RESOLUTE tutorial

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

November 6, 2014

1 Introduction

This tutorial gives a tour of the functionality of RESOLUTE. It does not provide a complete description of the language, which is rather provided by the user-manual of RESOLUTE[1]. This tutorial assumes that you have a working installation of RESOLUTE and that you are able to use the validation tool.

This tutorial uses several examples hosted on OSATE github repository. You can get the examples on the OSATE example repository (see `https://github.com/osate/examples/`) under the directory `core-examples/resolute`.

2 Verifying property definition

This first example checks the definition of a property on a component. The related file is `property_verification.aadl`. The instance model is shown in figure 1.

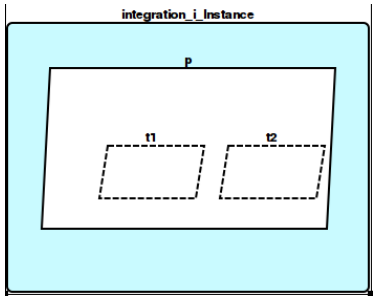

Figure 1: System instance in the `property_verification.aadl`

In this model, each `thread` has the property `resoluteps::foobar` defined:

- **t1** has a value of 10

[1]Documentation avaialble on `https://github.com/juli1/smaccm-improvements/tree/sei-improvements/documentation/resolute`

- **t1** has a value of 20

We define a theorem (`have_good_foobar`, see file `theorems.aadl` in the textual model) to check that all threads have a value for the `resoluteps::foobar` greater than 15.

The theorem works as follow:

1. It retrieves all the `thread` components associated to the process
2. It calls the theorem `check_thread` on each component.
3. The `check_thread` checks that the value of the `resoluteps::foobar` is greater than 15.

```
have_good_foobar(p : component) <=
   ** " Check threads in component " p **
    forall(t : thread). contained(t, p) => check_thread (t)

check_thread(t : thread) <=
   ** "The thread " t " as a foobar bigger than 15"  **
      (has_property (t, resoluteps::foobar)) and
      (property (t, resoluteps::foobar) > 15)
```

When running resolute on the initial model, the model is not validated because **t1** has a value of 10. To be able to validate the model, change the definition of t1 and associate a value greater than 15. For example, changing the definition of the task as follow will be sufficient to validate the model.

```
t1 : thread t.i {resoluteps::foobar => 20;};
```

After changing the model, run the analysis again, the model would be validated.
In this part, we introduce the following RESOLUTE concepts:

1. Call to another RESOLUTE theorem
2. Use the `forall` keyword
3. Use the `has_property` built-in function to check that a property is defined on a component.
4. Use the `property` built-in function to get the value of a property.

3 Analyzing Connections

Now that we have discussed the verification of property values in the components, we will present how you can analyze connections. In this example, we will check that each incoming port has only one incoming connection. This is a validation oen might want to validate in the model in order to ensure that there is only one sender for each communication port. This type of modeling restriction is required by some tools such as AGREE[2].

[2]AGREE actually does not support multiple fan-in, such a validation tool can then help designers to check compliance of their model against AGREE constraints

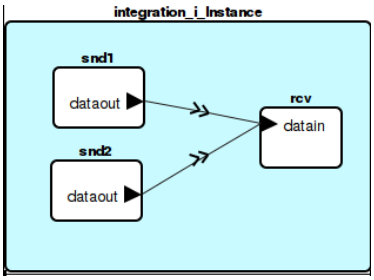

Figure 2: System Instance

The file related to this example is `check_fanin.aadl`. The related instance model is shown in figure 2. The theorem `no_double_fanin` retrieves all components within the model and validate them using the `has_single_fanin` theorem. This theorem gets all the incoming component features in a set and checks that they only have one connection.

```
no_double_fanin() <=
   ** " All incoming feature have only one connection" **
    forall (c : component) . true => has_single_fanin (c)


has_single_fanin (comp : component) <=
   ** " All incoming feature have only one connection on " comp **
   forall (f : features (comp)) . (direction(f) = "in") => (length (connections (f)) = 1)
```

The model is not validated (result shown in figure 3) because components `snd1` and `snd2` are connected to `recv` through the same incoming feature. In order to be able to validate the model, one solution is to add a new feature on `recv` and connect `snd1` and `snd2` to a single and distinct port.

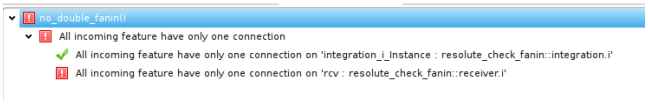

Figure 3: Analysis result for the double fanin system

In this part, we introduced the following RESOLUTE concepts:

1. Call to another RESOLUTE theorem
2. Use the `forall` keyword
3. Use the `direction` built-in function (returns either `"in"`, `"out"` or `"inout"`).
4. Use the `length` built-in function that returns the size of a set

4 Analyzing Connections Consistency

Something one might want to check is the consistency of connections in an architecture. For example, checking a characteristic (such as a property value) on features of all connections or on the sending/receiving component.

This new example defines several connected components with a property defined on their features (the same property as in the first example, `resoluteps::foobar`). We will then use RESOLUTE to check that the value on the connection source is lower than the value on the connection destination.

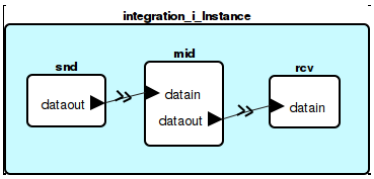

Figure 4: System Instance

The architecture of the system is shown in figure 4 and we associated the following value on the components interfaces:

- The outgoing feature of `snd` has a value of 5
- The incoming feature of `mid` has a value of 4
- The outgoing feature of `mid` has a value of 5
- The outgoing feature of `snd` has a value of 10

In order to check the architecture, we used theorem `minfoobarvalue` defined in `theorems.aadl`. The theorem get all the outgoing and incoming connections related to the component such as their source and destination have the property `resolute::foobar` defined. Then, for the source and the destination, the theorem checks that the property value on the source is lower than the property on the destination.

```
minfoobarvalue(c : component) <=
  ** " Check the minimum foobar for features on " c  **
  forall (conn : connections (c)) . has_property (source(conn), resoluteps::foobar) and
                                    has_property (destination(conn), resoluteps::foobar) =>
 (property(source(conn), resoluteps::foobar) < property(destination(conn), resoluteps::foobar))
```

Considering this constraint, the model will not be validated because in connection between `snd` and `mid`, the property value associated with the source is bigger than the one associated with the destination. Changing the property value on the source is then sufficient to validate this constraint.

In this part, we introduced the following RESOLUTE concepts:

1. Get the source and destination of a connection
2. Get the property value associated with a feature

5 Checking Compliance of an Architecture

For the last part of this tutorial, we will show how to check the structure of an architecture using RESOLUTE. To illustrate that, we will check the compliance of a model against the modeling guidelines to design ARINC653 systems. The AADL ARINC653 annex defines these rules. In a nutshell, there are the main guidelines:

- Each AADL `process` must be bound to an AADL `virtual processor`
- Each AADL `virtual processor` must be contained in an AADL `processor`
- An AADL `processor` must define its configuration using appropriate properties (health monitoring, scheduling, etc.)
- Each AADL `process` must be bound to an AADL `memory` (a memory segment), which it itself contained in another AADL `memory` (the physical memory)
- Communication interfaces (`event data port` or `data port`) must define their requirements (sampling period, queueing protocol, etc.)

We plan to validate these rules against a model introduced in the ARINC653 annex. The graphical representation of the model is shown in figure 5.

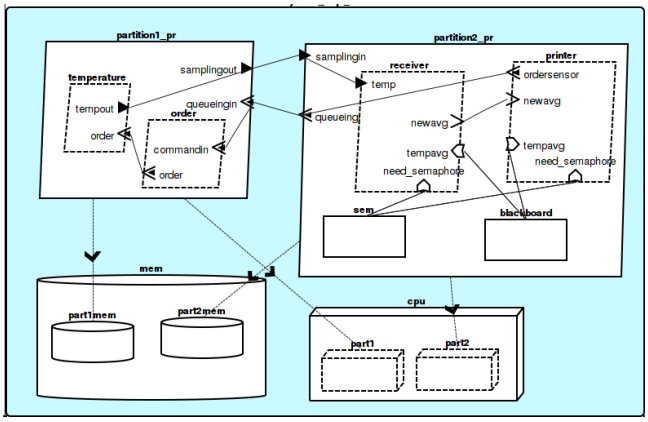

Figure 5: System Instance

In order to validate the architecture compliance with the ARINC653 modeling guidelines, we define several theorems. The top-level theorem `check_arinc653_compliance` works as follow:

1. Check the compliance of the architecture with the modeling guidelines to represent ARINC653 partitions by calling`check_arinc653_processes`. This theorem performs the following actions for each AADL `process`:
   (a) Call `check_arinc653_process_memory` to check that the AADL `process` is bound to an AADL `memory`. It also calls `check_arinc653_memory_segment`

on the memory to make sure that the memory associated with the process is contained in a top-level (physical) memory.

(b) Call `check_interfaces` to check that all communication requirements are defined using AADL properties:

- AADL `data port` define the property `ARINC653::Sampling_Refresh`
- AADL `event data port` define the property `ARINC653::Queueing_Discipline`

(c) Call `check_tasks` to check the requirements of AADL `thread` components contained in AADL `process` components. Mostly, the `check_tasks` check the interfaces for the `thread` components.

(d) Call `check_arinc653_process_virtual_processor` to check that the AADL `process` under validation is bound to an AADL `virtual processor` component.

2. Check the compliance of the architecture with the modeling guidelines to represent ARINC653 modules by calling`check_arinc653_processors`. This theorem performs the following actions for each AADL `processor`:

(a) Check that the AADL `processor` defines the properties to define its scheduling policy by calling the `check_arinc653_processor_scheduling` theorem. This theorem checks that the scheduling-related properties are correctly defined on the component (`ARINC653::Module_Schedule` and `ARINC653::Module_Major_Frame`)

(b) Check that the AADL `processor` defines the properties to define its health monitoring policy by calling the `check_arinc653_processor_hm` theorem. This theorem checks that the health-monitoring related properties are correctly defined on the component (`ARINC653::HM_Error_ID_Levels` and `ARINC653::HM_Error_ID_Actions`)

3. Check the compliance of the architecture with the modeling guidelines to represent ARINC653 partition runtime by calling`check_arinc653_virtual_processors`. For each AADL `virtual processor`, this theorem checks that it is contained in an AADL `processor` component but also defines appropriate configuration properties (`ARINC653::Partition_Identifier` and `ARINC653::Partition_Name`).

When trying to validate the model, the validation fails because the health monitoring properties are not declared (they are commented in the model). Once the properties are correctly defined, the model is validated.

In this part, we introduced the following RESOLUTE concepts:

1. Get the resource (processor or memory) bound to a component (built-in function `is_bound_to`
2. Verify the containment rules of components (built-in function `parent`)
3. Call many RESOLUTE statements within several theorems

```
check_arinc653_compliance () <=
   ** "Check compliance of the model with ARINC653 annex" **
   check_arinc653_processes() and
   check_arinc653_processors() and
   check_arinc653_virtual_processors ()


--
-- Procesor checks
--

check_arinc653_processors () <=
   ** "Check compliance of the processors" **
   forall (cpu : processor) . true => check_arinc653_procesor (cpu)

check_arinc653_procesor (cpu : processor) <=
   ** "Check compliance of processor " cpu **
    check_arinc653_procesor_scheduling (cpu) and
    check_arinc653_procesor_hm (cpu)

check_arinc653_procesor_hm (cpu : processor) <=
   ** "Check compliance of processor " cpu " for Health-Monitoring properties" **
    has_property (cpu, ARINC653::HM_Error_ID_Levels) and
    has_property (cpu, ARINC653::HM_Error_ID_Actions)

check_arinc653_procesor_scheduling (cpu : processor) <=
   ** "Check compliance of processor " cpu " for scheduling properties" **
    has_property (cpu, ARINC653::Module_Schedule) and
    has_property (cpu, ARINC653::Module_Major_Frame)


--
-- Virtual Processor checks
--

check_arinc653_virtual_processors () <=
   ** "Virtual Processors are in processors" **
   forall (vp : virtual_processor) . true =>
               (exists (cpu : processor) . parent(vp) = cpu) and
               (has_property (vp, ARINC653::Partition_Identifier)) and
               (has_property (vp, ARINC653::Partition_Name))

--
-- Process checks
--
check_arinc653_processes() <=
   ** "All processes are bound to a memory segment and a virtual processor" **
   forall (p : process) . true =>  check_arinc653_process_memory (p) and
                                   check_interfaces (p) and
                                   check_tasks (p) and
                                   check_arinc653_process_virtual_processor (p)

check_tasks (p : process) <=
   ** "Check tasks from process " p **
   forall (thr : thread) . (parent (thr) = p) => check_interfaces (thr)

check_interfaces (comp : component) <=
   ** "Check that component " comp " declares all necessary properties on its ports" **
   forall (poevent : features (comp)) . is_event_port (poevent) and (direction(poevent) = "in") => has_property (poevent, ARINC
   and
   forall (pononevent : features (comp)) . (is_event_port (pononevent) = false) and (direction(pononevent) = "in") => has_prope


check_arinc653_process_memory (p : process) <=
   ** "Check that process " p " is associated with a memory" **
   exists (segment : memory) . (is_bound_to (p, segment)) and check_arinc653_memory_segment (segment)

check_arinc653_memory_segment (segment : memory) <=
```

```
  ** "Check that the memory segment " segment " is contained in a memory" **
  exists (mem : memory) . (parent(segment) = mem)

check_arinc653_process_virtual_processor (p : process) <=
  ** "Check that process " p " is associated with a virtual processor" **
  exists (runtime : virtual_processor) . (is_bound_to (p, runtime))
```

6 Conclusion

This tutorial explains the core principles of RESOLUTE and how to use the language to validate an architecture. If you have any question or experience issue, please visit the project page [3] and submit an issue on the issue tracker.

[3] project page on github on `https://github.com/smaccm/smaccm`