AGREE tutorial

JULIEN DELANGE
Software Engineering Institute
jdelange@sei.cmu.edu

November 7, 2014

1 Introduction

This document is a tutorial to learn to use the AGREE language and its associated toolset. This is not a user-manual that covers all aspect of the language features, all these aspects are described in the AGREE user-manual[1]. This document is a way to learn how to use the language and its associated tools through several case studies.

1.1 Examples location

All the examples used in this tutorial are available online on the public OSATE github repository[2]. You can import the model into your workspace directly to reproduce the examples presented through this tutorial.

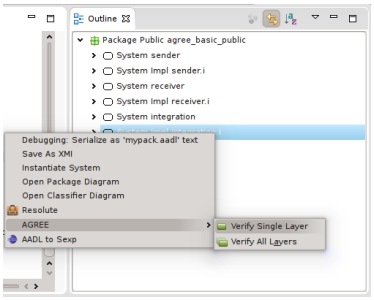

Figure 1: AGREE menu in OSATE outline

1.2 Use Analysis Tools

To use AGREE, model components must be annotated with AGREE annex subclauses. Then, invoking AGREE can be done by selecting the top-level system instance and

[1] https://github.com/smaccm/smaccm/tree/master/documentation/agree
[2] https://github.com/osate/examples/

make a right-click and select two options:

1. **Verify Single Layer**: analyze and verify only one depth of the component hierarchy.
2. **Verify All Layers**: analyze the complete components hierarchy.

1.3 Limitations

When using AGREE, your models must enforce some constraints. There is the list of the constraints your model has to enforce:

- **Execution Order**: the execution order of the model is done in the order of the declaration of the subcomponents.
- **Multiple fanin** are **not** supported. In other words, an incoming feature can have only one incoming connection.
- **Top-level component** must have an AGREE subclause, even if you do not want to verify anything and want to validate the subcomponent. Hopefully, you can insert a dummy subclause like the one shown below.

```
system mysystem
annex agree {**
   guarantee "dummy" : true;
**};
end mysystem;
```

1.4 Understanding Analysis Results

For each component, AGREE provides the following analysis:

1. **Contract Guarantees**
2. **Contract Assumptions**
3. **Contract Consistency**

2 First AGREE model

To understand AGREE basics, we will design a first model with some basics validation contracts. In this example, we will design a system with a sender and a receiver. The sender sends an integer through its outgoing interface.

The integration contract will then guarantee that the value sent through the interfaces is bound (with a range between 0 and 100) and the one received by the receiver has the same bound as well.

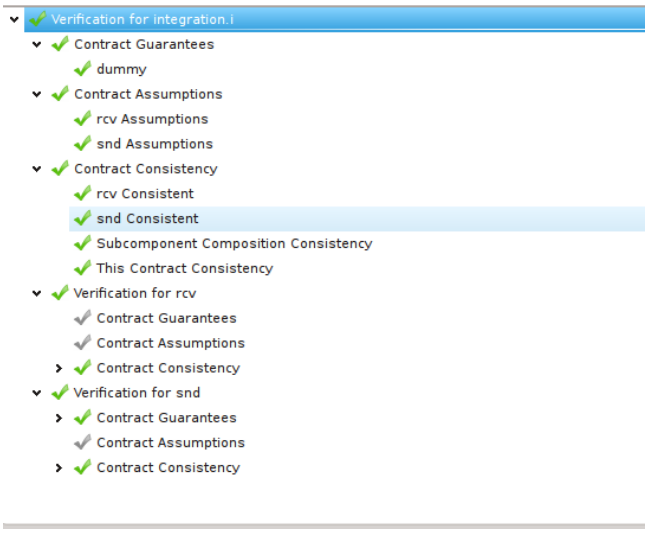

Figure 2: AGREE Results View in OSATE

2.1 Defining guarantees

In the following listing, we show the guarantee contract in the component type (the section defined with the `guarantee` keyword). The component implementation defines the behavior: the integer starts at 1, is incremented at each *tick* and eventually set to 10 when the value reached 90.

```
system sender
features
    dataout : out data port base_types::integer;
annex agree {**
    guarantee "data sent is between 0 and 100": dataout < 100 and dataout > 0;
**};
end sender;

system implementation sender.i
annex agree {**
    eq k : int = 1 -> if (pre(k) > 90) then 10 else pre(k) + 1;
    assert (dataout = k);
**};
end sender.i;
```

2.2 Defining assumptions

We also need to define components assumptions so that the analysis tool can check that they are consistent with the guarantees. In the following listing, we define the assumption for the receiver, specifying that the data received is bound within a range of 0 and 100.

```
system receiver
features
   datain : in data port base_types::integer;
annex agree {**
   assume "data produced between 0 and 100": (datain < 100) and (datain > 0);
```

```
**};
end receiver;
```

2.3 Running the tool

Once both `assume` and `guarantee` are defined, we integrate the components and can start the analysis tool. The results are shown in figure 2.

To show how AGREE can help you to investigate errors within your system, we will introduce an error in the contract. Let's change the `guarantee` of the sender component and specify that the data sent be within a range of 0 to 150. The component type specification should then looks like the following.

```
system sender
features
    dataout : out data port base_types::integer;
annex agree {**
    guarantee "data sent is between 0 and 150": dataout < 150 and dataout > 0;
**};
end sender;
```

When invoking the analysis tool again, it reports that the assumptions of the `rcv` component are not met, as shown in figure 3. When a contract is not validated, AGREE can then provide a counter example that details the different execution paths that lead to the validation error. To get the trace, right click on the assumption/guarantee/consistency contract not met and select the option to show a counter example. Counter examples can be shown in different format: text-based, spreadsheet (with Excel or OpenOffice) or in Eclipse, as shown in figure 4.

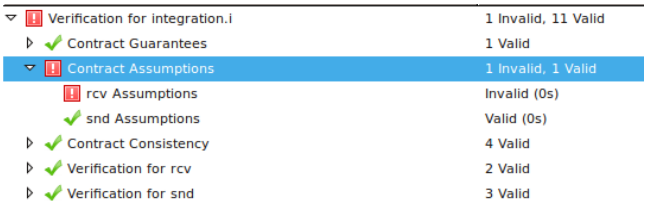

Figure 3: AGREE Results View in OSATE - Contracts not validated

3 The temperature example

The temperature example with explain how to capture the component behavior with SAVI by capturing a redundant example. The example is available under the file `agree-temp-control.aadl` on OSATE github examples repository[3].

The system instance is shown in figure 5. It consists of

- two temperature sensors
- one control panel (operated by the user) to activate heat or cooling

[3] See `https://github.com/osate/examples/tree/master/core-examples/agree`

Name	Step 1
integration_i_Instance	
CLOCK_rcv	true
CLOCK_snd	true
rcv Assumptions	false
rcv	
▽ rcv	
datain	100
snd	
▽ snd	
dataout	100

Figure 4: Counter Example in Eclipse

- one temperature regulation system that activate cooler or heater. It is composed of the following sub-components:
  - One voter that checks temperature consistency and ensure that values from both sensors is similar. Otherwise, it used the previous value
  - One controller that activates the heater or cooler according to the following rules
    - If the user requests more heat and the temperature is below 20, the heater is activated
    - If the user requests to cool down the tempeartur and the temperature is above 0, the cooler is activated

The system has the following constraints

1. Temperature is always between 0 and 20
2. We cannot activate the heater and the cooler at the same time

We add `assume` and `guarantee` statements in the model in order to reflect the system constraints. For example, for the temperature, assume and guarantees are associated in the features that receives or sends the values. The following code illustrates such statements for the `voter` component.

```
annex agree {**
    assume "incoming temperature on sensor1 between 0 and 20 (both included)":
        tempin1 <= 20 and tempin1 >= 0;

    assume "incoming temperature on sensor2 between 0 and 20 (both included)":
        tempin2 <= 20 and tempin2 >= 0;

    guarantee "outgoing temperature is between 0 and 20 (both included)":
```

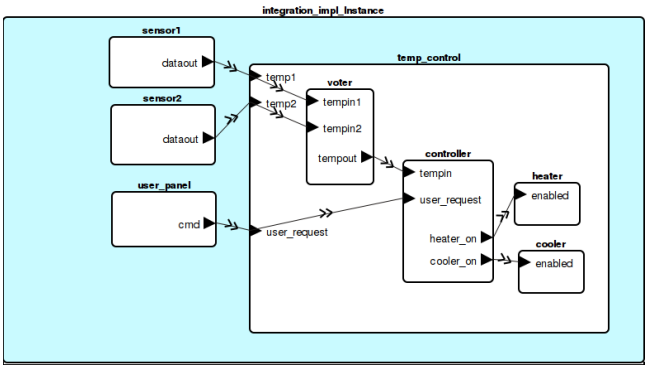

Figure 5: Instance Model of the Temperature System

```
        tempout <= 20 and tempout >= 0;
**};
```

On the other hand, this guarantee must be validated in the component behavior. So, engineers has to add the behavior of the component and make sure that assumptions and guarantees are enforced. In fact, the initial model does not enforce them, as shown in figure 6.

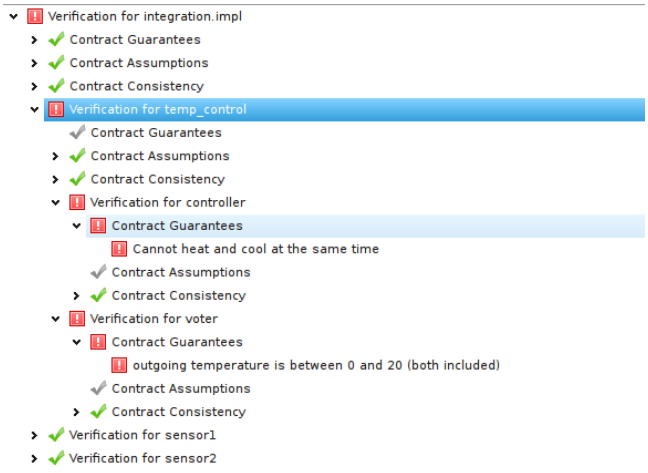

Figure 6: Initial AGREE analysis of the temprature system

When looking at the results, two guarantees are not enforced:

1. On the controller component: Cannot heat and cool at the same time
2. On the voter component: Outgoing temperature between 0 and 20

These guarantees cannot be verified because the component does not enforce them

in their behavior. Next sections will discuss how to specify the behavior in order to have them validated.

3.1 Fixing the controller guarantee

In the controller model, one must ensure that the heater and cooler are not activated at the same time. So, the behavior must specify how the component activates these functions.

The component *guarantees* that it cannot activates the heater and cooler at the same time using the statement below.

```
system controller
features
   tempin          : in data port temperature_type;
   user_request    : in data port command.impl;
   heater_on       : out data port base_types::boolean;
   cooler_on       : out data port base_types::boolean;
annex agree {**
    assume "incoming temperature is between 0 and 20 (both included)": tempin <= 20 and tempin >= 0;

   guarantee "Cannot heat and cool at the same time" : not (heater_on and cooler_on);
**};
end controller;
```

In order to validate that, one need also to define the behavior, *when* the `heater_on` and `cooler_on` features are activated.

When looking at the component implementation, the functions are always activated.

```
system implementation controller.impl
annex agree {**
   eq heating_request : bool = true;
   eq cooling_request : bool = true;
   assert (heater_on = (heating_request and (tempin <= 20)));
   assert (cooler_on = (cooling_request and (tempin > 0)));
**};
end controller.impl;
```

So, one needs to complete the component behavior. We define the activation rules for each features so that:

- The `heater_on` is true if the user issue a request to activate the heat **and** does not activate the cooling **and** the temperature is lower than 20
- The `cooler_on` is true if the user issue a request to activate the cool **and** does not activate the heating **and** the temperature is higher than 0

Using these rules, the controller cannot issue a heating and cooling request at the same time. The following code shows how we adress the issue.

```
system implementation controller.impl
annex agree {**
   eq heating_request : bool = false -> if ((user_request.heater_on = true) and (user_request.cooler_on = false)) then true els
   eq cooling_request  : bool = false -> if ((user_request.cooler_on = true) and (user_request.heater_on = false)) then true el

   assert (heater_on = (heating_request and (tempin <= 20)));
   assert (cooler_on = (cooling_request and (tempin > 0)));
**};
end controller.impl;
```

3.2 Fixing the voter guarantee

The voter component takes the values from both sensor, check that both values are the same and output either the value (if both sensors report the same value) or use the previous correct value.

In the system, the temperature must always be between 0 and 20. So, the voter defines the assumptions that the incoming temperature values are within 0 and 20 but also that the produced temperature is within this range. The following component declaration specifies these contracts.

```
system voter
features
   tempin1      : in data port temperature_type;
   tempin2      : in data port temperature_type;
   tempout      : out data port temperature_type;
annex agree {**
    assume "incoming temperature on sensor1 between 0 and 20 (both included)": tempin1 <= 20 and tempin1 >= 0;
    assume "incoming temperature on sensor2 between 0 and 20 (both included)": tempin2 <= 20 and tempin2 >= 0;
    guarantee "outgoing temperature is between 0 and 20 (both included)": tempout <= 20 and tempout >= 0;
**};
end voter;
```

As for the controller, these contracts must be implemented by the component behavior. The initial component implementation defines the behavior shown below, producing a temperature value outside the expected range.

```
system implementation voter.impl
annex agree {**
   eq selected_temp : int = -1;
    assert (tempout = selected_temp);
**};
end voter.impl;
```

In order to validate the component, one needs to specifies its behavior. To do so, we specify how the incoming values are processed using the following behavior (see below):

- The initial value (left part of the -> symbol) is 0
- If values from both sensor are similar, it is then sent by the voter
- If values from both sensor are different, the voter sends the previous valid value

```
system implementation voter.impl
annex agree {**
   eq selected_temp : int = 0 -> if (tempin1 = tempin2) then tempin1 else pre(selected_temp);

    assert (tempout = selected_temp);
**};
end voter.impl;
```

One the behavior is implemented, the system is correctly validated by the AGREE analysis tool.

4 Conclusion

This tutorial outlines the basics of AGREE to check components integration within an architecture model. We show how to specify components assumptions, guarantees and also define their behavior. Users might want to learn the internals of the language and

get more details about advanced features. This information can be found in the AGREE user manual available online.