rAlf Reference Manual

Table of Contents

Introduction

Purpose

The purpose of this document is to give a summary of the rAlf action language. It is not a detailed semantics specification; however, in some points some of the semantical details are described to provide better understanding of the rAlf syntax.

rAlf Concepts

The rAlf is used to define the semantics for the processing that occurs in an action. An action can be associated with the following modeled elements:

- States entry and exit action
- Transition effect
- Static and non-static operations

The rAlf provides different types of actions:

- data access
- signal sending
- operation call
- object model manipulation (create, link, unlink, delete objects)
- accessing object model (fetching instances, navigating associations)

- control structures

In a xUML-RT model, unlike conventional programming, there is no concept of a "main" function or routine where execution starts. Rather, the models are executed in the context of a number of interacting finite state machines. Any state machine, upon receipt of a signal (from another state machine or from outside the system) may respond by changing state. On entry or exit to or from the new state, a block of processing (an "action") is performed. This processing is sequential that is only one state machine action executed at the same time as processing associated with another state machine.

rAlf is used to define the processing executed during the action. The execution rules are as follows:

- Execution commences at the first statement in the action and proceeds sequentially through the succeeding lines as directed by any control logic structures.
- Execution of the action terminates when the last statement is completed. These rules also apply to actions defined for static and non-static operations.

Intended Audience

This manual is written for modelers and software engineers who use a process and set of modeling tools that support the creation, simulation, and translation of xUML-RT models. Specifically, guidance is provided for the correct specification of actions formulated in the syntax of the rAlf.

Structure

Overall Structure

An action consists of a number of statements. Each statement can be either a simple statement (such as an access to the attributes of a class) or a control logic structure (such as an `if` construct). rAlf statements are terminated by a semi-colon except `if` , `for`, `while` control constructs which are described later.

Comments

Comments may be inserted by the use of the `//` characters at any point in the line or /* */ for multi-line comments.

Names and Keywords

rAlf statements are composed of:

- keywords (see Keywords)
- logical and arithmetic operations
- names of modeled elements (classes, attributes, data types, association names and roles, signal name)
- local variables

Keywords and names (local variables, class attributes, association names and signal names) are case sensitive.

Names can be unqualified, qualified or fully qualified. Unqualified names are the ones which are not using namespaces in the name. For example a `C` class has a signal `S`. From an operation of class `C` one can write `send new S() to this;` which refers `S` with an unqualified name in the namespace of `C`. Qualified names are the ones containing namespaces separated with the namespace separator (‘`::`’), while the fully qualified name is a qualified name whose first element is the root named element of the model. The behaviors (Operations, state actions etc.) always have a current scope for the namespaces which controls the elements reachable with unqualified or qualified names. The current scope for the resolution of a reference to a name is the specific innermost namespace in which that named reference lexically appears (e.g the class for an operation or a state entry action).
A name is said to be visible in the current scope if a named element with the given name is a member of the current scope namespace or is visible in the namespace immediately enclosing the current scope (if any). Such an element may be referenced using an unqualified name. Otherwise the first name listed in a qualified name must be visible as an unqualified name in the current scope in which the qualified name occurs. Each succeeding name must be the name of a visible member of the preceding namespace.

Example

If the model is the following: package P1 including package P2 including Class A and B. A has a signal S and an operation OpA, B has a static operation OpB.
From OpA which have class A the innermost namespace the S is reachable with the following ways:

1. S (unqualified name)
2. A::S (qualified name using the enclosing namespace P2, A is unqualified name in this case)
3. P1::P2::A::S (fully qualified name)

From OpA which have class A the innermost namespace the OpB is reachable with the following ways:

1. B::OpB() (qualified name using the enclosing namespace P2, B is unqualified name in this case)
2. P1::P2::B::OpB() (fully qualified name)

White Space

White space (spaces and tabs) may be inserted at any point in an rAlf statements.

Keywords

The following enumerates the list of reserved words:

any	in	this
break	instances	to

delete	link	true
do	new	unlink
else	null	while
false	return	
for	sigdata	
if	send	

Data Items

Data Items in Actions

The rAlf expression of an action has access to and can produce certain data items. The following data items are available to be read at the start of and throughout an action:

- literals
- values of attributes of classes
- operation parameters
- signal parameters
- local variables (created by statements within the action)

Finally, the Object reference `this` may be used to refer to the currently executing instance in a state machine, to an instance in a non-static operation.

Data Types in Model Elements

All data items referenced or produced by an action must have a data type. The following data types are defined for class attributes, signal parameters:

- UML Primitive Types (Integer, String, Real, Boolean)
- Collection (set, sequence) of the above data types

In addition to those types, local variables and operation parameters may be of Object reference and object reference collection type.

The following data types are defined for transition, entry and exit actions parameter:

- signal
- `any`

Type declaration

All data items need explicit type declaration in rAlf except the sigdata data item whose type is always derived from the structure of the state machine (see in Receiving Signal Data).

Null Reference

`null` represents an invalid object instance reference. This is a special value that indicates that the object reference is not pointing to any object. Any operation on an object reference with `null` value results in an execution error. `null` value is only valid for data items whose type is object reference.

Local Variables

Local variables can be of any of the data types listed in Data Types in Model Elements. Two of these types require special consideration:

- Object reference: The identification of an instance of a class.
- Collection: a collection can be a set or sequence in rAlf.

Because Object references and Object reference collections can be obtained by selection from existing instances, the following situations can arise:

- a local variable of type object reference may be `null`, and
- a local variable of type object reference collection may refer to an empty collection. This case can be detected by using the supplied collection operators.

Syntax

```
<data type> <local variable>
```

`<data type>` Name of the data type listed in Data Types *in Model Elements*.
`<Local variable>` Name of the local variable

Notes

Attributes cannot be of type Object reference or Object reference collection.
In any non-static action within the class , a special Object reference called `this` is always available. `this` is always defined as a handle to the instance of the class that is executing the current action. The value of `this` cannot be changed by an action. The Object reference `this` has no meaning inside of an action for a static operation. In any action within a non-static operation or derived attribute, `this` is always available. `this` is always defined as an Object reference to the instance of a class against which the operation is being executed, or the instance for which the attribute is being read.
The local variable declarations can be combined with an initial value assignment (see in Assignment of Variables).

Example

```
Integer i;
Dog d // declaring a Class reference to the Dog Class
Sequence<Real> real_sequence;
Set<Dog> dog_set;
String s;
Boolean b;
```

Variable Initialization

Some situations arise where a variable may possibly be declared without being assigned a value. An obvious situation where this occurs is when an instance of a class with attributes is created. Variables declared without an explicit initial value are given their *zero value*.

The *zero value* is:

- null for Object references,

- empty collection for Collections,
- 0 for numeric types,
- false the Boolean type, and
- "" (the empty string) for Strings.

Scoping

The scope of a variable is defined as the block of code in which the variable may be accessed. A block of code can be the entire rAlf for the given action, or it may be a `<statements>` block within a control logic structure.
Each control logic structure contains at least one new scope. All variables that were accessible in the scope containing the structure are also accessible in the block or blocks contained by the structure, essentially causing the contained scopes to inherit variables from the parent scope. Any variables declared within a given control logic block fall out of scope when execution exits the block. Control logic structures may contain multiple scopes, either by repeated nesting of new structures or by using the `else if` or `else` constructs in an `if` structure. When nesting of control logic is used, each new structure defines a new scope. In an if statement, each `else if` or `else` structure contained within the `if` block defines a new scope, and each new scope inherits the scope of the block containing the `if` statement.

Notes

A local variable always need to be declared. The scope is limited to the current block.
Local variables have a maximum scope of the entire rAlf for the current action.
Scope of the local variables declared in a `for` loop declaration e.g. `for(int i in Integers)` is the same as the variables declared inside the `for` loop main block so they are not available outside the for loop block. Variable declaration is not possible in `if` and `while` declarations.
The `filter` clause has a special variable that has scope limited to the `<filter expression>`.
The scope of `this` for an instance's action, operation, or attribute, is the entire rAlf for the current action.

Example

```
// begin action
// scope1 - global scope for this action. Variables declared here
// are accessible anywhere in this action.
Integer sum = this.lower + 10*this.higher;
if (sum > 100){
      // scope2 - Variables declared here are only accessible
      // within if statement.
      Integer diff=this.getDiff();
} // All variables declared in scope2 are not accessible
// after the end if.
Set<Employee> empls = Employee::instances();
for(Employee empl in empls){
      // scope3 - Variables declared here are only accessible within
      // for statement.
      String name = empl.name;
} // All variables declared in scope3 are not accessible after
// the end for.
// empl is not available in the scope containing the for statement
// (scope1).
if (sum <= 100){
      // scope4 - Variables declared here are accessible within this
      // if block and in scope4.1 and scope4.2.
      if (office.location == "Europe"){
            // scope4.1 - Variables declared here are only accessible
            // within this if block.
      }
      if (office.location == "America"){
            // scope4.2 - Variables declared here are only accessible
            // within this if block.
      }
}
// end action
```

Type Conversion

A strict data typing scheme for primitive types, excluding Real and Integer, is used in rAlf. Only Integers are allowed to be implicitly promoted to Real values.
Automatic upcast is available for object references, that is, it is allowed to assign an object reference value to a data item (local variable, attribute, operation parameter) which type is a superclass of the object reference.

Example

```
Integer i = 5;
Real r = i;
Vehicle vehicle;
Truck truck = new Truck(); //Truck is a subclass of Vehicle
v = truck; // automatic upcast
```

Control Structures

Empty statement

Syntax

```
;
```

Note

Any empty statement has no effect when it is executed.

Block construct

Syntax

```
{<statements>}
```

Note

The execution of a block statement consists simply the execution of the statements contained by the block. A block statement defines a new scope (see the scoping rules in Scoping).

Example

```
{
  a = F(1);
  b = G(2);
}
```

If Construct

Syntax

```
if (<boolean expression>) {
 <statements> //Executed if <boolean expression> is true
}
if (<boolean expression>) {
 <statements> //Executed if above boolean expression evaluates to true
else if (<boolean expression>){
 <statements> //Executed if above boolean expression evaluates to true
}
// and previous boolean expression is false
else {
 <statements> //Executed if both boolean expressions evaluate to false
}
```

`<boolean expression>` is an expression evaluating to `true` or `false`.
`<statements>` are zero or more rAlf statements.

Notes

The `if` construct may contain as many `else if` clauses as desired.
Only one `else` clause may be used, and it must appear at the end of the `if` construct.
`if` construct without block statement is not supported.

Example

The following example shows an `if/else if/else if` construct:

```
// Assign x with a different number for each name.
Integer t;
if (this.type == "DB"){ t = 5; }
else if (this.type == "Webshop"){ t = 10;}
else if (this.type == "Game"){ x = 30;}
else{
  t = -1;
}
```

This example shows nested `if` constructs:

```
this.destination = signal.destination;
if (sum <= 100){
      // scope4 - Variables declared here are accessible within this
      // if block and in scope4.1 and scope4.2.
      if (office.location == "Europe"){
            // scope4.1 - Variables declared here are only accessible
            // within this if block.
      }
      if (office.location == "America"){
            // scope4.2 - Variables declared here are only accessible
            // within this if block.
        }
}
```

For Loop

The *for* loop allows for the iteration over a set of values in a collection (set, sequence) or iterate over a range of values.

Syntax

There are two ways to use a for loop. The first way is to iterate over a collection:

```
for(<variable> in <collection>){
  <statements>
}
```

<variable> is a local variable referring to a single value.

<collection> is a collection of values. It can be a local variable, a return value of an operation call or a result of an Association Navigation.

The second way is to iterate over a range of values:

```
for (<initialization>; <termination>; <increment>) {
  <statements>
}
```

When using this version of the for statement, keep in mind that:

- The *<initialization>* expression initializes the loop; it's executed once, as the loop begins.
- When the *<termination>* expression evaluates to false, the loop terminates.
- The *<increment>* expression is invoked after each iteration through the loop; it is perfectly acceptable for this expression to increment or decrement a value.

Notes

The statements in the *for* construct are executed once against each value in *<collection>*. The order in which the particular value are processed is depending on the collection type. In case of a *sequence* the order is always the same as the order in the *sequence*. In case of *set* the order of the values during the iteration is undefined.
The *<variable>* can be declared in-place or before the for loop

Example

```
// children is a set of <Object references> of Child class
Set<Employee> empls = Employee::instances()
for(Employee empl in empls){
  send new Employee::travel() to empl;
}
// Iterate over the instances of D without creating a local variable
// to store the values.
for( D d in D::instances()){
  send new D::sig1() to d;
}
//Setting the attr1 of B instances associated to 'a' which ID is
//greater than 10.
```

```
for(B b in a->b.filter(b : b.ID>10)){
  b.attr1 = 15;
}
// Create 10 employees with IDs 0-9
Integer i;
for(i=0;i<10;i++){
  Employee e = new Employee(ID=>i);
}
```

While Loop

The `while` construct is used to sequentially execute the code it contains for as long as the condition is evaluated as `true` .

Syntax

```
while (<boolean expression>){
  <statements>
}
```

`<boolean expression>` is an expression evaluating to `true` or `false`
`<statements>` are zero or more rAlf statements.

Note

When the `while` loop is executed, the statements in the `while` construct are executed consecutively as long as the `<boolean expression>` evaluates to `true`.

Example

```
// Create 10 employees with IDs 1-10
Integer i = 1;
while (i <= 10){
      Employee e=new Employee(ID=>i);
      i = i + 1;
}
```

Do Loop

The `do` construct is used to sequentially execute the code it contains at least once and repeatedly as long as the condition is evaluated as `false`.

Syntax

```
do {
  <statements>
} while (<boolean expression>)
```

`<boolean expression>` is an expression evaluating to `true` or `false`
`<statements>` are zero or more rAlf statements.

Note

When the `while` loop is executed, the statements in the `while` construct are executed consecutively as long as the `<boolean expression>` evaluates to `true`.

Example

```
// Create 10 employees with IDs 1-10
Integer i = 1;
do{
      Employee e=new Employee(ID=>i);
      i = i + 1;
} while (i > 9)
```

Break

The *break* statement allows the early termination of both *for* and *while* loops. This can have some significant performance implications in the case of large loops that need not step through the entire iteration.

Note

The *break* statement only applies to the current *for* or *while* loop containing it. To break out of nested loops, the *break* must be repeated for each loop construct the user wishes to exit.

Example

```
// Create and relate a B to every A while CTL says to keep creating.
while ( Employee::stillNeeded()){
      isEnough = false;
      for(Asset a in aset){
            // Break if job title equal to "Engineer"
            if (a.job == "Engineer"){
                  isEnough = true;
                  break;
            }
            // Create and relate a new b to the given a.
            B b = new B();
            R1::link(B=>b,A=>a);
      }
      // If enough B was created break the loop
      if (isEnough){
            break;
      }
}
```

Nested Control Logic

Control logic may be nested to any depth.

Example

```
// Send a 'time for bed' event to all children 5 and under.
for(Employee e in Employee::instances()){
  if (e.age <= 30){
    if (e.salary > MAX_SALARY){
      send new Employee::decrease() to e;
      if (team.type ==”DB”){
        send new Employee::changePosition() to e;
      }
    }
```

```
   }
}
```

Class Manipulations

Creating Instances

Creation of an instance of a class is achieved by use of the *new* statement.

Syntax

```
<Object reference> = new <Class name>(<constructor parameters>);
```

Notes

The *<Object reference>* returned is the handle of the newly created instance of class. The handle can be used only to refer to an instance of class. The *<Object reference>* in the *new* statement cannot be *this*. All unconditional associations that involve the newly created instance should be satisfied before completing the action in which the instance is created. This can be done by directly relating the associated instance either in the action block or in the constructor of the newly created class.

Examples

```
Monitor monitor = new Monitor();
// Object creation with passing parameter to the constructor
Car c = new Car(type=>"VW");
new Person(); // no Object reference needed
```

Querying Instances

The instance query statement can be used to assign an instance or set of instances to either an Object reference or an Object reference set respectively. The instance query statement is querying the extent of a class. The extent of a class is the set of objects that currently exist at the specific execution locus (engine) at which the class extent expression is evaluated.

Syntax

```
<Object reference set> = <Class name>::instances();
<Object reference> = <Class name>::instances().one();
```

<Object reference set> is a set of handles for all selected instances of the class specified by *<Class name>*.
<Object reference> is an instance handle of the class specified by *<Class name>*.

Notes

The result of the instance querying (*<Object reference set>)* can be used as a usual set so all set operation (see in Collection Library) is valid for it.

Example

```
// Query an arbitrary instance of the Employees.
Employee e = Employee::instances().one();
// Query all instances.
Set<Employees> empl_set = Employee::instances();
// Query an instance of Employee started after 2010.
```

```
Employee empl =
Employee::instances().filter(e:e.startDate>2015).one();
// Query a set of instances of Employees work on product DB
Set<Employee> db_employees = Employee::instances().filter(e:
e.product==”DB”);
```

Making selections across associations is described in Association Navigation.

Writing Attributes

Attributes of instances may be set to specified values by use of the attribute assignment statement.

Syntax

```
<Object reference>.<attribute> = <expression>;
```

`<Object reference>` is a handle to an instance of a class.
`<attribute>` is the name of an attribute of the class.
`<expression>` is either a Boolean, String, or arithmetic expression.
The assign statement takes the value in `<expression>` and assigns it to the attribute `<attribute>` for the instance specified by `<Object reference>`.

Notes

The `<expression>` must evaluate to the data type of `<attribute>`, unless `<attribute>` is either a Real or an Integer, in which case `<expression>` can be either a Real or Integer value.
Derived attributes cannot be written to, only read. The value of a mathematically dependent attribute must be set within the action of the attribute. See **Error! Reference source not found.**.

Example I

```
// Account is a class with attributes branch, account_number,
// and balance. Assume new_account, this_branch, and initial_deposit
// are event supplemental data items.
// First, create a new instance.
ACCT my_account = new ACCT();
// Now set attribute values using the returned Object reference.
my_account.branch = sigdata.this_branch;
my_account.account_number = sigdata.new_account;
my_account.balance = sigdata.initial_deposit;

// Create and initialize a row.
Row row = new Row();
row.width=5;
```

Reading Attributes

A class attribute may be referenced in an attribute access expression.

Syntax

```
<Object reference>.<attribute>
```

`<Object reference>` is a handle to an instance of a class.
`<attribute>` is the name of an attribute of the class.

Notes

You may read the value of any attribute, including derived attributes.
`<Object reference>.<attribute>` is an expression and can be used in any rAlf construct specifying an expression.

Example

```
this.convertToFahrenheit(this.temperature);
product.name=this.name+"Prod";
```

Deleting Instances

Syntax

```
delete <Object reference>;
```

Notes

This statement deletes the instance specified by `<Object reference>`. When an instance of a class is deleted, it is no longer available on the locus where the class is defined, that is, it will not handle Signals or Operation invocations. However, sophisticated software architectures can be imagined which support the notion that the instance is kept for logging purposes although the defining domain cannot see it.
The instance deletion is not deleting the association links of the given instance automatically.
Calling the `delete` statement will implicitly call the `destroy` operation of the object so object destruction related activities can be added to this operation. The `destroy` operation must not have any parameters.

Example

```
// Delete every instance of DG with name equal to Fido.
for(Dog dog in Dog::instances().filter(d : d.name == "Fido")){
  Owner owner = dog->owner.one();
  R23::unlink(dog,owner);
  delete dog;
}
```

Associations

Creating an association link

Syntax

```
Boolean <association>::link(<source> => <source object reference>,
<target> => <target object reference>);
Boolean <association>::link(<target> => <target object reference>,
<source> => <source object reference>);
```

`<source object reference>` is the handle of the source class instance to be linked.

`<target object reference>` is the handle of the destination class instance to be linked.
`<source>` is the name of the association end on the source of the association.
`<target>` is the name of the association end on the target of the association.
`<association>` is the name of the association from the source class to the destination class.

Notes

The source and/or destination may be `this`.
The `link` operation returns `true` if a new association instance is created or `false` if no new association instance is created. New association instance is not created if the association ends are unique and an association instance has already been created between the source and target object references.

Examples

```
DP dp_inst = DP::instances().one();
D d_inst = D::instances().one();
R1::link(dp_inst,d_inst);
```

Deleting an association link

Syntax

```
<association>::unlink(<source> => <source object reference>, <target>
=> <destination object reference>);
<association>::unlink(<target> => <target object reference>, <source>
=> <source object reference>);
```

`<source object reference>` is the handle of the first class instance to be unlinked.
`<target object reference>` is the handle of the second class instance to be unlinked.
`<source>` is the name of the association end on the source of the association.
`<target>` is the name of the association end on the target of the association.
`<association>` is the association name from the source to the destination class.

Notes

An attempt to unlink two instances that are not related by the specified association is regarded as a run-time error.
The source and destination class object reference may be `this`.
If an association is deleted, instances of participating classes will not automatically be deleted to remain consistent with the Class Diagram. It is the responsibility of the modeler to ensure that the Class Diagram is respected.

Examples

```
// Unlink a_inst from b_inst
R1::unlink(a_end=>a_inst,b_end=>b_inst);
```

Association Navigation

Association navigation is the function whereby associations specified on the Class Diagram are read in order to determine the instance or set of instances that are related to an instance(s) of interest.

Syntax

```
<Object reference> = <start> -> <association end> -> ... <association
end>.one();
<Object reference collection> = <start> -> <association end> -> ...
<association end>;
<Object reference> = <start> -> <association end> -> ... <association
end>.filter(<Filter expression>).one();
<Object reference collection> = <start> -> <association end> -> ...
<association end>.filter(<Filter expression>);
```

`<start>` is an `<Object reference collection>` or `<Object reference>`.
`<association end>` can be either an association end name or an association class name.
`<Filter expression>` is a type of boolean expression used in the filter operation.

Notes

An *association link chain* is the sequence of `<association end>`'s used to specify the path from the starting instance or set of instances to the destination.
The association phrases in the association link chain must be given in the direction of navigation.
If the starting `<Object reference>` is null then the result will be considered a run time error.
The returned `<Object reference>` can be `null` or `<Object reference collection>` can be empty if any of the associations in the chain are conditional in the direction of navigation or the `filter` operation filters out all elements from the collection.
If the optional `filter` operation is added, the returned instance or collection of instances will meet the criteria of `<Filter expression>`. This implies that the Object reference or the Object reference collection may be empty if no instance(s) matched. If the returned Object reference is empty then the value is `null`.

Example

```
Cat cat = owner->cat.one();
Dog dog = owner->dog.one();
Dog dog = owner->dog.filter(d : d.name == "Fido").one();
Set<Dog> dog = owner->dogs.filter(d : d.color == "black");
```

Signals

Receiving Signal Data

The keyword `sigdata` is the name of a data structure containing all of the data items received with a signal.

Syntax

`sigdata.<data item>`
`<data item>` is the name of the data item.

Note

`sigdata.<data item>` is an `<expression>` and so can be used in any rAlf construct specifying an expression.

The type of the `sigdata` data structure is depending on the state machine structure. In case of a transition with one trigger the type of the `sigdata` data structure is always the signal triggering the given transition. In case of multiple triggers the type of the `sigdata` is the effective common ancestor of all signals triggering the given transition. If no common ancestor is found then the type will be `any`. The rule is similar for entry and exit actions but instead of triggers the incoming and outgoing transitions are taken into consideration respectively.

Example

```
Employee empl = Employee::instances().one();
empl.salary = sigdata.salary;
empl.level = sigdata.level;
```

Signal Sending

Syntax

```
send <signal reference> to <target>;
send new <signal>(<data item> => <expression>) to <target>;
```

`<signal reference>` is an instance of a Signal. The Signal instance can be created in an earlier statement of the action block or in-place of the Signal sending.
`<signal>` Name of the signal
`<data item>` is the name of a data item defined in the Signal definition.
`<expression>` is a String, arithmetic, Boolean, simple, or compound expression. The data type of the expression must match the data type defined for the given data item.
`<target>` is an Object reference.

Notes

The `to` clause provides all the information necessary to identify the destination of the signal. When a signal is sent its data items are copied into a new signal which will be received by the target instance. As a consequence it is possible to modify the local variable representing the signal, that change will not affect the already sent signals.

Examples

```
// Send signal to existing instance
Employee empl = this->employees.one();
send new Employee::completed() to empl;
```

Expressions

Simple Expressions

Simple expressions are single unary or binary operations. An expression is not a complete rAlf statement, but is evaluated as part of a full rAlf statement such as `if`, `filter` , etc. Logical binary operators `&&` and `||` are supported for both compound and simple expressions.

Syntax

```
<read value>
<unary operator> <read value>
```

```
<read value> <binary operator> <read value>
```

`<read value>` is a constant, a local variable, the attribute of a class, a data item received with a signal, an operation invocation.
`<unary operator>` is any unary operator appropriate for the data type to which the expression evaluates. For Boolean read values, the unary operator is `!`. For arithmetic read values, the unary operators are `+` and `-` .
`<binary operator>` is any binary operator appropriate for the data types to which the expressions evaluate. For Boolean read values, the binary operators are `&` and `|`. For arithmetic read values, the binary operators are `+` , `-` , `*` , `/` , and `%` . For Object reference and Object reference collection read values, the binary operators are `==` and `!=` . For String read values, the binary operator is `+`.

Example

```
! (Company::get_status())
x + y
this.surname + this.lastname
type == "DB"
data1 - data2
```

Increment and Decrement Expressions

An increment expression is one that uses the increment operator ++. A decrement expression is one that uses the decrement operator --. Either of these operators may be used in either a prefix form, in which the operator appears before the operand expression, or a postfix form, in which the operator appears after the operand expression.

Syntax

```
<left value><postfix operator>
<prefix operator><left value>
```

`<left value>` is a local variable, the attribute of a class. It must have type `Integer`.
`<pre/postfix operator>` An increment expression is one that uses the increment operator `++`. A decrement expression is one that uses the decrement operator `--`.

Note

The effect of an increment or decrement expression is to increment (++) or decrement (--) the value of its operand and then reassigns the result to the operand. If the operator is used as a postfix, then the value of the expression is the value of its operand before it is reassigned. If the operator is used as a prefix, the value of the expression is the values of its operand after it is reassigned.
For example, if the local variable has the value 5, then both `a++` and `++a` assign the value 6 to a. However, the value of the expression `a++` itself is 5, while the value of `++a` is 6.

Examples

```
count++
size--
total[i]++
++count
--numberWaiting[queueIndex]
```

Compound Expressions

Compound expressions can be used to combine simple and increment and decrement expressions, allowing for multiple tests and more complex assignment arithmetic. Logical binary operators `&&` and `||` are supported for both compound and simple expressions.

Syntax

```
<operator> <expression>
<read value> <operator> <expression>
<expression> <operator> <read value>
<expression> <operator> <expression>
```

`<expression>` is a simple or a compound expression.
`<operator>` is any operator appropriate for the data types to which the expressions evaluate.
`<read value>` is a constant, a local variable, the attribute of a class, a data item received from a signal, an operation invocation.

Notes

The modeler can depend on the following rules regarding the order of evaluation of expressions:

- Parentheses can be used to override all other ordering rules.
- Standard mathematical precedence governs the order of evaluation for all mathematical operations (see the full set of precedence in the Table below).
- Order of evaluation of the subexpressions within any expression is unspecified. It is a good practice to combine only side-effect free expressions in a compound expression. There are exceptions to this rule for the &&, || operators which are noted in Boolean Expressions.

Operator	Description	Level	Associativity

`[]`
`.`
`()`
`++`
`--`
	access sequence element
	access attribute
	invoke a method
	post-increment
	post-decrement
		1	left to right

`++`
`--`
`+`
`-`
`!`
	pre-increment
	pre-decrement
	unary plus
	unary minus
	logical NOT
		2	right to left

`()`
`new`
	cast
	object creation
		3	right to left

`*`
`/`
`%`
	multiplicative	4	left to right

`+ -`	additive	5	left to right

`+`	String concatenation		
`<<` `>>` `>>>`	shift	6	left to right
`<` `<=` `>` `>=` `instanceof`	relational type comparison	7	left to right
`==` `!=`	equality	8	left to right
`&`	bitwise AND	9	left to right
`^`	bitwise XOR	10	left to right
`|`	bitwise OR	11	left to right
`&&`	conditional AND	12	left to right
`||`	conditional OR	13	left to right
`?:`	conditional	14	right to left
`=` `+=` `-=` `*=` `/=` `%=` `&=` `^=` `|=` `<<=` `>>=` `>>>=`	assignment	15	right to left

Examples

```
// examples of compound expressions:
!(motor.isOn() || light.off)
2 * (x + y) + Class1::remaining()
(a + b) / (c – d)

// examples of rAlf statements using
// compound expressions:
if ((I == 1) && (name == “DB”)){
  Real x = 0.5 * (y + z);
}

x = x * ((x + 1) / (x + 2));
```

Arithmetic Expressions

Arithmetic expressions are defined for Real and Integer data types only. These data types may be mixed for any given expression. Multiplicative operators are `*` , `/` , and `%` . Additive operators are `+` and –. Parentheses may be used to force precedence in arithmetic expressions.

Syntax

```
<unary arithmetic operator> <expression>
<expression> <binary arithmetic operator> <expression>
```

`<expression>` is any of the following that evaluates to a Real or Integer value: numeric constant, local variable, attribute of a class, simple expression, compound expression, operation invocation, bridge invocation, function invocation, or supplemental data item received from an event.
`<unary arithmetic operator>` is `+` or -.
`<binary arithmetic operator>` is `+` , `-` , `*` , `/` , or `%` (remainder from arithmetic division, valid only for Integer types).

Note

If any data item in the expression is Real, the expression will evaluate to a data type of Real. The `%` operator is valid only between two Integers.

Examples

```
-27
2 + 2
(x + y) / 2
(5/9)*(fahrenheit-32)
(plane.offset + Latitude::getLatitude())
```

Boolean Expressions

A boolean expression is any expression that evaluates to either a `true` or `false` value. boolean expressions are often used for comparison in statements like `if` and `while` , and also in `filter` clauses. Although boolean expressions usually contain other expression types (such as arithmetic or String expressions), they can also be used to compare Object references. There is also one unary operator, `!` , which can be used to logically negate a boolean expression.

Syntax

```
! <boolean expression>
<expression> <Boolean operator> <expression>
<handle> <Boolean operator> <handle>
```

`<expression>` is any expression, simple or compound. Both expressions must evaluate to the same type, either Boolean, arithmetic, or String.
`<boolean expression>` is a simple or compound expression that evaluates to a Boolean value.
`<Boolean operator>` is a logical operator.
`<handle>` an Object reference.

Note

The left and right values of a binary boolean expression must evaluate to the same data type, with the exception of Integer and Real.
A conditional logical expression is a binary expression using one of the conditional logical operators && (conditional-and) and || (conditional-or). The evaluation of their second operand expression is conditioned on the result of evaluating the first expression. In the case of the && operator, the second operand is evaluated only if the value of the first operand is true. In the case of the || operator, the second operand is evaluated only if the value of the first operand is false.

Examples

```
x == 1
id != "abc"
thi.error() || flag
(account.balance == 0.00) && (
last_pay_month=="October"))
```

Logical Operator	***Meaning***	***Valid Data Types***
==	equals	Integer, Real, Boolean, SString, object reference
!=	does not equal	Integer, Real, Boolean, String, object reference
<	less than	Integer, Real
>	greater than	Integer, Real
<=	less than or equal to	Integer, Real
>=	greater than or equal to	Integer, Real
&	bitwise and	Integer
^	bitwise XOR	Integer
|	bitwise or	Integer
&&	conditional and	Boolean
||	inclusive conditional or	Boolean
!	logical negation	Boolean

String Expressions

A String expression is any expression that evaluates to a String value. String expressions can be either a simple String or a concatenation of one or more simple Strings.

Syntax

```
<simple string>
<simple string> + ... + <simple string>;
```

`<simple string>` is any of the following that evaluates to a String value: String constant, local variable, attribute of a class, operation invocation, or a data item received from a signal.

Examples

```
"Hello"+", world!"
this.firstname + " " + this.lastname
```

Filter Expressions

A `filter` expression is a special type of boolean expression used in a `filter operation`.

Syntax

```
<filter variable> : <filter expression>
```

`<filter variable>` *name of the filter variable used in the filter expression*
`<filter expression>` *boolean expression using the* `<filter variable>`

Note

The type of this filter variable is always defined by the element on which the filter operation is called. In case of an Association Navigation (see in Association Navigation) the type of the filter variable is an object reference and it can be used to access the attributes of the given object instance. The `filter` expression must evaluate to a Boolean value.
It is not allowed to call operation on the object reference pointed by the <filter variable> .

Examples

```
Employee bob_employee = employees.filter(e : e.name == "Bob").one();
Set<Accounts> ok_accounts = accounts.filter(acc : (acc.status == "OK")
&& (acc.balance > (min_bal+200)));
// Use filter clause to find the CTO.
Employee cto = Employee::instances().filter(e : e.level == 10).one();
// Use filter clause to count the number of "Alice"
// in a String sequence
Integer alice_count = Strings.filter(s : s == "Alice").size();
```

Assignment of Variables

Syntax

```
<Local var> = <expression>;
<object reference var> = <object reference expression>;
<out/inout parameter> = <expression>;
```

`<expression>` is an Boolean, arithmetic, String or collection (Set, Sequence) expression.
The type of the expression must be compatible with the left hand side of the assignment.
`<object reference expression>` is an expression evaluating to an object reference.

`<Local var>` is a Boolean, Real, Integer, String or collection (Set, Sequence) typed local variable, or an attribute of a class instance.
`<object reference var>` is an object reference typed local variable or operation parameter.
`<out/inout parameter>` is an out or inout operation parameter.

Notes

Real variables can be assigned Integer values.
The assignment of variables is in itself an expression which evaluates to the value of the left-hand side of the expression. As a consequence of this rule it is possible to assign value to multiple variables in one assignment statement.
The assignments are value based. The result of a collection assignment is a new collection, that is, modifying the new collection does not affect the original collection except for collection of object references. In case of object reference only the object reference is assigned to the new value (no new object is created). Therefore, modifying the newly assigned variable, for example changing the value of the object or deleting the object, affects the original variable as well.

Examples

```
Integer x = 1;
pass = true;
name = "DB";
Sequence<Integer> int_seq = {1,2,3};
// Declare the local variable and initialize it later
```

```
Set<String> String_set;
String_set = {"apple","ananas","banana"};
inoutparam=4;
Employee oneEmployee = Employee::instances().one();
Employee tmpEmployee = one_employee;
tmpEmployee.name="Bob"; //modifying the object pointed by oneEmployee
// variable as well
a=b=4;
```

Literals

In many of the examples, constants have been used as parts of expressions. While this serves well for the purposes of illustration, it should be noted that most analysis models require minimal use of constants since such data is more commonly stored as attributes of specification classes.

Syntax

The syntax depends on the base data type:
Integer: 1, 42, -127, etc.
Real: 1.0, 4.5, -56.0, etc.
String: "String"
Boolean: true, false

Notes

Constants may be defined for the above data types only.
A constant may be used in any construct requiring an expression.

Cast Expression

Syntax

```
(<target_type>)<expression>;
```

Note

A cast expression is used to filter the values of its operand expression to those of a given type. The type is named within parentheses and prefixes the operand expression as an effective unary operator. This version of rAlf does not support the casting of collections.

Example

```
(MyType)this.getTypes()
(Person)invoice.payingParty
(MyAncestor)this
```

Operations

Operation Invocation

The modeler may define static and non-static operations as desired. An operation invocation can be used as a stand-alone statement or it can be used in an expression.

Syntax

As an operation expression:

```
<class>::<static operation name> (<data item>=>
<expression>, ...)
<Object reference>.<non-static operation name> (<data item> =>
<expression>, ...)
```

As a stand-alone operation assignment statement:

```
<variable> = <class>::<static operation name> (<data
item> => <expression>, ...);
<variable> = <Object reference>.<non-static operation name>
(<data item> => <expression>, ...);
```

`<class>` is the name of a class.
`<Object reference>` is a handle to an instance of a class.
`<static operation>` and `<non-static operation>` are the name of the operation.
`<data item>` is the name of a data item defined as input for `<static operation name>` or `<non-static operation name>` .
`<expression>` is a String, arithmetic, Boolean, simple, or compound expression. The data type of the expression must match the data type defined for the given data item.
`<variable>` is a class attribute or a local variable.

Notes

An operation expression may be used anywhere an expression is valid (e.g., assignment statement read values, control logic expressions, etc.). The user may use an operation expression as a read value and use the return value to assign to a `<variable>`.
Parentheses are required even if there are no data items.
The type of `<variable>` either:

- the same data type as the output value of the operation or
- the output value of the operation is of type `Integer` and `<variable>` is of type `Real`.

For `in` parameters, the argument is a value that is assigned to the corresponding parameter.
For `inout` and `out` parameters, the argument must be an expression of the form that is legal on the left hand side of an assignment, that is, a local variable or an attribute access.

Example

```
// operation expressions without assigning the return value
window.open();
// static operation call
Log::LogInfo(message=>”Ready”);
// operation expression as an assignment statement read value
volume = hifi.getVolume();
```

Return Statement

Since class operations can return a value, the `return` statement is accepted within their actions.

Syntax

```
return <expression>;
return;
```

<expression> is a String, arithmetic, Boolean, simple or compound expression. The data type of the expression must match the data type defined for the return value of the class operation.

Notes

When executed, the *return* statement causes control to be returned to the caller.
The value returned to the caller is *<expression>*.
If there is no return value specified for the operation, then *<expression>* must be omitted.

Example

```
Dog dog = Dog::instances().one();
return dog.weight;
if(i>5){
      return;
}
```

Parameters

Static and non-static operations can accept parameters. Parameter has a direction; *in, out or inout. A* parameter is similar to a variable. One can read its value, and change its value (only for *out* and *inout* parameters) using the name of the parameter defined in the Operation signature.

Syntax

```
<parameter>
```

<parameter> is the name of a parameter.

Note

<parameter> is an *<expression>* and so can be used in any rAlf construct specifying an expression.

Example

```
// For an invocation like MATH::SQR(x=>3)
return x * x;

//Changing the value of an inout parameter which is a collection of
// Integers
param_inout[2] = 5;
```

Use of this keyword

Non-static operations can use the keyword *this* to reference the instance to which the operation is currently being applied. The *this* keyword can be used anywhere that is valid.

Note

Static operations cannot use the *this* keyword since they are not related to any instance.

Example

```
// attribute access
this.a = this.b;
// event generation
```

```
send new E::one() to this;
```

Other Differences

Static and non-static operations may not refer to the `sigdata` keyword. This keyword is only allowed within a state or transaction action since these are the only actions that can receive signals.

Collection Library

The rAlf::Library::CollectionClasses package contains a set of template classes related to collections. These provide the ability to create and manipulate collections in a manner familiar from object-oriented programming languages.

Set<T>

Concrete unordered, unique collection. It doesn't support duplicate entries.

Operations

1. *`<T> one()`*
   Returns any element from the set or null if the set is empty.
2. *`Boolean isEmpty()`*
   Return true if this collection contains no elements.
3. *`Integer size()`*
   Return the number of elements in this set.
4. *`Boolean add(<T> element)`*
   Insert the given element into this set. Return true if a new element is actually inserted.
5. *`Set<T> filter(FilterExpression)`*
   Returns a new set containing elements on which the *`FilterExpression`* evaluated to true.

Sequence<T>

Concrete ordered by position, non-unique collection. Supports duplicate entries.

Operations

1. *`<T> sequence[Integer index]`* Returns the element in the given index.
2. *`<T> one()`*
   Returns any element from the sequence or null if the sequence is empty.
3. *`Boolean isEmpty()`*
   Return true if this collection contains no elements.
4. *`Integer size()`*
   Return the number of elements in this sequence.
5. *`add(<T> element)`*
   Appends the specified element to the end of this list.
6. *`add(Integer index, <T>)`*
   Insert the specified element in the specified position into this sequence. Shifts the element currently at that position (if any) and any subsequent elements to the right (adds one to their indices).

7. *`Sequence<T> filter(FilterExpression)`*
   Returns a new sequence containing elements on which the *`FilterExpression`* evaluated to true.