear integer arithmetic is undecidable and most theorem provers do not use a decidable decision procedure for non-linear real arithmetic. So it is recommended that you only use linear expressions.

3.7.12 Relation Expressions

Relation expressions can be performed on integers or reals, but not a combination of both. Equality can be used on Booleans as well.

3.7.13 Boolean Expressions

Boolean expressions have the standard associative properties and order of precedence.

Chapter 4

AGREE/OSATE Tool Suite

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

- Section 4.1
- Section 4.2
- Section 4.3

4.1 Tool Suite Overview

Figure 4.1 shows an overview of the AGREE/OSATE tool suite. As presented in the figure, OSATE is installed as an Eclipse plugin that serves as the IDE for creating AADL models. AGREE runs as a plugin inside OSATE that provides both a language (AADL annex to annotate the models with assume-guarantee behavioral contracts) and a tool (for compositional verification of the contracts reside in AADL models). AGREE translates an AADL model and its contract annotations into Lustre and then queries the JKind model checker to perform the verification. JKind invokes a backend Satisfiability Modulo Theories (SMT) solver (e.g., Yices or Z3) to validate if the guarantees are valid in the compositional setting.

4.2 Installation

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

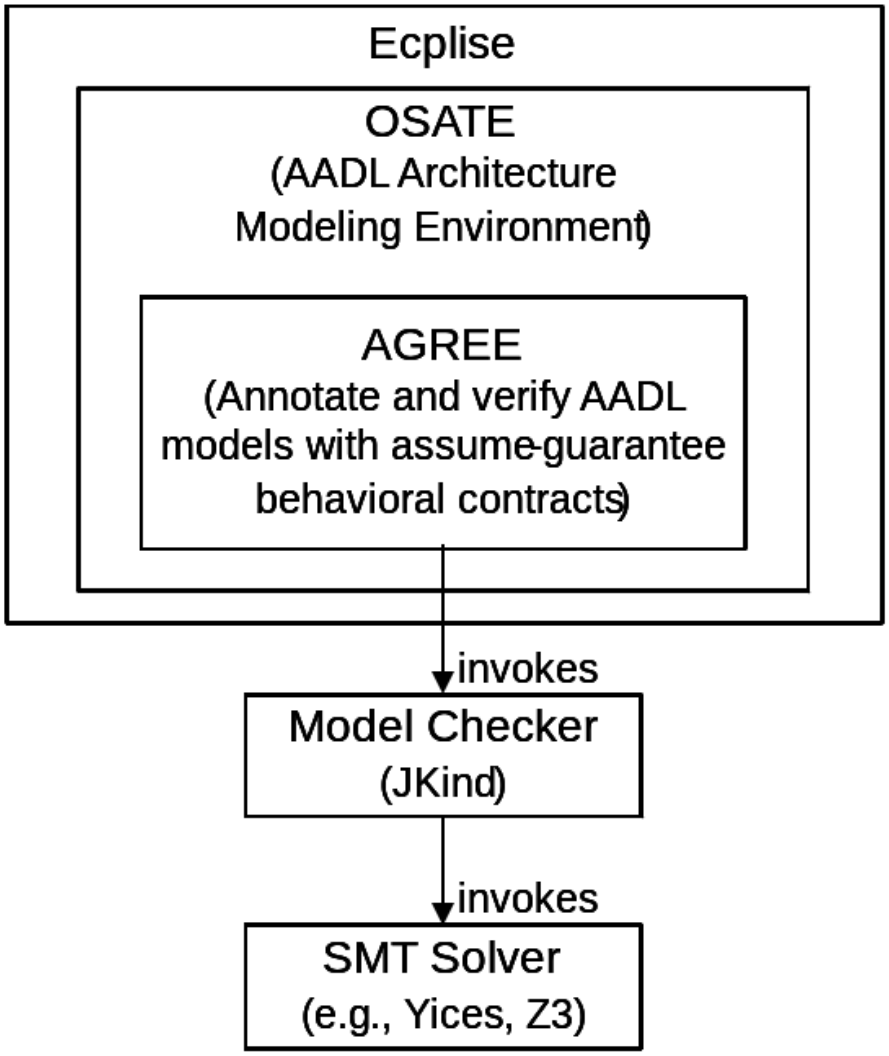


Figure 4.1: Overview of AGREE/OSATE Tool Suite

4.2.1 Install OSATE

Binary releases of the OSATE tool suite for different platforms are available at "OSATE Releases" (http://osate.org/osate-releases.html). Choose the most recent version of OSATE that is appropriate for your platform. At the time of writing this document, there are binary tar ball installations for 64-bit Linux and Apple MAC OSX and binary ZIP installations for 32-bit and 64-bit Microsoft 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 splash screen shown in Figure 4.2 should appear, and OSATE should begin loading:

Figure 4.2: OSATE Splash Screen

If OSATE loads successfully, continue to the next step in the installation process. If not, and you are running Windows, the most likely culprit involves mismatches between the 32-bit and 64-bit version of OSATE and the bit-level of the Windows OS. Please check to see whether the version of OSATE matches the bit-level of your version of Windows OS. If running Windows 7, this information can be found in the System Control Panel as shown below in Figure 4.3. Note that this information is also required for downloading the correct version of the SMT Solver in the next installation step.

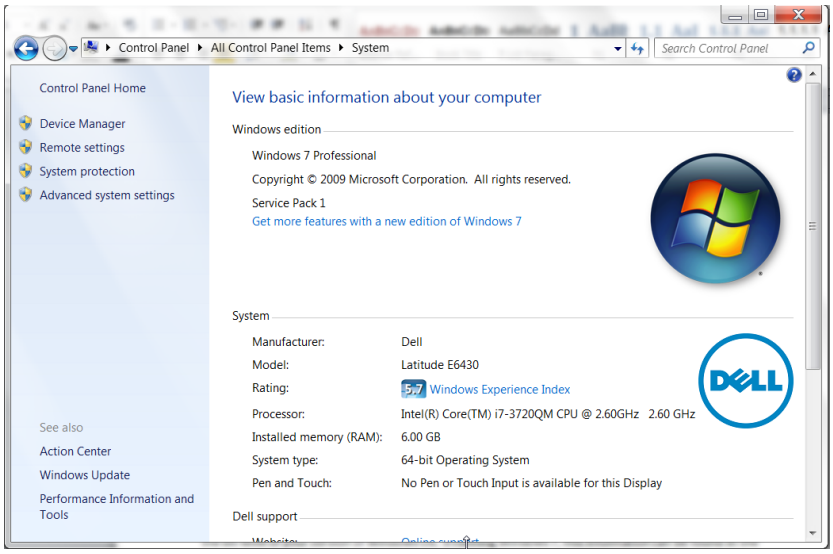


Figure 4.3: Windows OS Version and Bit size information

4.2.2 Install the SMT Solver

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

To download Yices, navigate to "Yices Install" (http://yices.csl.sri.com/) and download the version of Yices appropriate for your platform.

To download Z3, navigate to "Z3 Releases" (https://github.com/Z3Prover/z3/releases) and download the version of Z3 appropriate for your platform.

Either tool must be unzipped and placed in a directory somewhere in the file system. Then this directory must be added to the system path. For directions on how to add directories to your path, please see "How to permanently set $PATH on Linux/Unix?" (http://stackoverflow.com/questions/14637979/how-to-permanently-set-path-on-linux), or see "Add to the PATH on Mac OS X 10.8 Mountain Lion" (http://architectryan.com/2012/10/02/add-to-the-path-on-mac-os-x-mountain-lion/#.VszAv_krJph) for Mac OS.

To add directories to your system path in Windows, first navigate to the System Control Panel and choose the "Advanced system settings" button on the left side of the panel. The system properties dialog will appear. Choose the "Advanced" tab in the dialog as shown in Figure 4.4, then click "Environment variables".

The environment variables dialog box is shown in Figure 4.5.

In order to make the application available to all user accounts choose the PATH

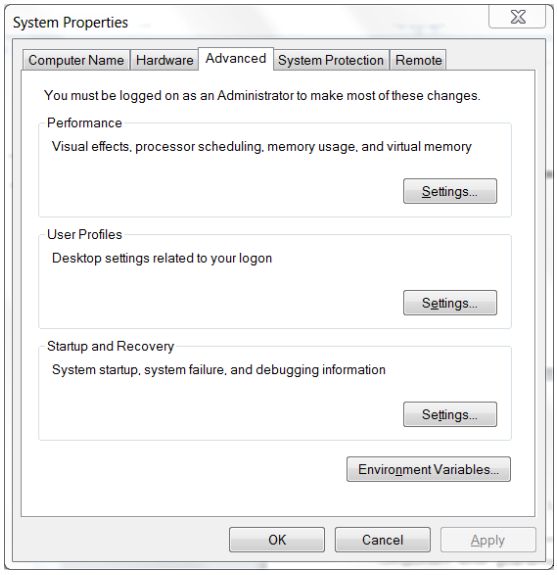


Figure 4.4: System Properties Dialog Box

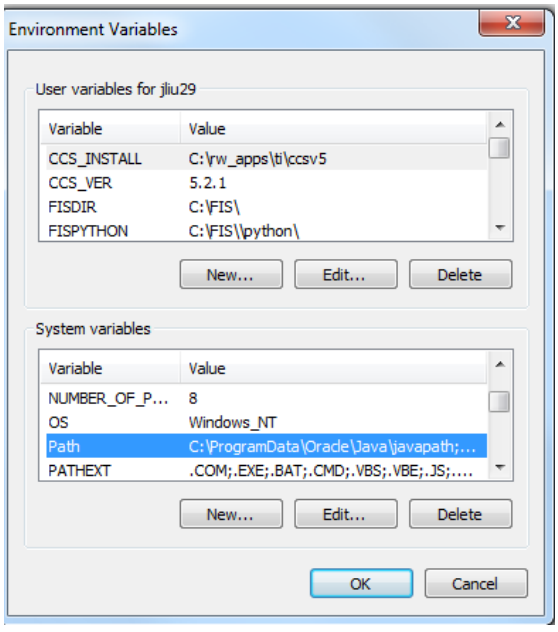


Figure 4.5: Environment Variables Dialog Box

environment variable in the "System variables" section and click "Edit…". This will bring up a text edit box, as seen in Figure 4.6. If the existing path string in the text edit box does not end with a semicolon (';'), add a semicolon first, then append the path to the SMT solver's "bin" directory, and click "OK" on the dialogs. The bin directory for the Yices tool is underneath the main Yices directory, e.g., C:\Apps\ yices-2.4.2-x86_64-pc-mingw32-static-gmp\yices-2.4.2\bin. The bin directory for the Z3 tool is underneath the main z3 directory, e.g., C:\Apps\z3-4.4.1-x64-win\z3-4.4.1-x64-win\bin.

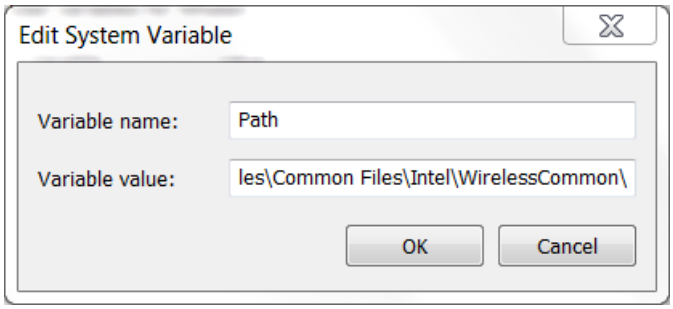


Figure 4.6: System Variable Text Edit Box

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

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

4.2.3 Install the JKind Model Checker

Download the latest release of jKind at: "https://github.com/agacek/jkind/releases" (https://github.com/agacek/jkind/releases) and unzip it into a location in the file system. Place the directory containing jkind.jar 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>
-excel generate results in Excel format
-help print this message
-induct_cex generate inductive counterexamples
-interval generalize counterexamples using interval analysis
-n <arg> maximum depth for bmc and k-induction (default: 200)
-no_bmc disable bounded model checking
```

```
-no_inv_gen disable invariant generation
-no_k_induction disable k-induction
-pdr_max <arg> maximum number of PDR parallel instances (0 to disable PDR)
-read_advice <arg> read advice from specified file
-scratch produce files for debugging purposes
-smooth smooth counterexamples (minimal changes in input values)
-solver <arg> SMT solver (default: yices, alternatives: cvc4, z3,
yices2, mathsat, smtinterpol)
-support find a set of support and reduce invariants used
-timeout <arg> maximum runtime in seconds (default: 100)
-version display version information
-write_advice <arg> write advice to specified file
-xml generate results in XML format
-xml_to_stdout generate results in XML format on standard out

C:\apps >
```

4.2.4 Install AGREE

Download the latest release of AGREE from "SMACCM Releases" (https://github.com/smaccm/smaccm/releas and unzip it into a location in the file system. Unzipping the file should create a directory called "plugins" containing a set of .jar files. Then start OSATE if haven't already, go to Help->Installation Details, and in the "Installed Software" click on AGREE and uninstall the plugin. After uninstall the default AGREE that comes with OSATE, go in to the OSATE/plugins folder and delete the three .jar files that start with "com.rockwellcollins.atc.agree". Then copy all the .jar files from the plugin folder of the AGREE's latest release into OSATE/plugins folder. Figure 4.7 shows a screen capture of the OSATE/plugins folder after copying the .jar files that come with AGREE release v2.2.0.0.

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 4.8.

Note: If Yices version 2.X.X (e.g., 2.4.2) is used, select "Yices 2" in OSATE "Window" menu -> "Preferences" -> "Agree" -> "Analysis" -> SMT Solver, as shown in Figure 4.9. Users can also adjust the timeout and maximum depth for k-induction to use in the "Analysis" configuration dialog.

4.3 Main Features

In this section we walk through the main features involved with the AGREE tool suite.

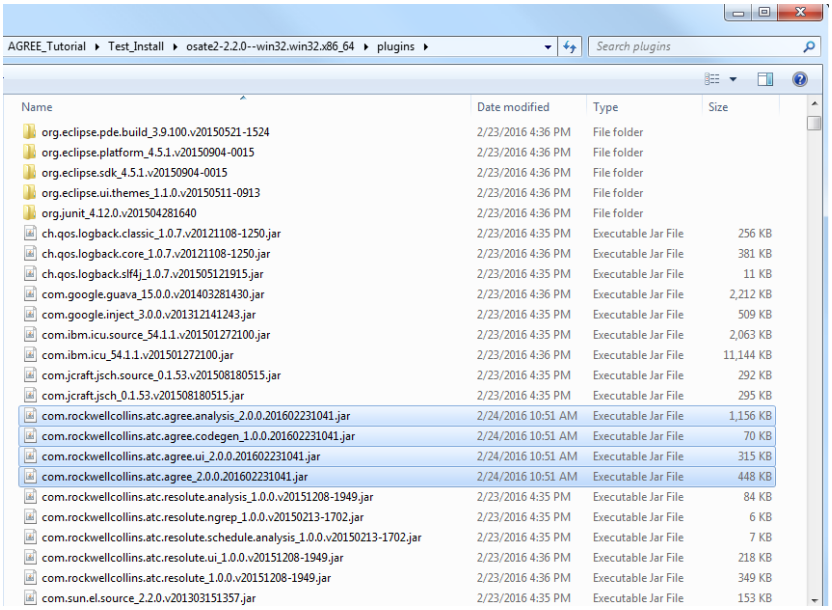


Figure 4.7: OSATE/plugins Directory with .jar Files Replaced

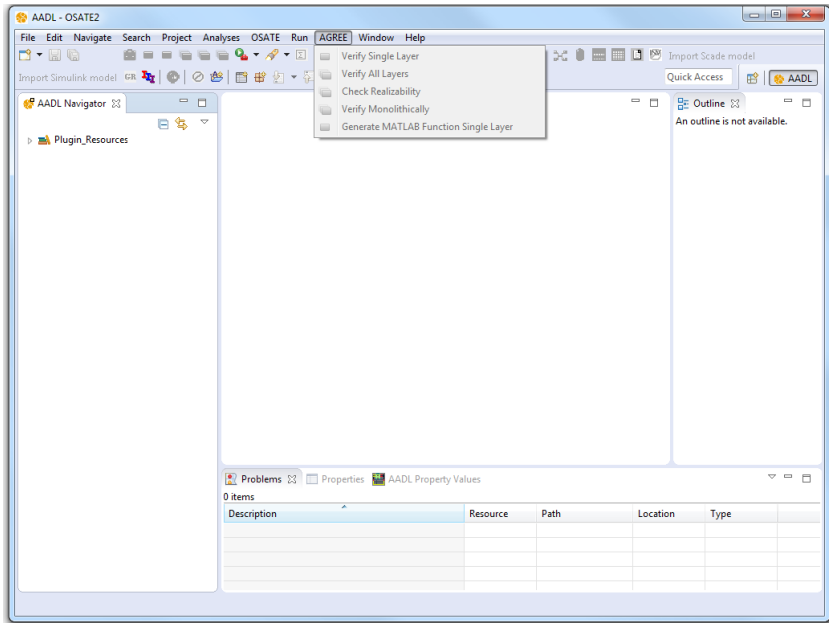


Figure 4.8: AGREE Install Test

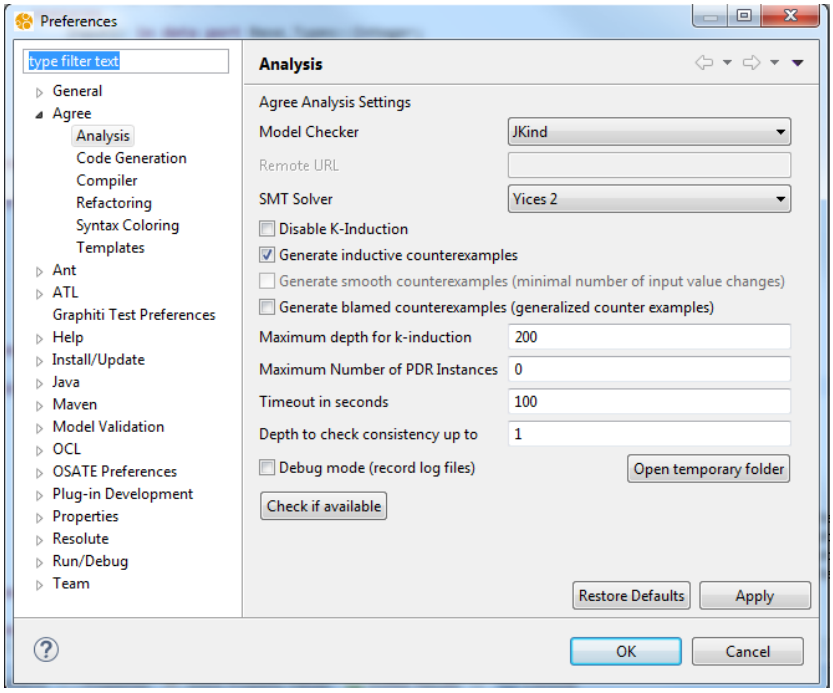


Figure 4.9: SMT Solver Selection

4.3.1 Import Existing Projects

To import an existing project into AGREE, users need to first select "Import" through the File menu as shown in Figure 4.10. This will open an Import dialog box, as shown in Figure 4.11. From there, users can choose to import from different sources. This section presents a few most commonly used sources. After the import project selection is done, you should see the project in the AADL Navigator in the left-hand-side pane in OSATE.

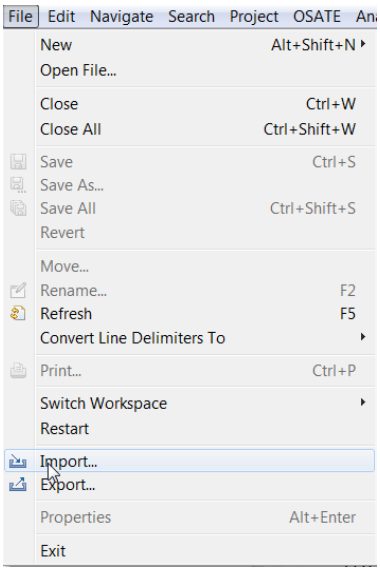


Figure 4.10: Import Project from the File Menu

Import Archived Projects. To import an archived project (previously exported project in the form of a zip or tar file format), choose "Existing Projects into Workspace" under the "General" category from the Import Dialog box in Figure 4.11 and click "Next". Choose "Select archive file", navigate to the location and select the archived project file, and click "Finish". An example is shown in Figure 4.12.

Import Projects from a Directory. To import a project from a directory, choose "Existing Projects into Workspace" under the "General" category from the Import Dialog box in Figure 4.11 and click "Next". Choose "Select root directory", navigate to the location and select the project folder, and click "Finish". An example is shown in Figure 4.13.

Import Git Projects. To import a project from Git, choose "Projects from Git" under the "Git" category from the Import Dialog box in Figure 4.11 and click "Next". Choose the "Existing local repository", then select a local repository that contains the projects, and explore to the specific project folder to

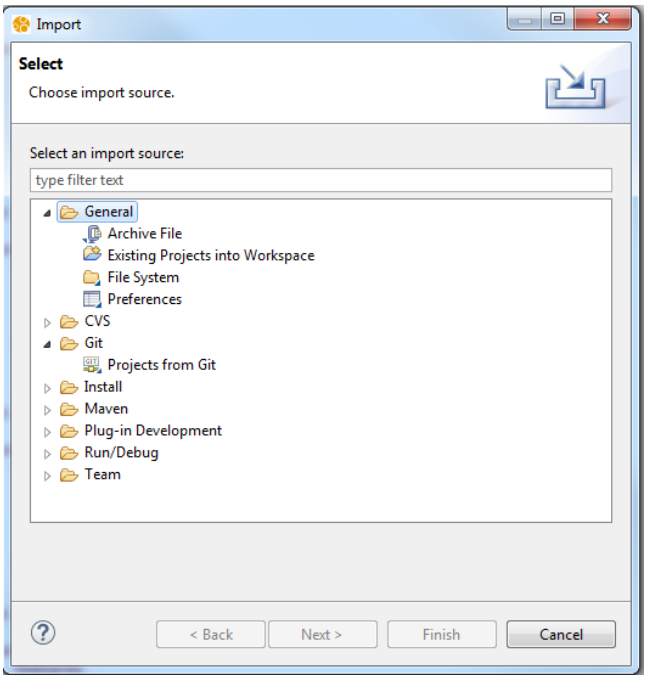


Figure 4.11: Import Dialog Box

import.

Note: Example projects with AGREE contracts in the AADL models can be obtained from the "SMACCM Git repository" (https://github.com/smaccm/smaccm/tree/master/models). These files are from GitHub and could be retrieved via the Git configuration control tool. More information about the Git tool and the download information can be found at "Git-SCM" (https://git-scm.com/).

4.3.2 Create New Projects

After started OSATE, the AADL perspective should be the default. If not, the AADL perspective can be selected via selecting the "Window" menu -> "Perspective" -> "Open Perspective" -> "Other..." -> "AADL". (See the "AADL" text on the upper right corner of the OSATE window as shown in Figure 4.8.)

Create a new AADL project by selecting the "File" menu -> "New" -> "Project...", and select "AADL Project" under the "AADL" category. An example is shown in Figure 4.14. More detailed information about creating AADL models in a project can be found at "Editing a First AADL Model" (https://wiki.sei.cmu.edu/aadl/index.php/Editing_a_first_AADL_model).

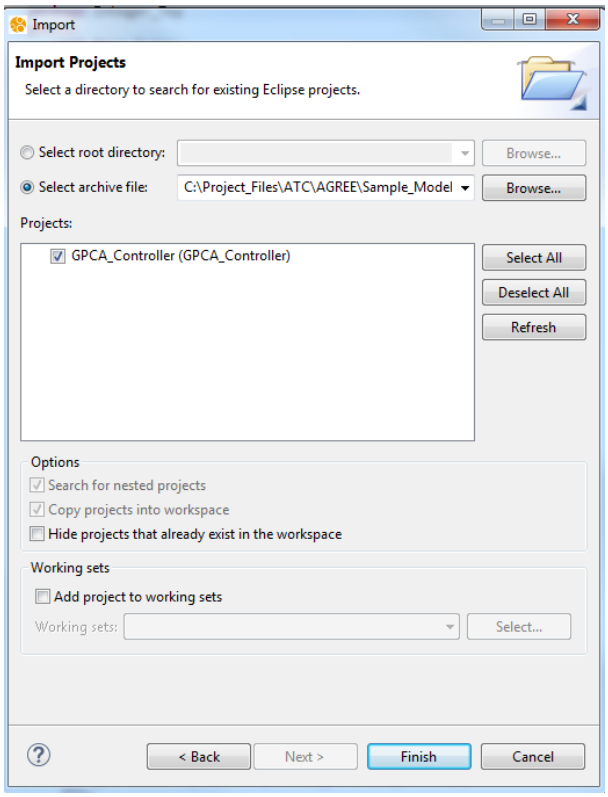


Figure 4.12: Import Archived Projects

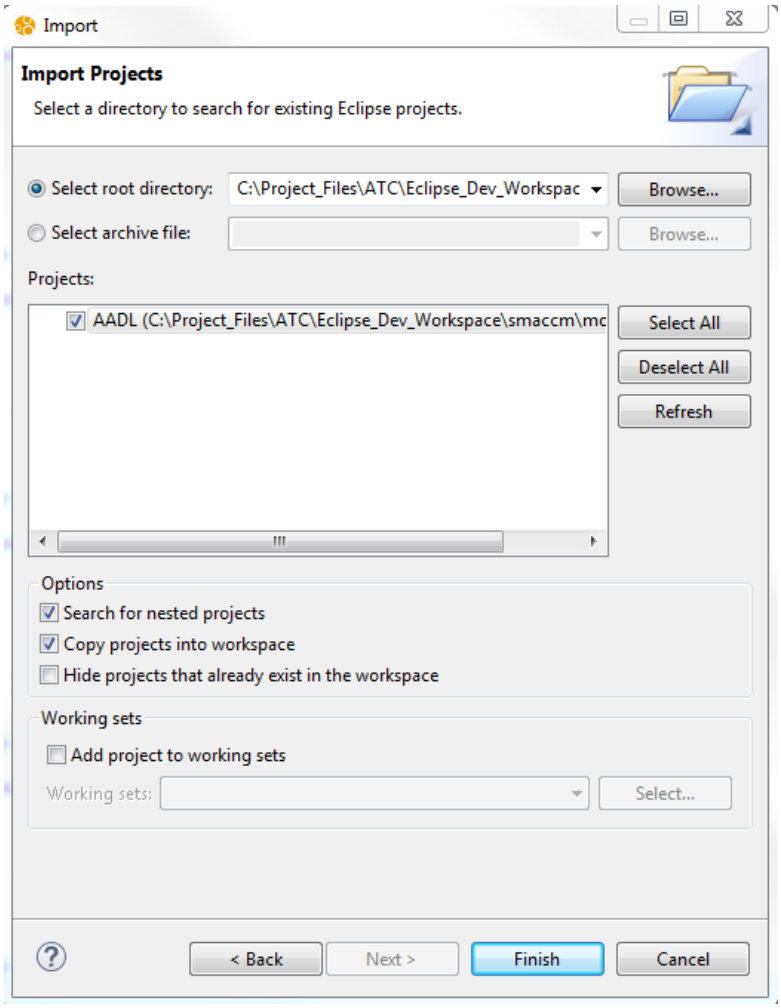


Figure 4.13: Import Projects from a Directory

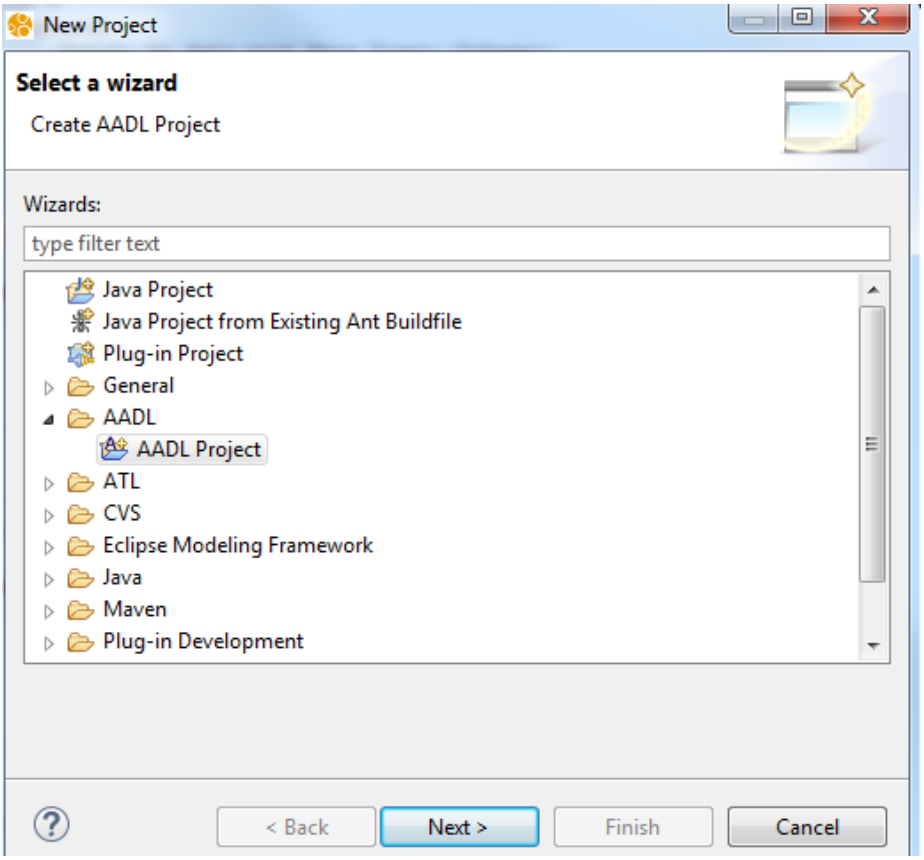


Figure 4.14: Create a New AADL Project

4.3.3 Verify Contracts

As described in the previous chapters, each of the system and component contracts in AGREE are formalized as Assumptions and Guarantees. A component's contracts contain a set of Assumptions about the component's inputs and a set of Guarantees about the component's outputs. The Assumptions and Guarantees may also contain predicates that reason about how the state of a component evolves over time.

The goal of compositional verification is to prove that each component's contract is satisfied by the interaction of its direct subcomponents as described by their respective contracts. Users can start the verification by selecting a system implementation of a component in the outline pane on the right side of OSATE, and select either the "Verify Single Layer" or the "Verify All Layers" option from the right-click menu (shown in Figure 2.5) or the AGREE menu (shown in Figure 2.6). As the names suggest, "Verify Single Layer" performs verification at the current layer of the architecture hierarchy, while "Verify All Layers" performs verification at the current layer and each layer below. For a given layer of the architecture, the verification uses the Assumptions and Guarantees of the direct lower level components as supporting evidence, to prove if the Assumptions and Guarantees at the current layer are satisfied.

Another verification option that is available through the right-click menu or AGREE menu when selecting a system implementation is "Verify Monolithically". This option utilizes Assumptions and Guarantees not only from the direct subcomponents at the lower level, but from components all levels below as evidence for the verification. This is needed when the constraints imposed by components lower in the hierarchy are needed in the proof, so users do not need to manually copy the Assumptions and Guarantees from those lower level components up to their parent components in the hierarchy.

To help clarify the difference between the co-verification options, consider the picture in Figure 4.15. In this Figure, the blocks represent AADL system implementations for different components. In compositional verification ("Verify Single Layer" and "Verify All Layers"), AGREE will only use the Assumptions and Guarantees of component 2 and 3 as evidence ($\texttt{A}_2$, $\texttt{G}_2$, $\texttt{A}_3$, and $\texttt{G}_4$) to prove the Assumptions and Guarantees of component 1 ($\texttt{A}_1$ and $\texttt{G}_1$); the tool ignore any of the constraints posed by components 4 through 7. Monolithical verification, on the other hand, will use Assumptions and Guarantees of components 2 through 7 to prove the Assumptions and Guarantees of component 1.

The philosophy of the compositional verification option 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. Compositional verification should be used as the default verification option with AGREE.

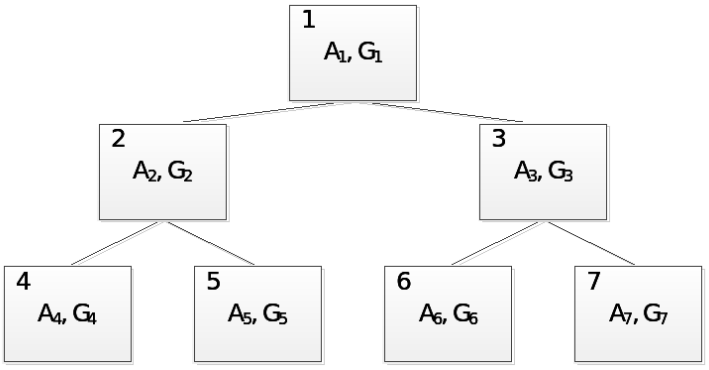


Figure 4.15: A Hierarchical Model

Note 1: Details of lower level components (e.g., implementation details besides the assumptions and guarantees on the interface of the components) are abstracted away during verification of higher level component contracts.

Note 2: Component contracts at the lowest level of the architecture are assumed to be true by AGREE. Verification of these component contracts must be performed outside of the AADL/AGREE environment, as demonstrated in Section 4.3.5.

When the verification starts, a results view appears at the bottom of the screen with the status of all the contracts being checked. Results for each component are grouped by guarantees, assumptions, and consistency. The consistency check verifies if the composition of the subcomponents are consistent, and if the contracts being analyzed for a component is consistent. For example, it checks if the conjunction of a system's guarantees is satisfiable.

If a counterexample for a contract is found then it will have a red icon next to it in the results dialog. Right-clicking on one of these results will bring up a menu where you can choose to view the counterexample in the console, in a spreadsheet, or in a collapsible menu.

Note: Some Guarantees may take longer than the set analysis time to produce a "Valid" or "Invalid" result. In such a case, users may extend the timeout time (e.g., from 100 seconds to 1000 seconds) and/or enlarge the maximum depth for k-induction to use, in OSATE "Window" menu -> "Preferences" -> "Agree" -> "Analysis", as shown in Figure 4.9.

4.3.4 Check Realizability

The need for the realizability checking can be motivated through the following example. In the Microwave AADL model, the following two requirements were

imposed:

Requirement 1: While the microwave is in cooking mode, seconds_to_cook shall decrease.

Requirement 2: If the display is quiescent (no buttons pressed) and the keypad is enabled, the seconds_to_cook shall not change.

On input conditions that satisfy both requirements (i.e., when the microwave is in cooking mode, the display is quiescent and the keypad is enabled), there is a conflict in the value of the seconds_to_cook variable, resulting in the two requirements not being able to be satisfied at the same time in those input conditions.

Realizability checking determines whether or not the component works in all input environments that satisfy the component assumptions. It can be invoked by selecting the system implementation of a component in the outline pane on the right side of OSATE, and select either the "Check Realizability" from the right-click menu (shown in Figure 2.5) or the AGREE menu (shown in Figure 2.6).

4.3.5 AGREE/AADL to Simulink Exporter

Component contracts at the lowest level of the architecture are assumed to be true by AGREE. Verification of these component contracts must be performed outside of the AADL/AGREE environment. The Exporter feature automatically exports the AGREE contracts into properties in a MATLAB function. The MATLAB function can be connected to the component's Simulink model and serves as a synchronous observer for its behavior. Simulink Design Verifier can be invoked to check if the component's Simulink model satisfies the properties exported.

The Exporter feature works with two Simulink models: the implementation model, which contains the subsystem that encapsulates the behavior of the component, and a verification model, which connects the exported AGREE contracts to the implementation.

The implementation model can be auto-generated from the AADL model, if desired, or can be manually created. When auto-generated it contains the inputs and outputs specified in the AADL model and an empty subsystem, ready for the modeler to insert the behavioral logic into. The auto-generated model should be hand edited to contain the implementation of the behavioral aspects of the component, and can be used to generate the code for the target.

The verification model uses the implementation model (as a Simulink model reference block) to create a model suitable for verification. It creates an observer block to wrap the MATLAB function that contains the AGREE contracts as MATLAB properties. It also creates input and output ports according to the

AADL model, instantiates a reference to the implementation model, and connects the observer block to the implementation model. The verification model is not suitable for code generation for the target, but is suitable for verification of the implementation. Since the model being verified is a model reference (and not a copy), formal credit may be taken from the verification activity.

The Exporter does not directly generate Simulink models, but rather generates MATLAB scripts which will generate the models. This allows the exporter to remain largely isolated from issues associated with Simulink versions, as well as providing visibility to the model generation actions should it be necessary for debugging purposes.

The Exporter automatically updates the AADL model with the appropriate path and file name information via the "Source_Text" AADL property. This associates the information of the component implementation in the design model, and allows users to proceed directly to action selection in subsequent exports, if no changes to the saved information.

Note that Real-time patterns (as seen in Section 3.6.8) in AGREE cannot be exported to MATLAB functions, and should not be. These patterns refer to the scheduling and performance of the component, rather than its specific behavior. Thus, the model will need to be decomposed further (i.e., so the real-time constraints will be turned into behavioral constraints on each individual component's inputs and outputs) before generating a Simulink implementation.

To use the Exporter feature, follow the steps below:

1. Specify the data type to be mapped from AGREE to Simulink through the "Window" menu -> "Preferences" -> "Agree" -> "Code Generation". Users may select one of the MATLAB supported integer types (i.e., (u)int8, (u)int16, (u)int32, (u)int64) to represent integers in AGREE and one of the MATLAB supported floating point types (i.e., single, double) to represent reals in AGREE. Note that this option only applies to the abstract "integer" and "real" AADL types; concrete types with a size specification are automatically mapped to the corresponding MATLAB type;

2. Select the system implementation of the component whose contracts are to be exported, and select the "Generate Simulink Models" option in either the right-click menu (shown in Figure 2.5) or the AGREE menu (shown in Figure 2.6).

3. The dialog box shown in Figure 4.16 will be presented. After completing the data fields, select the activity to be performed by clicking on one of the action buttons. The fields and actions are described in detail below.

Output Directory Path: This is the location of the exported MATLAB scripts that, when executed, will actually generate the Simulink models (either the implementation or verification model).

Implementation Model Path: This is the location of the implementation model in Simulink. If the model does not exist, the script can generate an

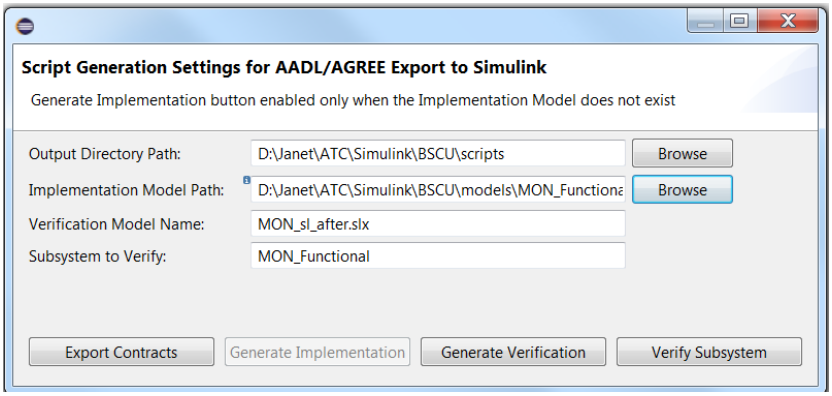


Figure 4.16: General Simulink Models Dialog

empty subsystem with the interface consistent with the AADL model. Note that the implementation model must exist before the verification model can be generated. If the inputs/outputs to the implementation model contain aggregate types or structured data, these IO data will be represented in Simulink by bus objects. These bus objects are auto saved in a file with a filename of the form "<ImplementationModel>_busobjs.m" in this directory.

Verification Model Name: This is the file name for the verification model that will be generated. It will be placed in the output directory specified above.

Subsystem to Verify: This is the name of the subsystem in the verification model that the observer block connects to. The subsystem contains the implementation logic for the chosen component. An auto generated implementation model will contain only the subsystem named after the text entered in this field. A manually created implementation model may contain multiple subsystems, and the subsystem name entered in the field selects the specific subsystem to verify.

Export Contracts – This button will generate the MATLAB script containing the AGREE contracts, translated into MATLAB properties. This script will be wrapped inside an observer block in the generated Verification model. The script will be placed in the Output Directory.

Generate Implementation – This button will generate the script to create an implementation model. When the script is running in MATLAB, an error will be reported if the implementation model already exists. The script will be placed in the Output Directory.

The implementation model is considered non-overwritable (and will not be overwritten with an export). If users wish to regenerate the implementation model, they will first need to delete the existing model (and its bus object storage file, if applicable).

Generate Verification – This button will generate the script containing the AGREE contracts (as in *Export Contracts*) as well as the script to create a verification model. The generated scripts will be placed in the Output Directory.

The verification model is considered overwriteable and can be regenerated at-will. This is because the verification model contains no logic that is not already captured in other locations (the AGREE contracts or the implementation model contents). If the AGREE contracts (but not the inputs/outputs) are changed, users can click this button to regenerate the verification model with the new contracts.

Verify Subsystem – This button will generate the script containing the AGREE contracts (as in Export Contracts), as well as the script to create a verification model (as in Generate Verification) and run Simulink Design Verifier on the generated verification model. Note that Design Verifier can still be run from the Simulink environment as needed – the generated script automates all the steps needed to verify an implementation model against its AGREE contracts, completing the verification workflow.

Chapter 5

Introduction 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.

Chapter 6

AADL 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 appendix, 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: Model-Based Engineering with AADL.

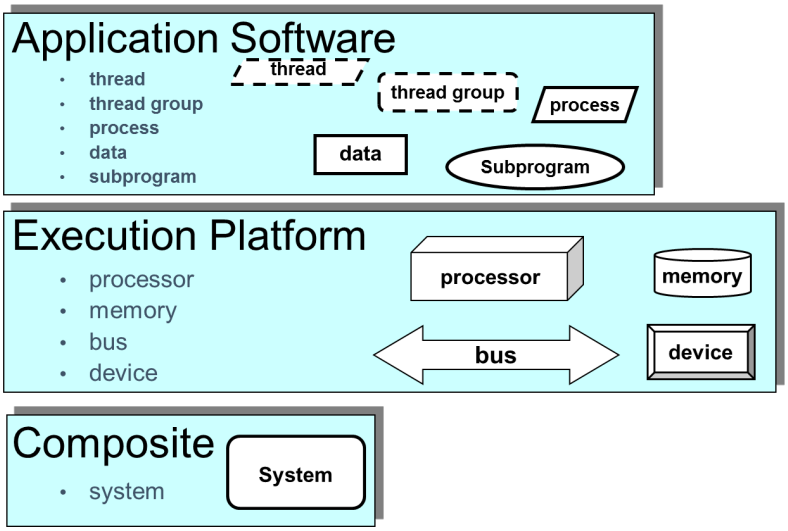


Figure 6.1: Overview of AADL Components

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

AADL can be used to describe both software and the physical platform on which it executes, as shown in Figure 6.1. 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 6.2. 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.*

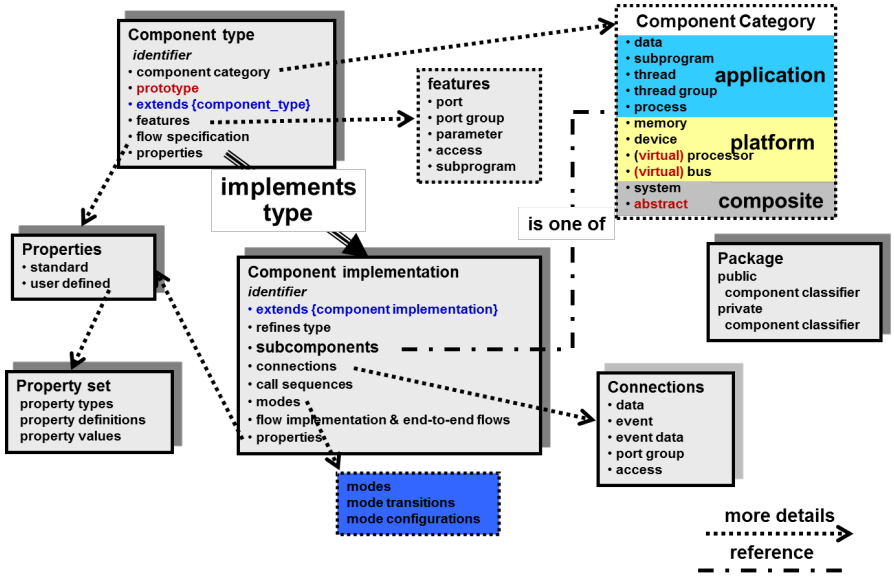


Figure 6.2: Component Types and Implementations in AADL

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

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 1 in in the code segment below. 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,

[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.

event ports, 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;
```

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.

Note 1: By default, AGREE assumes that connections are *immediate.* The best practice is to explicitly state whether each connection is *immediate* or *delayed.*

Note 2: 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.