Resolute Documentation

Peter Feiler (phf@sei.cmu.edu), Julien Delange (jdelange@sei.cmu.edu)

Jan 6, 2015

Introduction

Resolute allows users to define a set of claim functions and associate them with an AADL model. The claim functions can be used to represent the requirements to be satisfied, the verification actions used to verify them, and assumptions made by a verification action in order to produce a valid result. The requirements are expressed as a predicate on subrequirements and verification actions. The verification actions and assumptions are expressed as predicates and computational functions on the AADL model (the predicate and computational function notation has its roots in Lute and REAL). The claim functions can be organized into a hierarchy where a claim function is satisfied only if its subclaim functions are satisfied according to the specified predicate logic.

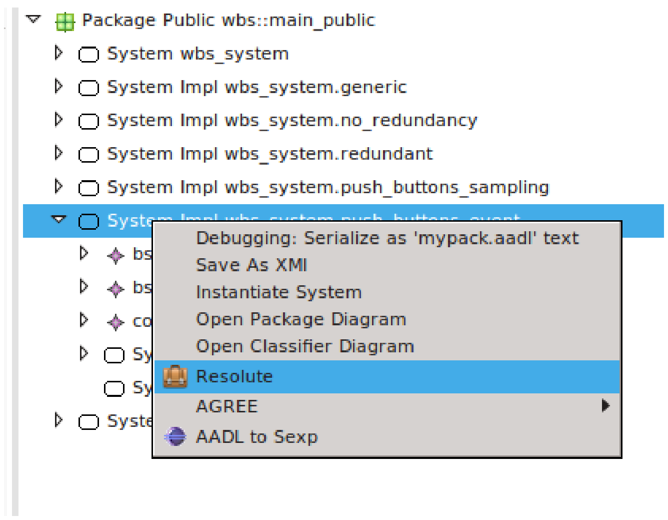
Claim functions and computational functions are defined in Resolute annex libraries, i.e., Resolute annex clauses placed directly in an AADL package. The example shows a claim function called SCSReq1 that represents a requirement and consists of two verification actions that must be satisfied for the requirement to be satisfied (expressed by **and**). The example also shows a computational function that calculates the sum of budgets of all direct subcomponents. Finally, the example shows a global constant definition that can be referenced in a Resolute library or Resolute subclause. We used it to specify the maximum weight, which the verification actions will compare the total against.

```
package BudgetCase
public

annex Resolute {**
    MaximumWeight : real = 1.2kg

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

    AddBudgets(self: component) : real =
        sum({WeightBudget(t) for (t: subcomponents(self))})
```

The claim functions representing requirements are then associated with component types or implementations by prove statements declared in Resolute annex subclauses. The example shows the prove statement for SCSReq1 with the component itself passed in as parameter.

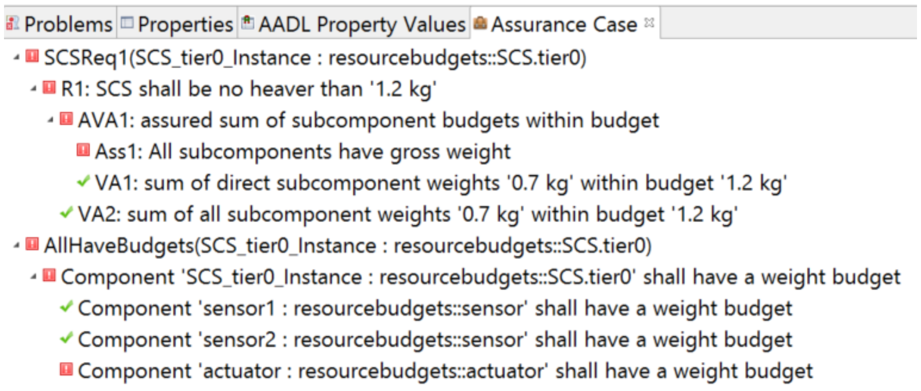
```
system implementation SCS.Phys
    subcomponents
        sensor1: device sensor;
        sensor2: device sensor;
        actuator: device actuator;
    annex Resolute {**
        prove (SCSReq1(this))
```

Users 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.


Package Public wbs::main_public
System wbs_system
System Impl wbs_system.generic
System Impl wbs_system.no_redundancy
System Impl wbs_system.redundant
System Impl wbs_system.push_buttons_sampling
Debugging: Serialize as 'mypack.aadl' text
Save As XMI
Instantiate System
Open Package Diagram
Open Classifier Diagram
Resolute
AGREE
AADL to Sexp

The verification results are then displayed in a view labelled *assurance case.* For each claim function invoked by a **prove** statement Resolute evaluates any claim function called by the claim function expression.

Note: Currently, each **prove** statement is shown at the top-level of the Assurance Case view. This is the case for different **prove** statements of the top-level component as well as any subcomponent that has **prove** statements. In the future we may want to show the proofs nested according to the architecture hierarchy.



```
SCSReq1(SCS_tier0_Instance : resourcebudgets::SCS.tier0)
  R1: SCS shall be no heaver than '1.2 kg'
    AVA1: assured sum of subcomponent budgets within budget
      Ass1: All subcomponents have gross weight
      ✓ VA1: sum of direct subcomponent weights '0.7 kg' within budget '1.2 kg'
    ✓ VA2: sum of all subcomponent weights '0.7 kg' within budget '1.2 kg'
AllHaveBudgets(SCS_tier0_Instance : resourcebudgets::SCS.tier0)
  Component 'SCS_tier0_Instance : resourcebudgets::SCS.tier0' shall have a weight budget
    ✓ Component 'sensor1 : resourcebudgets::sensor' shall have a weight budget
    ✓ Component 'sensor2 : resourcebudgets::sensor' shall have a weight budget
    Component 'actuator : resourcebudgets::actuator' shall have a weight budget
```

Note: throughout the document we use an example of resource budget analysis, analysis of weight in particular. A reference to the full example is given at the end of the document.

Claim Functions

Claim functions are defined in Resolute annex libraries. Claim functions have a name, zero or more parameters, a description and a predicate expression. The description can be a combination of strings and parameter references. The claim function has a return type of Boolean (true to represent success and false to represent failure). The parameters typically represent the model element to which the claim function applies and values used in the predicate expression to determine whether the predicate is satisfied. The description explains the role of the claim function and is displayed as part of the result in the Assurance Case view. The evaluation of the predicate determines the success (true return value) or failure (false return value) of the claim function.

Claim functions are used to represent requirements, verification actions and assumptions. There is currently no syntactic distinction in their use. Below we describe conventions that can be followed to distinguish the different uses.

The syntax of a claim function is as follows:

```
Claim_Function ::=
name (  ( parameter ( , parameter )* )? ) <=
** description ** claim_function_expression
parameter ::= Name : type
description ::= “text” ( parameter_reference | “text” )*
```

The parameter declaration consists of a name and a type (see later section on the 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 (**let** statements). If their values are numerical with a unit, users 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**, **forall**, implies (**=>**)) to represent a predicate.

A Resolute annex library is declared directly in a package through the **annex** Resolute **{**** <library content> ****};** statement. Such a statement can be placed in a package by itself, combined with library declarations for other annexes, or combined with classifier declarations in the same package. A package cannot contain multiple Resolute annex library declarations, i.e., each library has to be placed in a separate package.

Note: Resolute currently assumes a global name space for the names of claim functions and computational functions. Therefore, their names have to be globally unique. They can be referenced in **prove** statements or in other claim functions or computational functions without qualification by a package name.

Application of Claim Functions

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

When you have a system with subsystems, you can associate claim functions with each component implementation in the system hierarchy. The verification of this claim will then be performed for each instance of the component implementations with a **Prove** statement. Typically, you have **prove** statements for the top-level system that involve verifying the system across multiple layers, e.g., performing resource budget analysis across the whole system. You may also perform compositional verification by verifying a component with respect to its direct subcomponents. In this case, you place a **prove** statement at each layer, i.e., with each component implementation of components involved in the system. In the case of resource budget analysis, we illustrate that by analyzing budgets of direct subcomponents against the specified budget limit for each layer of the architecture.

Note: Currently a **prove** statement cannot be declared for a component type. A declaration with the component type would mean that it applies to all instances of all implementations.

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 )* )?  )  )
```

The parameter values can be integer, real, string, Boolean literals, model elements, and references to global constants, or 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.

Note: the keyword **this** can only be used in the **prove** statement – not in verification action expressions. This means that **this** must be passed to a claim function for it to know what model element it operates on – unless we have a global claim functions (see later).

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

Users can also identify a subcomponent of **this**, i.e., they can associate the verification action with a component down a path of the architecture hierarchy. This allows users to specify a verification action for a specific component instance. The example below shows how a **prove** is applied to a subcomponent called subsystem1.

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

Note: 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 only associate **prove** statements with a path if the prove is intended for that particular instance of a subcomponent.

Uses of Claim Functions

In this section we describe different uses of claim functions. These uses are conventions that are not enforced by the Resolute compiler.

The compiler does enforce that claim functions can only be invoked in prove statements, and as operands in **and**, **or**¸ **=>** (implies), and **forall** operations, and cannot be invoked in computational functions.

Claim Function as Requirement to be Satisfied

The claim function invoked by the **prove** statement typically represents a requirement on the component. Therefore, the description should describe the requirement and possibly include a requirement identifier. We reference a Resolute global constant *MaximumWeight* to indicate the desired limit. This allows us to change the limit by editing one location.

The expression specifies the verification actions whose results are used as evidence that the requirement has been met.

```
MaximumWeight : real = 1.2kg

-- requirement expressed in terms of the Maximum weight constant
SCSReq1(self : component) <=
**  "R1: SCS shall be no heaver than " MaximumWeight%kg  **
 SCSReq1AssureSubcomponentTotals(self, MaximumWeight)
 and SCSReq1AssureRecursivetotals(self, MaximumWeight)
```

If the weight limit is already specified as a property in the AADL model then we can reference it to be supplied as value to the verification action as shown below. In addition, we can verify that the property and the global constant have the same value.

```
SCSReq1VA1(this, property(self,SEI::WeightLimit))
```

The requirement is associated with an implementation of the system called *SCS* as

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

A requirement may be specified vaguely such that it cannot be verified. In this case we may need to refine the requirement into subrequirements that are verifiable. This is the case the claim function expression refers to claim functions that represent the subrequirements.

The claim function expression in a “requirement” claim function may contain an **and,** an **or**, a **forall,** or an **implies**.

- The **and** and **forall** indicate that all operands are evaluated, and then the claim is true if all operands return **true**.
- The **or** indicates that alternative sets of verification actions. The operands are evaluated in order and the first returning **true** determines true for the claim.
- The **=>** (implies) indicates that the right hand side is evaluated if the left-hand side returns true. The value of the claim is that of the right hand side, or **true** if the left hand side evaluates to **false**.

Note: Resolute currently allows computational expressions to be mixed with subclaim functions in a claim function expression. Unfortunately, the results of their evaluation are not shown in the assurance case view. This leads to the scenario where all subclaims are shown as satisfied (**true**), but the claim itself is shown as failed because the computational constraint expression happens to evaluate to **false**. Therefore, we recommend for users not to mix subclaims and computational expressions.

Claim Functions as Verification Actions

Claim functions also represent verification actions, whose execution produces a verification result as evidence. In this case the expression represents the predicate to be evaluated in order for the verification action to pass. One of the elements of the predicate typically is a constraint value to be met and the other is a computational function call or reference to a computational result to compare. In the example below the verification action computes the sum of the subcomponent budgets and compares them to the maximum. AddBudgets is a computational function that returns a **real** value (see later).

Note: The identifier VA1 is added into the description text by convention.

```
SCSReq1VA1VerifySubcomponentTotals(self : component, max :real) <=
** "VA1: sum of direct subcomponent weights " actuals%kg " within budget "  max%kg **
let actuals : real = AddSubcomponentWeightBudgets(self);
(actuals <= max)
```

If we want to produce a special error message for a failing verification condition we can use an implies (**=>**) operation. We negate the predicate of the example above and then generate a fail exception message using the **fail** construct. It gets called if the predicate does not hold. If the predicate does hold, i.e., the left hand side evaluates to false due to the **not**, the whole implies expression returns **true** – as desired.

```
SCSReq1VA1VerifySubcomponentTotals(self : component, max :real) <=
** "VA1: sum of direct subcomponent weights " actuals%kg " within budget "  max%kg **
let actuals : real = AddSubcomponentWeightBudgets(self);
-- checking for over budget results in a fail exception. Otherwise return true
(actuals > max) => fail ** self " weight sum " actuals%kg " over budget " max%kg **
```

The verification actions are associated with the requirement calling them in the claim function expression as shown earlier.

Verification Assumptions

Verification actions may make assumptions. We can specify those assumptions as computational functions or as claim functions. We can support the following scenarios:

1) The assumption is evaluated and if false the verification action is skipped. For this purpose we define a computational function that returns a Boolean. The example shows such a function that determines whether all subcomponents have a weight property. It is then used on the left of an implies (**=>**) with the verification action on the right.

```
-- assure all direct subcomponents have a weight budget
SubcomponentsHaveWeightBudget(self:component): bool =
    forall (sub: subcomponents(self)). has_property(sub,SEI::GrossWeight)
```

2) The assumption is evaluated. If it fails we report the failure and consider the verification action failed. In this case we add a **fail** declaration that throws a fail exception. This fail exception is caught by the enclosing claim function and recorded as a Fail result. The verification action is not executed. If the assumption is satisfied the verification action is executed. In the example below we also calculate the percentage of subcomponents with the weight property when reporting

the failure.

```
-- assure all direct subcomponents have a weight budget
SubcomponentsHaveWeightBudgetFail(self:component): bool =
    let ratio : int = SubcomponentWeightBudgetCoveragePercent(self);
 not(forall (sub: subcomponents(self)). has_property(sub,SEI::GrossWeight))
   => fail ** "Percentage of subcomponents with weight " ratio "%" **
```

3) The assumption is evaluated and the verification action is evaluated. We report a failure if either the assumption or the verification action fails. In this case we find out what the result of the verification action is even though the assumption is not met. In the budget example, we can find out that we are over the limit even though only 50% of the subcomponents have a weight property instead of forcing the user to complete the specification and then find out that the requirement has not been met.
   In this case we define the assumption as a claim function and combine it with the verification action in an **and** expression of the enclosing claim function.

```
SCSReq1AssureSubcomponentTotals(self : component, max :real) <=
** "AVA1: assured sum of subcomponent budgets within budget" **
-- we only evaluate the weight total if subcomponents have weight budget
SCSReq1VA1SubcomponentsHaveWeight(self) and SCSReq1VA1VerifySubcomponentTotals(self, max)

SCSReq1VA1VerifySubcomponentTotals(self : component, max :real) <=
** "VA1: sum of direct subcomponent weights " actuals%kg " within budget "  max%kg **
let actuals : real = AddSubcomponentWeightBudgets(self);
-- checking for over budget results in a fail exception. Otherwise return true
(actuals > max) => fail ** self " weight sum " actuals%kg " over budget " max%kg **

SCSReq1VA1SubcomponentsHaveWeight(self : component) <=
** "Ass1: All subcomponents have gross weight" **
let ratio : int = SubcomponentWeightBudgetCoveragePercent(self);
not(forall (sub: subcomponents(self)). has_property(sub,SEI::GrossWeight))
=> fail ** "Percentage of subcomponents with weight " ratio "%" **
```

Global Claim Functions

Claim functions can be defined without parameter. In this case the claim function has to first query the instance model for objects and then apply a claim function to each of the element of the query. The query is accomplished by identifying a collection of model elements and then applying a claim function to each element of the collection using **forall** or **exists**. Model element collections can be identified by category, e.g., **threads** for all threads in the instance model, or by set constructors such as **instances**(classifier) which returns all component instance of the specified classifier. Collections and their operations are discussed in detail later.

The example shows a global constraint that all threads must have a period property value.

```
SystemWideReq1() <= ** "All threads have a period" **
forall (t: thread). HasPeriod(t)

HasPeriod(t : thread) <= ** "Thread " t " has a period" **
has_property(t,Timing_Properties::Period)
```

Global claim functions could be included as **prove** statements on any model element, since they query the whole model independent of a specific model element. However, it is recommended that they be placed in the top level component implementation to which the *Resolute* command is applied – otherwise they may be invoked multiple times.

Computational Functions and Constants (Final Variables)

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 may be preceded by a local constant declaration. Note that the expression language includes conditional and loop operations (see below).

```
Computational function::= functionname ( parameter ( , parameter )* ) : return_type =
computationalexpression
```

- 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.

Global Constants

Global constants represent parameters to the verification whose value is set once and can be used in any computational expression, including parameter to a claim function call. 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::= constantname : type = computationalexpression
```

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

Local Constants

Resolute also supports pre-computation of local constants. They 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 constantname : type = computationalexpression ;`
- The scope of a local constant is the expression, i.e., they can only be referenced from within the succeeding expression.
- Local constants are used to pre-compute values that may be referenced multiple times in the succeeding expression.

Predicate Expressions and Computational Expressions

A constraint expression results in a Boolean value. Computational expressions are used in computational functions and must be of the specified type.

Predicate expressions support the following operators in increasing precedence order:

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

- Implies: a **=>** b
- Disjunction: a **or** b
- Conjunction: a **and** b
- negation: **not** a
- Quantified logical expressions: (**forall** | **exists**) **(** variablename **:** collectionconstructor **) .** logicalexpression

Computational expression 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.
  - **+** | **-**
  - ***** | **/**
  - Negation: **-** a
- Precedence brackets: **(** a **)**

Type related operators:

- type test (a **instanceof** type)
- Type cast: **(** type **)** a

Atomic expressions can be used as operands of above list of operators:

- Base type values: integer value, real value, string value, 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 or local constant reference, variable reference by their identifier
- Computational function invocation: function **(** (parameter_value (**,** parameter_value)*)? **)**
- Conditional value: **if** condition **then** expression **else** expression
- Qualified classifier or property definition: (ID **::**)* ID (**.** ID)?
  - Classifier used only as 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

Exception operator – the equivalent of an exception throw with the enclosing claim function representing an implicit catch.

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

Collection related operators:

- Basic collection: **{** expression (**,** expression)* **}** | **{ }**
- Filtered collection: **{** filteredelement **for** (**(**elementname **:** collectionconstructor **)**)+ **|** filterexpression

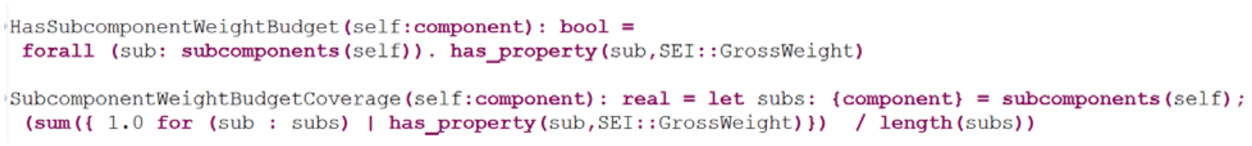
- note: filteredelement refers to one of the set element names
- note: filterexpression is of type Boolean

collectionconstructor ::= Basic collection | Filtered collection | AADL model element type | reference to global or local constants holding collections | computational function invocation returning a collection

Note: Function invocations returning a collection can be a user defined computational function or a built-in function (see below). 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 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 collection of real values of value 1.0 for each sub component that satisfies the **has_property** constraint function and then perform the summation of the resulting **real** 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): real = let subs: {component} = subcomponents(self);
 (sum({ 1.0 for (sub : subs) | has_property(sub,SEI::GrossWeight)})  / length(subs))
```

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

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

Use of **fail** expression: It can be used in any computational function and can be view 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, i.e., it is shown as a failure with the provided text explaining the failure.

Resolute Type System

type ::= collection type | base type | AADL model element type

collection type::= **{** type **}**
The collection concept allows multiple elements of the same value. In the *SubcomponentWeightCoverage* example it has multiple instances of the value 1.0 and each is counted in the summation.

Built-in Base Types

Base type:

- **int**
- **real**
- **string**
- **bool**
- **range**

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. For presentation of 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**.

AADL Model Element Types

AADL model element type 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_acces**
- **subprogram_group_access**
    - **provides_subprogram_group_access**
    - **requires_subprogram_group_access**

Note that 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 below).

Built-in Functions

Built-in functions on Collections

union(collection, collection) : collection - returns a collection that is the union of the two inputs.

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

length(collection): int - returns the size of the collection

member(element, collection): Boolean – returns true if element is a member of the collection

sum(numeric collection): numeric – calculates the sum of a collection of integers or a collection of real.

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

Built-in functions on any model element (of the instance model):

has_property (namedelement, property): boolean - the namedelement has the property.

property (namedelement, 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.

has_parent (namedelement): boolean - returns true if the component has an enclosing model element.

parent (namedelement): namedlement - returns the parent of the namedelement. The parent must exist.

name (namedelement): string - returns the name of the namedlement.

has_type (namedlement): 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 it is the type of the feature that is the connection end.

type (namedelement): Classifier - returns the classifier of a component, feature, or connection. In the case of a connection it is the type of the feature that is the connection end. The named element must have a type.

is_of_type (namedelement, classifier): boolean – true if the named element has the classifier or one of its type extension. The named element must have a type. The named element can be a component, feature, or connection instance. In the case of a connection it is the type of the feature that is the connection end.

has_member (component, string): boolean – true if the component has a member with the specified name (string). Members are features, subcomponents. Component can be a component instance or a component classifier.

- Note: feature instances representing feature groups can have feature instances as members, but are not handled by this function.

source (connection): connectionendpoint – returns the component or feature instance that is the source of the connection instance

destination (connection): connectionendpoint – returns the component or feature instance that is the destination of the connection instance

direction (feature): string – returns the direction of a feature instance as string (in, out, inout/in_out?)

- Note: we want to replace this by **incoming**(feature) and **outgoing**(feature)

is_event_port (feature): boolean – true if feature instance is an event port

- Note: we are missing test for data port, event data port.

is_bound_to (component, bindingtarget): Boolean – true if the component instance is bound to the binding target.

- Note: connection bindings are not handled.

Model Element Collections

Note that Resolute operates on the instance model, this the collections are of instance model elements.

features (namedelement): {feature} - returns a collection containing the features of the namedelement.

subcomponents (namedelement): {component} - returns a collection containing the subcomponents (component instances) of the namedelement.

connections (namedelement): {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 (componentclassifier): {component} – returns the collection of instances in the instance model for a given component classifier

instance (componentclassifier): component – returns the component instance for a given component classifier. The method assumes that there is only one instance.

External Functions

analysis (function: string, args): boolean – 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.

Error Model Functions

propagate_error (namedelement, errortype: string): boolean – true if the component or feature instance propagates the error type.

error_state_reachable (component, state: string): boolean – true if the error state of the component instance is reachable.

Pre-declared Resolute Computational Function Library

Binding Related Predicate Functions

bound (component, bindingtarget): Boolean – true if the component instance is bound to the binding target by actual processor, memory, or connection binding. Note: connection instance should be handled as well.

processor_bound (component, bindingtarget): Boolean – true if the component instance is bound to the binding target by actual processor binding.

memory_bound (component, bindingtarget): Boolean – true if the component instance is bound to the binding target by actual memory binding.

connection_bound (component, bindingtarget): Boolean – true if the component instance is bound to the binding target by actual connection binding. Note: should handle connection instances.

Connection Related Functions

source_component (connection): component – returns the component that is the source of the connection instance. This is the component that contains the feature instance as connection endpoint.

destination_component (connection): component – returns the component that is the destination of the connection instance. This is the component that contains the feature instance as connection endpoint.

is_port_connection (connection): boolean – true if connection is a connection between ports

is_data_port_connection (connection): boolean – true if one of the connection endpoints is a data port. Note: should be determined by the destination.

is_event_port_connection (connection): boolean – true if one of the connection endpoints is a event port. Note: should be determined by the destination.

is_event_data_port_connection (connection): boolean – true if one of the connection endpoints is a event data port. Note: should be determined by the destination.

is_data_access_connection (connection): boolean – true if one of the connection endpoints is a data access feature. Note: should be determined by the destination.

Model Element Containment

contained (named element, container component): Boolean – true if named element is contained in 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.

Handling of Feature Groups

Note: 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.

Resolute Examples

Weight Budget Assurance Case

This is the example used throughout the document. It can be found on Github at https://github.com/osate/examples under the project core-examples in folder resolute as the files resourcebudgets.aadl for the system model and BudgetCase.aadl for the Resolute library.

Debugging Models with Resolute

Get a model element trace in the assurance case view:

```
print_aadl(a : aadl) <=
  ** a **
  true

print_set(s : {aadl}) <=
  ** s **
  true
```

Reachable Collections of Model Elements

This is a snippet from the Smaccmcopter example on https://github.com/smaccm/smaccm/tree/master/models.

```
reach(c : component) : {component} =
  recursive_reach({c})

recursive_reach(curr : {component}) : {component} =
  let next : {component} = union(curr, next_reach(curr));
  if next = curr then
    curr
  else
    recursive_reach(next)

next_reach(curr : {component}) : {component} =
  {y for (x : curr) (y : reachable_components(x)) | not is_decrypt(x)}

reachable_components(comp : component) : {component} =
  {c for (conn : connections(comp))
           (c : reachable_components_via_connection(comp, conn))}
```

```
-- What components (either a single one or none) can 'comp' reach directly via 'conn'
--
-- This is complicated due to data access connections which seem to ignore
-- normal aadl directionality, and instead use an access rights property
reachable_components_via_connection(comp : component, conn : connection) : {component}
=
  -- a direct port connection
  if is_port_connection(conn) then
    if source_component(conn) = comp then {destination_component(conn)} else {}
  -- a component reading from 'comp' as a data component via read access on 'conn'
  else if comp instanceof data then
    if comp = source(conn) and has_read_access(destination(conn)) then
      {destination_component(conn)}
    else if comp = destination(conn) and has_read_access(source(conn)) then
      {source_component(conn)}
    else
      {}
  -- 'comp' writing to a data component via write access on 'conn'
  else if destination(conn) instanceof data and has_write_access(source(conn)) then
    {destination_component(conn)}
  else if source(conn) instanceof data and has_write_access(destination(conn)) then
    {source_component(conn)}
  -- Other connections unsupported at this time
  else
    {}
```