Resolute Reference Guide

Peter Feiler Julien Delange

Contents

Chapter 1

Introduction

Resolute claim functions, computational functions, and global constants are defined in Resolute annex libraries, *i.e.*, Resolute annex clauses placed directly in an AADL package.

```
package BudgetCase
public

annex Resolute {**
    MaximumWeight : real = 1.2kg

    SCSReq1(self : component) <=
        ** "R1" SCS shall be no heavier than " MaximumWeight%kg **
        SCSReq1VA1(self, MaximumWeight) or SCSReq1VA2(self,
            MaximumWeight)

    AddBudgets(self : component) : real =
    sum([WeightBudget(t) for (t: subcomponents(self))])
```

Listing 1.1: Resolute Claim Functions

A claim function can be associated with component implementations by `prove` statements declared in a Resolute annex subclause. The example shows the `prove` statement for a claim function `SCSReq1` with the component itself passed in as a parameter.

```
system implementation SCS.Phys
    subcomponents
        sensor1: device sensor;
        sensor2: device sensor;
        actuator: device actuator;
    annex Resolute {**
```

```
        prove (SCSReq1(this))
```

Listing 1.2: Prove Statement for a Claim Function

Invoke the *Resolute* command on a component implementation. This results in an instantiation of the component implementation and the application of all claim functions associated with all of the components in the instance model via the `prove` statements.

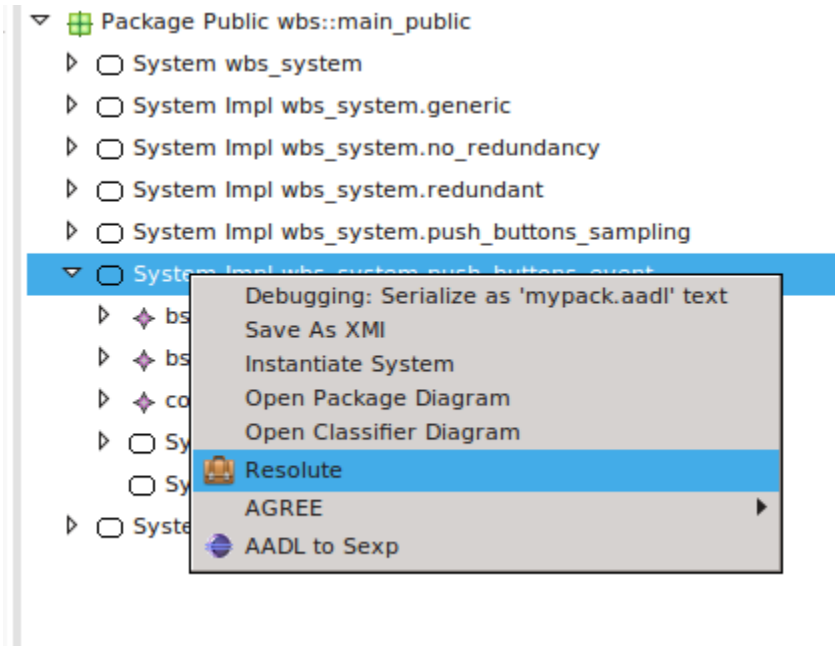

Figure 1.1: Instantiation from OSATE menu

The verification results are then displayed in a view labeled *Assurance Case*.

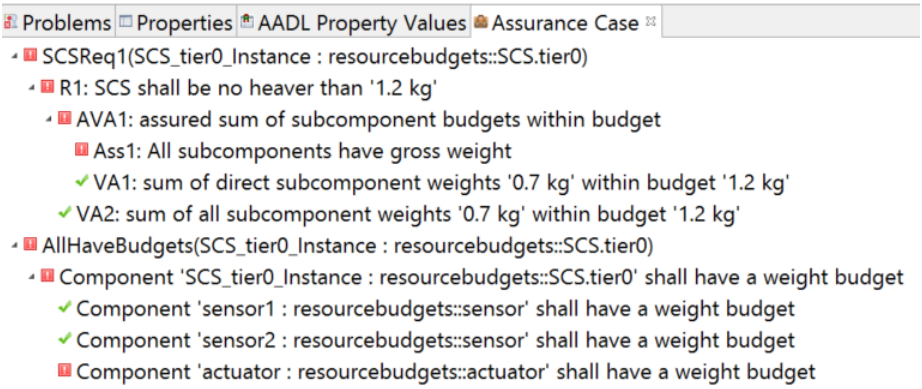

Figure 1.2: Assurance Case Tree View

Chapter 2

Claim Functions

The syntax of a claim function is as follows:

```
<Claim_Function> ::=

    <name> "(" (<parameter> "," <parameter) )* )? ")" "<="
    "**" <description> "**" <claim_function_expression>
<attribute> ::= <name> ":" <value>
<parameter> ::= <name> ":" <type>

<description> ::= <text> ( <attribute_reference> |
    <parameter_reference> | <text>)*
```

Listing 2.1: Claim Function Syntax

Attributes can be `context`, `assumption`, `strategy`, or `justification` statements. Each attribute has a name and a value. `context` values can be any Resolute type (see Resolute Type System), but `assumption`, `strategy`, and `justification` values are currently limited to text strings.

The parameter declaration consists of a name and a type (see Resolute Type System).

The description for a claim function consists of a sequence of strings and references to claim function parameters, global constants, or local constants (defined by `let` statements). If your values are numerical with a unit, you can specify the unit to be used for display by indicating the desired unit after a `%`, *e.g.*, WeightLimit%kg. The units are those defined by Units property types in property sets and do not have to be qualified by the Units type.

The claim function expression is assumed to be a logical expression (`and`, `or`, `andthen`, `orelse`, implies (`=>`)) or quantified expressions (`forall`,`exists`) to

represent a predicate. You can also and `let` expressions to compute local variables.

In the case of `and`, `or` and `forall`, all expression elements are executed and then the result is evaluated and returns true only if all claim functions return true. Executing all claim functions allows Resolute to inform the user of all failing claim functions rather than not executing the remaining claim function calls after a failing claim in the case of `and` and `forall` and after a successful claim in the case of `or`.

`andthen` conditionally executes the second operand if the first one is true, i.e., the first operand acts as a precondition to the second one. This is useful when checking that a property exists (`has_property`) before retrieving it (`property`) as `property` throws an exception when the value is not found.

`orelse` conditionally executes the second operand as an alternative if the first one is false. For example, this is useful when a predicate is evaluated in a compute function and if it returns false we want to provide an explanatory message using `fail`.

`exists` will execute elements in the provided collection until one evaluates to true. Note that provided collections are not always in the order you expect, e.g., `subcomponents(c)` returns a collection that is ordered by component name, not the declaration order.

`=>` only executes the second operand if the first one is true. Note, however that if the left operand is false the result of `=>` is true without execute the right operand.

A Resolute annex library is declared directly in a package through the `annex` Resolute `{**` *<library content>* `**};` statement. Typically you want to place a Resolute library in a package by itself - separate from component declarations.

Resolute assumes that there is a global name space for the names of claim functions and computational functions. Therefore, their names must be globally unique. Claim and computational functions can be referenced in `prove` statements, other claim functions, or computational functions without qualification by a package name.

> You may have two claim functions with the same name in different packages. Resolute will not complain and will use the first one it encounters when resolving references.

Context attributes are visible within the claim in which they are declared, as well as all descendant claims.

2.1 Application of Claim Functions

Claim functions are invoked on a component by specifying a `prove` statement in a Resolute annex subclause of the component implementation. This claim function is then executed on every instance of this component implementation when the *Resolute* command is invoked.

A component implementation can contain multiple `prove` statements.

```
<Resolute_subclause> :== "annex" "resolute" "{**" <prove_statement>
    ( <prove_statement> )* "**}" ";"

<prove_statement> ::= "prove" "(" <claim_function_invocation> ")"

<claim_function_invocation> ::= <claim_function_name> "("
    ( <parameter_value> ("," <parameter_value>)* )? ")"
```

Listing 2.2: Multiple Prove Statements

The parameter values can be integers, real numbers, strings, Boolean literals, model elements, references to global constants, and collections of these types. They must match the specified type for the parameter.

One special model element reference is expressed by the keyword `this`. It refers to the instance model object of the model element that contains the `prove` statement. This is the root object of an instance model as Resolute create a separate instance model every time it encounters a prove statement.

```
prove ( Memory_safe ( this ))
```

You can also identify a subcomponent of `this`; *i.e.*, associate the verification action with a component down a path of the architecture hierarchy. This allows you to specify a verification action for a specific component instance. This example shows how a `prove` statement is applied to a subcomponent called subsystem1:

```
prove (Fully_Connected ( this.subsystem1 ))
```

> The keyword `this` can only be used in the `prove` statement — not in claim functions or compute functions. Inside claim or compute function you can access elements of the instance model through set constructors such as `component` or `thread`.

The `prove` statement can be associated with the component classifier of the subcomponent. In that case, it applies to all instances of that component. We recommend that you associate `prove` statements with a path only if the `prove` is intended for that particular instance of a subcomponent.

2.2 Uses of Claim Functions

The compiler does enforce that claim functions can be invoked only in `prove` statements, and in other claim functions, and cannot be invoked in computational functions.

Chapter 3

Goal Structuring Notation

Resolute can be used to construct assurance cases that comply with the GSN v2 standard. The core elements in the GSN are * Goal * Strategy * Context * Assumption * Justification * Solution * Undeveloped classifier

Goals can be supported by goals and strategies. Strategies can only be supported by goals. Both goals and strategies can contain contextual information such as assumptions and justifications. Leaf nodes of the GSN argument will either be evidence to support the claim, or an indication that the argument requires further development.

A Resolute claim is the GSN equivalent of a goal or strategy. A claim in Resolute can be annotated with the `goal` or `strategy` keyword to distinguish between the two. A Resolute claim that is not annotated with either keyword is assumed to be a goal. For example, explicit declarations of claims as a goal and strategy would look like:

```
goal G25(Monitors : {component}, Contingency_Manager : component) <=
    ** "Runtime monitors detect the error condition and intervene
        when it occurs" **
    S25(Monitors, Contingency_Manager)

strategy S25(Monitors : {component}, Contingency_Manager : component)
    ** "Argue based on simplex architecture" **
    G27(Monitors) and G28(Contingency_Manager)
```

Listing 3.1: Resolute Goal

In the example above, the goal contains the rules for evaluating whether sufficient evidence exists to support the goal. Assurance arguments will typically include one or more strategies for supporting a goal. In Resolute, strategies are referenced in goals as claim calls. In the example above, the call to S25()

indicates that the goal is supported by strategy S25(). In turn, strategy S25() call two sub-goals G27() and G28().

Strategies can be declared using a short-hand notation as well. When it is not necessary to create a unique claim to represent a strategy, a goal can declare a strategy "in-line" as an attribute. For instance, the following is equivalent to the example above:

```
goal G25(Monitors : {component}, Contingency_Manager : component) <=
    ** "Runtime monitors detect the error condition and intervene
        when it occurs" **
    strategy S25 : "Argue based on simplex architecture";
    G27(Monitors) and G28(Contingency_Manager)
```

Listing 3.2: Resolute Strategy

Goal and strategy claims can include zero or more *claim attributes*. Claim attributes are declared after the claim description and before the claim body. The in-line `strategy` attribute can only be declared in a goal. The other three types of claim attributes, `context`, `assumption`, and `justification` can be declared in both goals and strategies.

The context attribute provides a means for attaching contextual information to a goal. As per the GSN standard, context is visible to a goal and all its sub-goals. In Resolute, this means that a context element can be references by descendants of a goal without having to pass the context as an argument from one goal to the next. For example, the following is valid syntax:

```
goal G25(Monitors : {component}, Contingency_Manager : component) <=
    ** "Runtime monitors detect the error condition and intervene
        when it occurs" **
    context C1 : Monitors;
    strategy S25 : "Argue based on simplex architecture";
    G27() and G28(Contingency_Manager)

goal G27() <=
    ** "The error condition can be detected when it occurs" **
    strategy S27: "Reason over multiple monitors"
    forall(m : C1) . G102(m)
```

Listing 3.3: Resolute Context

In the above example, context C1 is declared and set to the value of the Monitors parameter passed into the goal. Sub-goal G27() is then evaluated with zero arguments. However, G27() can still refer to context C1 declared in an ancestor goal. Context attributes can be any Resolute type.

Unlike context attributes, assumption, justification, and strategy attributes can only have string values. These elements are mainly used to provide descriptive information within the assurance argument to aid human evaluators in

an assessment. Therefore, Resolute does not evaluate these types of attribute statements.

To determine whether sufficient evidence exists to support a goal, Resolute evaluates the body of claims. However, in some instances the user will want to *assert* that the evidence exists without requiring Resolute to do the evaluation. Resolute therefore includes a `solution` keyword to be placed in the body of a claim. For example:

```
goal G28(Contingency_Manager : component) <=
    ** "When the error condition is detected a recovery mechanism
        intervenes" **
    solution Sln28 : "Recovery mechanism implemented"
```

Listing 3.4: Resolute Solution

When Resolute evaluates a claim that contains a solution element in its body, the claim evaluates *true*.

When an argument has not been completed, it is necessary to indicate that. GSN defines an *undeveloped* element that can be applied to a goal or strategy. Resolute includes support for such arguments with an `undeveloped` keyword that can be placed in the body of a claim. For example:

```
goal G28(Contingency_Manager : component) <=
    ** "When the error condition is detected a recovery mechanism
        intervenes" **
    undeveloped
```

Listing 3.5: Resolute Undeveloped

When Resolute evaluates a claim that contains an undeveloped element in its body, the claim evaluates *false*.

Chapter 4

Computational Functions and Constants

4.1 Computational Functions

Computational functions are used to calculate a value of any type. The result can be Boolean, numeric, model elements, or collections of items of a specific type. Computational functions take parameters that are typed. Computational functions have a single expression that can be preceded by a local constant declaration.

```
<computational_function> ::=
    <function_name> "(" <parameter> ("." <parameter>)* ")" ":"
        <return_type> "="
    <computational_expression>
```

Listing 4.1: Computational Functions

- Computational functions are defined in Resolute libraries.
- Computational functions can be invoked in expressions of claim functions. Typically they are invoked in claim functions that represent verification actions or assumptions.

4.2 Global Constants

Global constants represent parameters to the verification whose value is set once and can be used in any computational expression, including parameters to claim function calls. Global constants can also hold the result of a computational function or a set constructor whose value can be determined at startup time of a

Resolute command. For example, a global constant may be used to precompute various sets of model element instances, *e.g.*, all elements that are reachable from a component of a certain component type.

```
<global_constant> ::=
    <constant_name> ":" <type> "=" <computational_expression>
```

Listing 4.2: Global Constants

- Global constants are defined in Resolute libraries.
- Global constants can precompute any expression.

4.3 Local Constants

Resolute also supports precomputation of local constants, which are used inside a claim function or computational function. One or more local constants can be defined before any expression. Typically, they are used in a verification action or computational function before the logical or computational expression. However, they can also be used before any subexpression, *e.g.*, before the right-hand subexpression of an `and` or `+` operator.

```
<local_constant> ::=
    "let" <constant_name> ":" <type> "=" <computational_expression>
        ";"
```

Listing 4.3: Local Constants

- The scope of a local constant is the expression; *i.e.*, they can be referenced only from within the succeeding expression.
- Local constants are used to precompute values that may be referenced multiple times in the succeeding expression.

Chapter 5

Predicate Expressions and Computational Expressions

A constraint expression results in a Boolean value.

5.1 Predicate Expressions

Predicate expressions support the following operators in increasing precedence order:

Logical operators (the operands a, b are expressions of type Boolean):

- a `=>` b
- a `or` b *and* a `orelse` b
- a `and` b *and* a `andthen` b
- Negation: `not` a
- Quantified logical expressions: (`forall` | `exists`) `(` <variablename> `:` <collection_constructor> `)` `.` <logical_expression>

5.2 Computational Expressions

Computational expressions are used in computational functions and must return a value of the specified type. Computational expressions include constraint expressions, arithmetic expressions, and operations on collections of values and model elements.

Relational operators (the operands are of type `real` or `int`):

- `<` | `<=` | `>` | `>=` | `=` | `<>`

Arithmetic operators (the operands are of type `real` or `int` and may include a unit):

- `+` | `-`
- `*` | `/`
- exponent: `^`
- Negation: `-` a
- Precedence brackets: `(` a `)`

  The precedence order is Relational operators (low) to Arithmetic in increasing order. They are higher than `and` *and* `andthen`.

5.2.1 Type-related operators

- Type cast: `(` <type> `)` a

  Type cast is the same precedence as `not` *and* <negations>.

5.2.2 Atomic expressions

Atomic expressions can be used as operands and have highest precedence.

- Base type values: integer value, real value, string value, and Boolean value. Integer and real values can be annotated with a unit. Any unit defined by a Unit property type in any of the property sets is acceptable.
- Global and local constant reference and parameter reference by their identifier
- Computational function invocation:
  function_name `(` (<parameter_value> (`,` <parameter_value>)*)? `)`
- Conditional value: `if` condition `then` expression `else` expression
- Qualified AADL classifier or property definition: (<ID> `::`) *<ID> (`.` <ID>)?
  - Classifier used only as a parameter to `instance` or `instances` and property definition only in `property` built-in function
- Instance model reference: `this` (`.` <ID>) *
  - Used only as parameter in `prove` statement

5.2.3 Collection-related operators

Collections in Resolute can be either lists or sets. Lists can contain multiple identical elements, preserve insertion order, and support `head` and `tail` accessors for defining recursive processing. By contrast sets can contain no more than one element of a given value, do not necessarily support insertion order, and support set `intersect` and `union` operators. Both lists and sets support quantifiers `forall` and `exists` and filtered collections. Operators exist to convert a list into a set (collapsing duplicated elements) and a set into a list.

- Basic collection: <Basic_list> | <Basic_set>
- Basic list: `[` <expression> (`,` <expression>)* `]` | `[ ]`
- Basic set: `{` <expression> (`,` <expression>)* `}` | `{ }`
- Filtered collection: <Filterd list> | <Filtered set>
- Filtered list: `[` <filtered_element> `for` (`(` <element_name> `:` <collection_constructor> `)` +) (`|` <filter_expression>)? `]`
- Filtered set: `{` <filtered_element> `for` (`(` <element_name> `:` <collection_constructor> `)` +) (`|` <filter_expression>)? `}`
    - Note: <filtered_element> refers to one of the set element names
    - Note: <filter_expression> is of type Boolean

```
<collection_constructor> ::=
    <basic_collection> | <filtered_collection> |
        <AADL_model_element_type> |
    <global_constant> | <local_constant> |
        <computational_function_invocation>
```

Listing 5.1: Collection-Related Operators

Note: The constants must have collections as their values, and the invoked function must return a collection. Function invocations returning a collection can be a user-defined computational function or a built-in function (see Built-in Base Types). The *constant reference* has to be of a collection type.

The following examples illustrate the use of collections. The first example uses the built-in `subcomponents` function to get a collection of subcomponents. The `forall` then iterates over the (set) collection and executes the built-in `has_property` constraint function on each element.

In the second example, we precompute the collection of subcomponents and hold on to them with a local constant. We then construct a list of real values of value 1.0 for each subcomponent that satisfies the `has_property` constraint function, then perform the summation of the resulting `real` (list) collection, and divide it by the size of the subcomponent collection.

```
HasSubcomponentWeightBudget(self:component) : bool =
    forall (sub: subcomponents(self)) .
        has_property(sub,SEI::GrossWeight)

SubcomponentWeightBudgetCoverage(self:component) : bool =
    let subs : {component} = subcomponents(self);
    (sum([ 1.0 for (sub : subs) |
        has_property(sub,SEI::GrossWeight)]) / length(subs))
```

Listing 5.2: Subcomponent Weight Coverage Example

Collections can also be precomputed in global constants. This is useful when you want to make use of certain collections of instance model objects repeatedly. In this example, the global constant declaration `MOTORS` represents the set of instances of a particular component type.

```
MOTORS : {component} = instances(PX4IOAR::Motor)
```

5.2.4 The `fail` expression

The `fail` expression can be used in any computational function and can be viewed like an exception that is thrown. It is automatically caught by the closest enclosing claim function, interpreted as a fail of the claim, and reported as a sub-result to the claim function. That is, the `fail` expression is shown as a failure, and the provided text explains the failure.

- Exception: `fail` <string value> or `fail **` <description> `**` with the description syntax the same as for claim functions

Chapter 6

Resolute Type System

```
<type> ::=
    <list_type> | <set_type> | <base_type> |
        <AADL_model_element_type>

<list_type> ::= "[" <type> "]

<set_type> ::= "{" <type> "}
```

Listing 6.1: Resolute Type System

The list and set collections allow multiple elements of the same value. In the *SubcomponentWeightCoverage* example, the collection concept has multiple instances of the value 1.0, and each is counted in the summation.

6.1 Built-in Base Types

Base type:

- `int`
- `real`
- `string`
- `bool`
- `range`

6.2 Arithmetic with Integers and Reals

`int` and `real` — as well as the min and max of a `range` — can be values specified with a measurement unit. Any of the unit literals defined in AADL2 Units

property types are acceptable. The Units property type definition specifies the ratios to be used to perform conversion between the units. For `int` and `real` values with units, Resolute converts the value to a value relative to the base unit (the first unit defined in the Units type). All arithmetic is performed based on those values. To present results in the description of a claim function of a `fail` operation, the value is converted to the unit specified in the description specification.

Resolute can retrieve property values with built-in functions. The property values for `aadlinteger` are mapped into `int`, `aadlreal` into `real`, and `range of` into `range`.

6.3 AADL Model Element Types

AADL model element types have an implied type hierarchy. The nesting level indicates this type hierarchy.

- `aadl` [any AADL model element]
  - `component` [any category of AADL component]
    - `abstract` [AADL abstract component]
    - `bus`
    - `data`
    - `device`
    - `memory`
    - `processor`
    - `process`
    - `subprogram`
    - `subprogram_group`
    - `system`
    - `thread`
    - `thread_group`
    - `virtual_bus`
    - `virtual_processor`
  - `connection` [AADL connection instance]
  - `property` [AADL property definition]
  - `feature` [any AADL feature]
    - `port` [any AADL port]
      - `data_port`
      - `event_port`
      - `event_data_port`
      - `feature_group`
    - `access` [any AADL access feature]
      - `bus_access`
        - `provides_bus_access`
        - `requires_bus_access`

- data_access
  - provides_data_access
  - requires_data_access
- subprogram_access
  - provides_subprogram_access
  - requires_subprorgam_access
- subprogram_group_access
  - provides_subprogram_group_access
  - requires_subprogram_group_access

Resolute operates on the instance model; *i.e.*, the model elements represent instances. Built-in collection functions operate on instance model elements or retrieve the set of instances for a given classifier (see Built-in Functions).

Chapter 7

Built-in Functions

7.1 Built-in Functions for Collections

`union`(<set>, <set>): set - returns a set collection that is the union of the two inputs

`intersect`(<set>, <set>): set - returns a set collection that is the intersection of the two inputs

`length`(<collection>): int - returns the size of the given set or list collection

`size`(<collection>): int - returns the size of the given set or list collection (same as length)

`member`(<element>, <collection>): Boolean - returns true if the element is a member of the set or list collection

`sum`(<numeric_list>): numeric - calculates the sum of a list collection of integers or a list collection of real

`head`(<list>): type - returns the first element of the list collection

`tail`(<list>): list - returns all but the first element of the list collection

`append`(<list>, <list>): list - returns a list collection that is the concatenation of the two given list collections

`as_set`(<list>): set - returns a set collection containing all of the unique elements contained in the given list collection

`as_list`(<set>): list - returns a list collection containing all of the elements contained in the given set collection

7.2 Built-in Functions for Ranges

`upper_bound`(<range>): numeric - returns the upper bound of the range

`lower_bound`(<range>): numeric - returns the lower bound of the range

7.3 Built-in Functions for Properties

`has_property`(<named_element>, <property>): Boolean - the named element has the property.

`property`(<named_element>, <property>, <default value>*): value - returns the value of the property. If a default value is supplied, then it is returned if the element does not have the property value. If no default is supplied and the value does not exist, a resolute failure exception is thrown, which is caught by the closest enclosing claim function and interpreted as a fail.

`property`(<property>) value - returns the value of the property constant.

`property_member`(<record_property_value>, <field name>): Boolean - return the value of the record field.

> Note: There is no constructor for record values. To compare a property record value to some actual record value you have to write a function that compares each of the fields. `=` can be used to compare two property record values.

`enumerated_values`(<property>): [<string>] - return the an ordered set of string values.

7.4 Built-in Functions for Any Model Element and Components

`name`(<named_element>): string - returns the name of the named element

`has_type` (named_element): Boolean - returns true if the named element has a classifier. The named element can be a component, feature, or connection instance. In the case of a connection, the type of the feature is the connection end.

`type`(<named_element>): Classifier - returns the classifier of a component, feature, or connection. In the case of a connection, the type is that of the connection source (if not present the destination) feature. The named element must have a type, otherwise a resolute failure exception is thrown and caught by the closest enclosing claim function.

`is_of_type`(<named_element>, <classifier>): Boolean - true if the named element has the classifier or one of its type extensions. The named element must have a type. The named element can be a component, feature, or connection instance. In the case of a connection, the type of the feature is the connection end.

`has_parent`(<named_element>): Boolean - returns true if the component has an enclosing model element

`parent`(<named_element>): named_element - returns the parent of the named element. The parent must exist.

`has_member`(<component>, <string>): Boolean - true if the component has a member with the specified name (string). Members are features, subcomponents, etc. The component can be a component instance or a component classifier.

> Note: Feature instances representing feature groups can have feature instances as members, but they are not handled by this function. See pre-declared library below for flattening feature instances in feature groups.

`is_in_array`(<component>): Boolean - returns true if the component instance in in an array, i.e., has an index into the array.

`has_prototypes`(<component>): Boolean - returns true if component classifier contains prototype declarations.

`has_modes`(<component>): Boolean - returns true if component directly contains mode instances.

`is_procesor`(<component>): Boolean - true if the component instance is a processor

Other built-in component category tests are: `is_virtual_procesor`, `is_system`, `is_bus`, `is_virtual_bus`, `is_device`, `is_memory`, `is_thread`, `is_process`, `is_data`, `is_subprogram`.

> Missing tests (`abstract`, `thread_group`, `subprogram_group`) can be tested by `\<object\> instanceof \<aadl model element type\>`

7.5 Built-in Functions for Features

`direction`(<feature>): string - returns the direction of a feature instance as string (`in`, `out`, `in out`/`in_out`). Access features do not have direction.

`is_event_port`(<feature>): Boolean - true if the feature instance is an event port or event data port

`is_data_port`(<feature>): Boolean - true if the feature instance is an data port or event data port

`is_port`(<feature>): Boolean - true if the feature instance is a port

`is_abstract_feature`(<feature>): Boolean - true if the feature instance is an abstract feature

> Note that you can test any feature or component by writing `\<object\> instanceof \<aadl model element type\>`

7.6 Built-in Functions for Connections

`source`(<connection>): connection_endpoint - returns the component or feature instance that is the source of the connection instance

`destination`(<connection>): connection_endpoint - returns the component or feature instance that is the destination of the connection instance

`is_data_access`(<connection>): Boolean - true if one end of a connection is a data component

`is_bus_access`(<connection>): Boolean - true if one end of a connection is a bus component

`is_bidirectional`(<connection>: Boolean - true if connection is bidirectional

7.7 Built-in Functions for Bindings

`is_bound_to`(<binding_source>, <binding_target>): Boolean - true if the binding source (a component or connection instance) is bound to the binding target (a component). It handles processor bindings, memory bindings, and connection bindings.

> Note: The `is_bound_to` function is the same as library function `bound`.

> Note: The `is_bound_to` function does not consider function_bindings. See *Resolute_Stdlib.aadl* for how this can be done by mirroring *processor_bound*.

7.8 Built-in Functions for Sets and Lists

Resolute operates on the instance model; this means that the collections are of instance model elements.

`features`(<named_element>): {feature} - returns a collection containing the features of the named element

`subcomponents`(<named_element>): {component} - returns a collection containing the subcomponents (component instances) of the named element

`connections`(<named_element>): {connection} - returns a collection of connection instances for which the named element is an end point (source or destination). The named element can be a component instance or a feature instance.

`instances` (<component_classifier>): {component} - returns the collection of instances in the instance model for a given component classifier

`instance` (<component_classifier>): component - returns the component instance for a given component classifier. The method assumes that there is only one instance.

7.9 Built-in Functions for Flows

`end_to_end_flows` (<component>): { <end_to_end_flow> } - returns set of end to end flows contained in component instance.

`flow_elements` (<end_to_end_flow>): { <flow_element> } - returns set of flow elements, which are connection instances, flow spec instances, or components instances.

`flow_specifications` (<component>): { <flow_spec> } - returns set of flow specification instances of a component.

`flow_source` (<flow_spec>): feature - returns the source of a flow specification.

`flow_destination` (<flow_spec>): feature - returns the destination of a flow specification.

7.10 Built-in Functions for Error Models

`error_state_reachable` (<component>, <state: string>): Boolean - true

`propagate_error` (<component>, <error_type: string>): Boolean - true if the component instance propagates out the error type on any of its features if the error state of the component instance is reachable by an incoming transition

`receive_error` (<component>, <error_type: string>): Boolean - true true if the component instance receives the error type of a propagated error on any of its features. Not supported yet.

`contain_error` (<component>, <error_type: string>): Boolean - true if the the component instance has an error event with the specified error type. Not supported yet.

7.11 External Functions

`analysis` (<function: string>, <args>): <ResoluteValue> - invocation of a Java function registered as an external function extension point. The function is specified as string identifier of the extension point. The arguments are additional parameters of the analysis function.

The return value must be one of the ResoluteValue subclasses: Boolvalue, IntValue, ListValue, NamedElementValue, RangeValue, RealValue, ResoluteRecordValue, SetValue, StringValue.

While the `analysis` capability in Resolute enables the execution of external plugins, a separate plugin is required for each analysis call. This is an inefficient mechanism for encapsulating a group of related functions. For example, a library of string manipulation functions (such as concat(), length(), substring(), etc.) would each require an individual plugin using the analysis() call, as in analysis(concat, str1, str2).

Resolute external function library support provides a mechanism for packaging multiple functions into a single plugin, which can then be called in a Resolute claim using the syntax > `\<LibraryName\>.\<LibraryFunction\>(Arg1, Arg2, …)`

For example, the concat function in string manipulation library is called as > `StringLib.concat(str1, str2)`

and returns a string.

7.12 Debug Functions

`debug` (<args>): true - writes one or more arguments (strings and other base types, sets, lists, model elements as names) to the console.

> The debug trace is written to a console in OSATE.
>
> You can enable and disable debug tracing through commands in the context menu of the Assurance Case View that comes with Resolute.

Chapter 8

Common Resolute Function Library

8.1 Binding Related Functions

`bound`(<component, binding_target>): Boolean - true if the component instance is bound to the binding target by actual processor, memory, or connection binding. Note: `bound` is the same as the built-in `is_bound_to` function.

`processor_bound`(<component>, <binding_target>): Boolean - true if the component instance is bound to the binding target by actual processor binding

`memory_bound`(<component>, <binding_target>): Boolean - true if the component instance is bound to the binding target by actual memory binding

`connection_bound`(<component>, <binding_target>): Boolean - true if the component instance is bound to the binding target by actual connection binding.

> Note: You may want to implement a `function_bound` function. See *Resolute_Stdlib.aadl* for how this can be done by mirroring *processor_bound*.

8.2 Connection Related Functions

`connected`(<source component>, <connection>, <destination component>): Boolean - returns true if the components are the soruce and destination components of the connection..

`source_component`(<connection>): component - returns the component that is the source of the connection instance. This component contains the feature

instance as a connection end point.

`destination_component`(<connection>): component - returns the component that is the destination of the connection instance. This component contains the feature instance as a connection end point.

`is_port_connection`(<connection>): Boolean - true if the connection is a connection between ports

`is_data_port_connection`(<connection>): Boolean - true if one of the connection end points is a data port. Note: should be determined by the destination.

`is_event_port_connection`({connection>): Boolean - true if one of the connection end points is an event port. Note: should be determined by the destination.

`is_event_data_port_connection`(<connection>): Boolean - true if one of the connection end points is an event data port. Note: should be determined by the destination.

`is_data_access_connection`(<connection>): Boolean - true if one of the connection end points is a data access feature.

8.3 Model Element Containment

`contained`(<named_element>, <container_component>): Boolean - true if the named element is contained in the container component. The named element can be a component or feature instance. Note: also works for connection instance.

`containing_component`(<named_element>): component - returns the directly containing component instance. The named element can be a component or feature instance. Note: also works for connection instance.

8.4 Handling of Feature Groups

Feature groups are represented in the instance model as a hierarchy of feature instances reflecting the nesting of the feature group.

`flatten_feature`(<feature>): {feature} - returns a set of feature instances that are the leaf elements of a given feature instance. If no elements are contained in the feature instance, the feature instance itself is returned as a set.

`flatten_features`(<{feature}>) : {feature} - returns a set of feature instances that are the leaf elements of a given feature instance set.

Chapter 9

Resolute Examples

9.1 Debugging Models with Resolute

Get a model element trace on the Console View. You enable and disable console logging through a command in the context menu of the Assurance Case View of Resolute. Once enabled the built-in function **debug** will write to the console when executed.

The built-in function **debug** takes one or more parameters and writes them out. It can handle any Resolute base type, lists and sets, as well as AADL model elements.

```
    let conns : {connection} = connections;
    debug( "Set of connections ",conns)
```

Listing 9.1: Model Element Trace in the Assurance View

Get a trace recorded as a result object in the Assurance Case View of Resolute we use claims functions as the examples shown below :

```
-- record a model element in the result structure
record_aadl(a : aadl) <=
    ** a **
    true

-- record a set of model elements in the result structure
record_set(s : {aadl}) <=
    ** s **
    true

-- record a list of model elements in the result structure
```

```
record_list(s : [aadl], msg: string) <=
     ** msg ": "s **
     true
```

Listing 9.2: Model Element Trace in the Assurance View

9.2 Reachable Model Elements

This is a set of functions that collect AADL components that are directly or indirectly reachable via connections.

```
-- return set of reachable components that have no outgoing
    connections.
reachable_endpoints(c : component) : {component} =
      let outconns: {connection} = outgoing_component_connections(c);
      if  outconns = {}
      then {c}
      else
        {c2 for (conn : outconns) (c2 :
            reachable_endpoints(destination_component(conn)))}


-- all components that are reachable via port or data access
    connections from the given component
-- we consider direct and indirect reachability
-- the use of a set ensures that we do not recurse forever
all_reachable_components(c : component) : {component} =
    recusive_reach({c})

-- support method to expand on a set of reachable components by the
    next directly rachable components
-- the use of a set ensures that we do not recurse forever
recusive_reach(curr: {component}) : {component} =
    let next_ones : {component} = {y for (x : curr) (y :
        directly_reachable_components(x))};
    let next_new_ones : {component} = {ele for (ele : next_ones)|
        not(member(ele,curr))};
    if (next_new_ones = {} ) then
        curr
    else
        union(curr, recusive_reach(next_new_ones))


    -- return set of components that are directly reachable via
        outgoing connections.
```

```
directly_reachable_components(comp: component):{component} =
let outconns : {connection} =
    outgoing_component_connections(comp);
{otherend for (conn: outconns) (otherend:
    other_connection_end(conn,comp))}


-- return all outgoing connections
-- in the case of data access conneciton we need to consider
    acces rights to determine the direction
-- in the case of port connecitons we need to filter out in/out
    ports with incoming connections.
outgoing_component_connections(comp: component): {connection} =
{ conn for (conn: connections(comp)) |
    if (is_access_connection(conn)) then
    -- access feature has write access
        source(conn) instanceof access and
            has_write_access(source(conn)) or
        destination(conn) instanceof access and
            has_write_access(destination(conn))
    else
    -- we have a directed feature connection
    -- it is not an incoming connection with the source an out
        only feature
        not(destination_component(conn) = comp
        and direction((feature)source(conn)) = "out"
    )}

-- For a given connection and a given component return the
    component on the other end of the connection
-- This funciton is useful to deal with access connections or
    bi-directional port connections
-- where the source could be either end.
other_connection_end(conn: connection,comp: component): component
    =
     if (destination_component(conn) = comp)
     then
         source_component(conn)
     else
         destination_component(conn)
```

Listing 9.3: Reachability Example

Chapter 10

Copyright

This material is based upon work funded and supported by the Department of Defense under Contract No. FA8721-05-C-0003 with Carnegie Mellon University for the operation of the Software Engineering Institute, a federally funded research and development center.

DM-0002203