Annex A: Java to UML Activity Mapping

(normative)

A.1 General

The specifications for the methods of operations in the execution model in Clause 8 are written as Java code. However, as discussed in sub clause 8.1, this Java code is to be interpreted as a surface syntax for UML activity model. This annex defines the normative mapping from the Java syntax used in the execution model to UML activity models.

Sub clause A.2 defines the correspondence between type names in the Java code and types in UML. The remaining sub clauses map Java behavioral code to UML activity models. In each case, the mapping is giving in terms of a pattern of Java code and the pattern for the corresponding UML activity model (except for the case of a Java while loop, which is mapped to an equivalent Java do-while loop and, from that, to a UML activity model). The rules for the mapping are also described textually. The textual rules are intended to be used in conjunction with the graphical depiction of the mapping.

This mapping does not cover the entire Java language. Rather, there are specific conventions, noted in the following sub clauses, which must be followed in the Java code in order to allow it to be mapped to UML. Further, the result of this mapping is only subset of the full set of possible UML activity models. This subset defines the behavioral modeling capabilities included in the *Base UML* (or "bUML") subset of fUML that is used to write the fUML execution model. Clause 10 gives the base semantics for the bUML subset.

A.2 Type Names

Table A.1 defines the mapping from type names mentioned in the Java code to corresponding UML types.

Note that Java variables typed by a class are always allowed to have the empty value "null." This is considered to correspond, in UML, to the empty case of no values. Thus, all Java types are mapped to UML multiplicity elements with a lower bound of 0.

Further, types with names of the form "…List" map to UML multiplicity elements with an unlimited upper bound. See A.6 for more on list types.

Table A.1 - Java to UML Type Name Mapping

Java	UML
Primitive types	
int	Integer
float	Real
boolean	Boolean
String	String
fUML.Syntax.UnlimitedNatural	UnlimitedNatural
Classes	
<class name>	<class name> [0..1]

Table A.1 - Java to UML Type Name Mapping

<package name>.<class name>	<package name>::<class name> [0..1] (Note: “.” separators are replaced by “::” in the package name)
Lists	
<type name>List	<type name> [*] {ordered, non-unique}

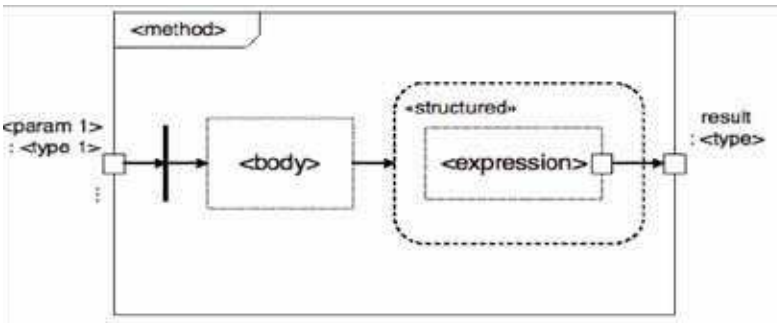
A.3 Method Declaration

Java

```
public <type> <method>
    (<type 1> <param 1>, ...)
{
     <body>
     return <expression>;
}
```

- A method with a non-void type must have a single return statement at the end of its body. A void method may not have any return statements, except that a method with no other statements may have a single “return” statement in its body.

UML



- A method maps to an activity with the corresponding operation as its specification.
- The parameters of the method map to input parameters of the activity, with corresponding activity parameter nodes. The result type of the method, of other than void, maps to a single result parameter of the activity, with a corresponding activity parameter node. If the method has a void type, the activity has no result parameter.
- The body of the method is mapped as a sequence of statements (see sub clause A.4.1).
- Each input activity parameter node is connected by an object flow to a fork node. A use of a method parameter in the body maps to an object flow from the fork node connected to the corresponding input activity parameter node into the mapping of the body.

- A return statement (with an expression) maps to a structured activity node containing the mapping of the return expression (see A.5), with a control flow dependency on the mapping of the final statement of the body (unless this is empty). Object flows may flow from within the mapping of the body into the mapping of the expression. An object flow connects the result pin of the expression to the result activity parameter node. (A return statement for an otherwise empty method is not mapped to anything.)

A.4 Statements

The following mappings are for statements and sequences of statements that appear in the bodies of methods and structured statements. Statements often have embedded expressions, which are mapped according to the mappings given in A.5.

A.4.1 Statement Sequence

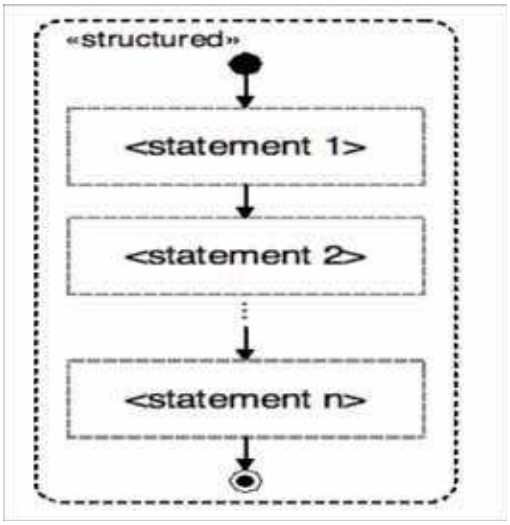
Java

```
<statement 1>;
<statement 2>;
...

<statement n>
```

- Allowable statements include only those with a form that has a mapping defined in the remainder of this sub clause.

UML



- The mapping of a sequence of statements consists of a structured activity node that contains the mapping of each statement (as given in the remainder of this sub clause).
- The mapping of the first statement in the sequence has an incoming control flow from an initial node. The mapping of each subsequent statement has a control flow from the mapping of the previous statement. The mapping of the last statement has an outgoing control flow to an activity final node. (An empty sequence maps to a structured activity node with an initial node connected directly to a final node.)

- Object flows from within the mapping of one statement may flow into the mapping of a subsequent statement.

Notes

- The actual sources and targets of the control flows within the statement mappings are noted in the mapping for each kind of statement.

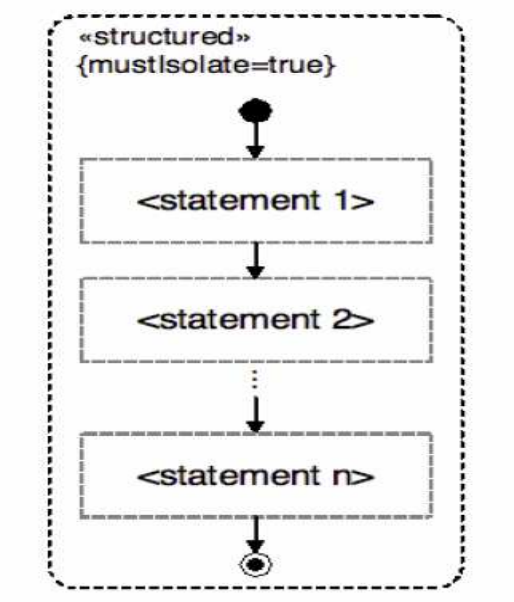
A.4.2 Statement Sequence (isolated)

Java

```
_beginIsolation();
     <statement 1>;
     <statement 2>;
     …
     <statement n>;
_endIsolation();
```

- A set of statements the must run in “isolation” are represented by a sequence of statements, the first statement of which is a call to the “_beginIsolation()” method and the last statement is a call to the “_endIsolation()” method.
- A user class may not define methods called “_beginIsolation” or “_endIsolation.”

UML



- The sequence of statements in the block is mapped in exactly the same way as for a normal sequence of statements (see sub clause A.4.1), but the enclosing structured activity node has mustIsolate=true.

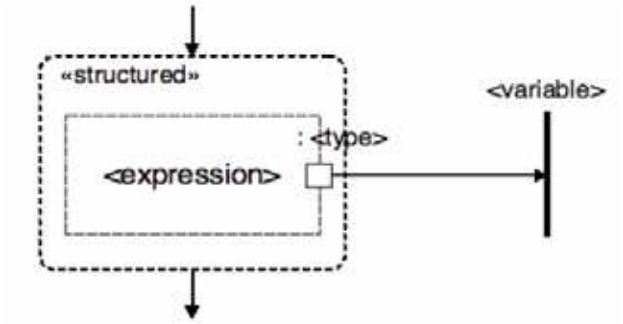
A.4.3 Local Variable Declaration

Java

```
<type> <variable> =
<expression>;
```

- A local variable declaration is required to have an initialization expression.
- It is not permitted to reassign the value of a local variable, except as specifically allowed in the context of an if statement or loop (see A.4.9 and A.4.2).

UML



- A local variable declaration maps to fork node that receives an object flow from the result of the mapping of the initialization expression (see A.5).
- The mapping of the initialization expression is nested inside a structured activity node. Incoming and outgoing control flows (if any) attach to the structured activity node.

Notes

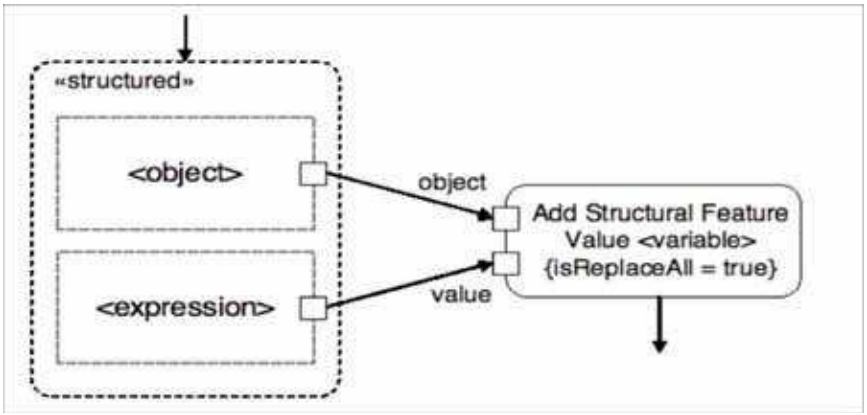
- The use of the fork node models the ability to read the value of a local variable multiple times.
- If the local variable is re-assigned as part of a subsequent if statement or loop, then uses of the variable after that point will be mapped to flows from a different fork node than the one resulting from the mapping of the variable declaration (see A.4.1 and A.4.2).

A.4.4 Instance Variable Assignment (non-list)

Java

```
<object>.<variable> = <expression>;
```

UML



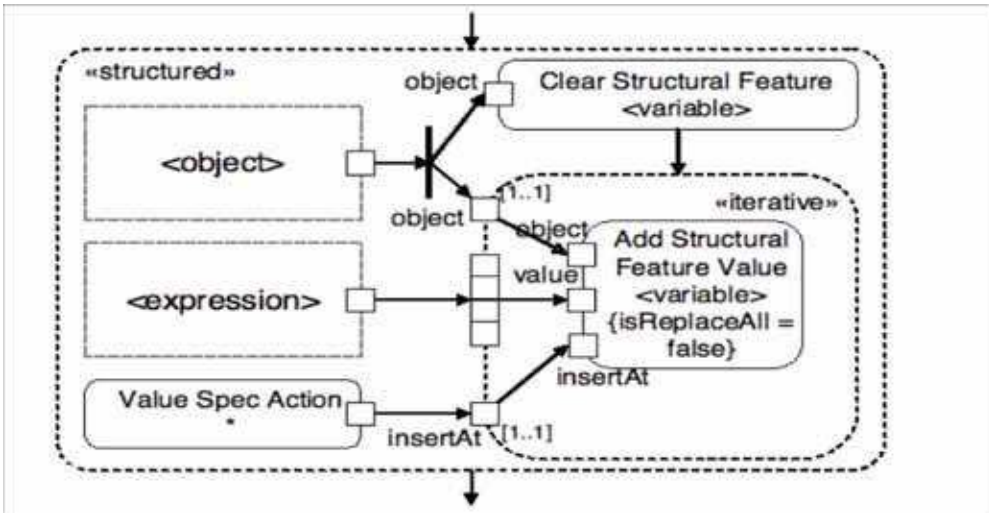
- The assignment of a non-list instance variable maps to an add structural feature value action with isReplaceAll = true. (For instance variables of a list type, see A.4.5).
- The object and assigned expressions map as given in A.5. Their mappings are nested in a structured activity node.
- The object input pin of the add structural feature value action is connected by an object flow to the result pin of the mapping of the object expression.
- The value pin of the add structure feature value action is connected by an object flow to the result pin of the mapping of the assigned expression.
- An incoming control flow (if any) attaches to the structured activity node containing the expression mappings. An outgoing control flow (if any) attaches to the add structural feature action.

A.4.5 Instance Variable Assignment (list)

Java

```
<object>.<variable> = <expression>;
```

UML



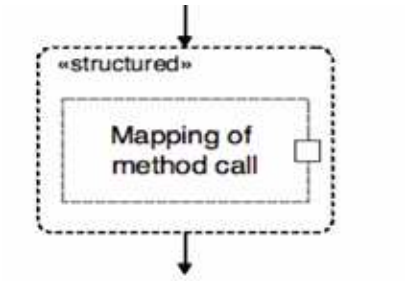
- The assignment of a list instance variable maps to a clear structural feature action followed by an expansion region containing an add structural feature value action with isReplaceAll = false. (For assignment of an instance variable of a non-list type, see A.4.4).
- The object and assigned expressions map as given in A.5.
- The object input pin of the clear structure feature action and an input pin (multiplicity [1..1]) of the expansion region are connected by object flows to a fork node that is connected by an object flow to the result pin of the mapping of the object expression.
- An input expansion node on the expansion region is connected by an object flow to the result pin of the mapping of the assigned expression.
- An input pin on the expansion region is connected by an object flow to the result ping of a value specification action that produces an unlimited natural * (unbounded) value.
- Inside the expansion region, the object input pin of the add structural feature value action is connection by an object flow to the object input pin of the expansion region, its value pin is connected by an object flow to the expansion node and its insertAt pin is connected by an object flow to the insertAt input pin of the expansion region.
- The expression mappings, clear structural feature action and expansion region are all nested in a structured activity node. Incoming and outgoing control flows (if any) attach to the structured activity node.

A.4.6 Method Call Statement

Java

```
<object>.<method>(<argument 1>,…);
```

UML



- A statement containing only a method call maps as a method call expression (see A.4.1). The result pin of the mapping (if any) has no outgoing object flow.
- The mapping of the method call is nested inside a structured activity node. Incoming and outgoing control flows (if any) attach to the structured activity node.
- A statement containing only a super call maps in a similar manner, but with a super call expression (see A.4.2) rather than a method call expression.

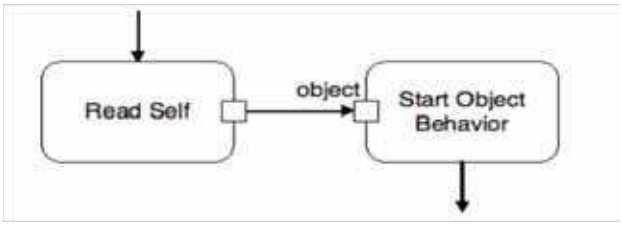
A.4.7 Start Object Behavior

Java

```
_startObjectBehavior();
```

- A class may not define a user operation called "_startObjectBehavior."
- The _startObjecttBehavior method may not be called explicitly on any other object.

UML



- A _startObjectBehavior call maps to a start object behavior action.
- The object input pin of the start object behavior action is connected by an object flow to the result pin of a read self action.
- An incoming control flow (if any) attaches to the read self action. An outgoing control flow (if any) attaches to the start object behavior action.

Notes

- An object can have at most one classifier behavior. The _startObjectBehavior method starts this in a separate thread and returns immediately.
- This mapping is an exception to the normal “Method Call Statement” mapping of sub clause A.4.6.

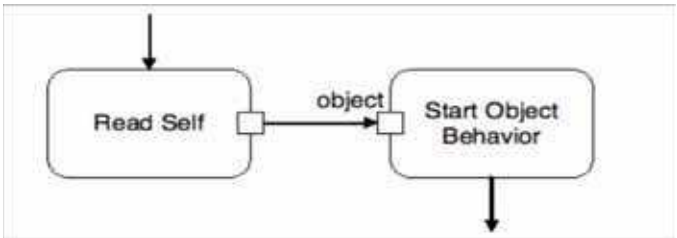
A.4.8 Signal Send

Java

```
_send(new <signal>());
```

- The constructor for a signal may not have any arguments. (Signals with attributes are not allowed.)

UML



- A _send method call maps to a send signal action for the constructed signal.
- The target input pin of the send signal operation is connected by an object flow to the result pin of a read self action.
- An incoming control flow (if any) attaches to the read self action. An outgoing control flow (if any) attaches to the send signal action.

Notes

- This is an exception to the normal “Method Call Statement” mapping of sub clause A.4.6.
- The classifier behavior of the class containing the method making the _send call must have an accept event action for the signal.

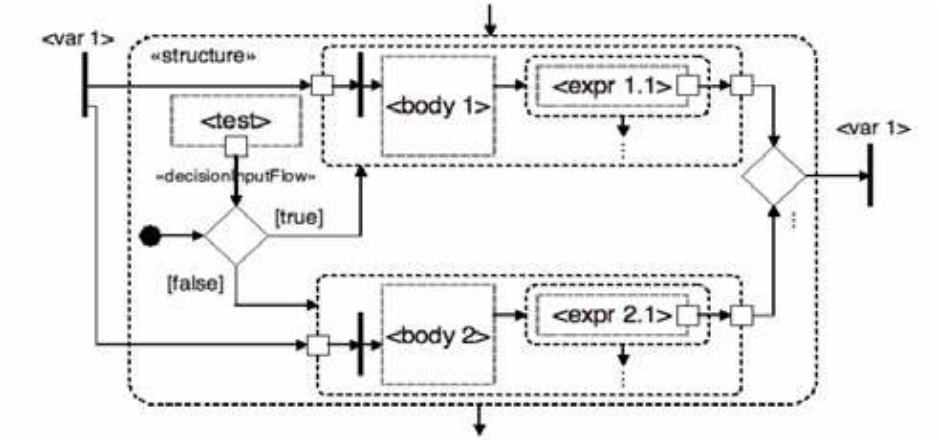
A.4.9 If Statement

Java

```
if (<test>) {
     <body 1>
     <var 1> = <expr 1.1>;
     …
} else {
     <body 2>
     <var 1> = <expr 2.1>;
     …
}
```

- At the end of the body of each branch of the if statement, there may be assignment statements for local variables declared outside the if statement.

UML



- An if statement maps to a decision node with two outgoing control flows, one with the guard “true” and one with the guard “false” and an incoming control flow from an initial node.
- The decision node has a “decision input” data flow from the result pin of the mapping of the test expression (see A.5).
- Each body maps as a structured activity node containing the mapping of a sequence of statements (see A.4.1). The “true” control flow from the decision node connects to the structured activity node for the first body. The “false” control flow similarly connects to the structured activity node for the second (“else”) body.
- The structured activity nodes for each branch have input and output pins corresponding to the variables assigned in either branch. Object flows connect the source for each variable to the corresponding input pins and each input pin to a fork node within the structured activity node for the branch. Object flows connect the two output pins corresponding to a variable (one from each branch) to a merge node, which then has an object flow to a fork node. The fork node is acts as the source for all uses of the variable subsequently to the if statement.
- Each variable assignment maps to a structured activity node containing the mapping of the assigned expression. The first structured activity node has a control flow dependency on the mapping of the last statement of the branch body (if any) and each subsequent node has a control flow dependency on the previous node. The result pin of each expression has an object flow to the output pin for the branch corresponding to the variable being assigned. If a variable is not assigned in a branch, then the input pin for the variable is connect by an object flow directly to the output pin, within the structured activity node for the branch. If a variable is used in a subsequent assignment expression, then a fork node must be inserted to fork the object flow out of the expression result to both the branch output pin and any subsequent variable use(s).
- The input pins of a structured activity node for a branch act as the source for all uses of the corresponding variables within the branch. For any other variable uses, object flows may flow directly into the mappings of the parts of the if statement. Object flows from within the mapping of the body may flow into the mappings of the expressions.

- If the if statement has no else branch, but there are variable assignments in the “true” branch, then there is still a structured activity node for the “false” branch, with all input pins connected to output pins. If there is no else branch and no variable assignments, then the “false” branch structured activity node may be replaced by an activity final node.
- Incoming and outgoing control flows (if any) attach to the outermost structured activity node.

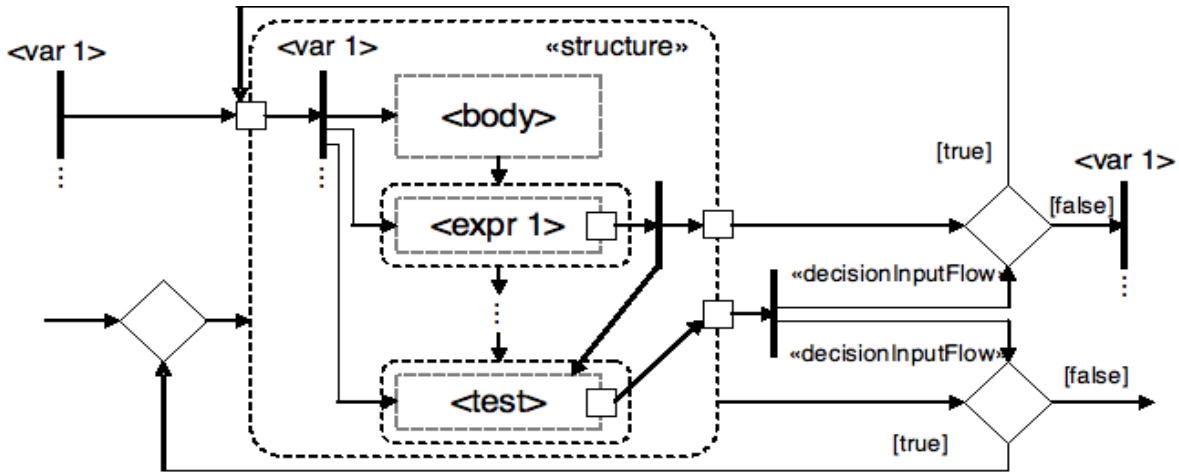
A.4.10 Do-While Loop

Java

```
do {
     <body>
     <var 1> = <expr 1>;
     …
} while (<test>)
```

- A do-while loop may contain variable assignments at the end of its body for local variables declared outside the loop.

UML



- A do-while loop maps to a structured activity node with a looping control structure outside it, as shown above. An incoming control flow comes into the merge node shown on the left, and the outgoing control flow is the “false” flow out of the decision node shown on the bottom right.
- The body of the do-while loop maps as a sequence of statements (see A.4.2).
- Every variable referenced in the body of the while loop (whether it is assigned or not) has corresponding input and output pins on the structured activity node for the loop. The input pin for the variable is connected by an object flow to the mapping for the variable from before the loop. Inside the structured activity node for the loop, each loop variable input pin is connected by an object flow to a fork node. This fork node is used as the source for the mapping of all uses of the variable within the loop (unless, possibly, if the variable is re-assigned within the loop - see below).

- Each variable assignment maps to a structured activity node containing the mapping of the assigned expression. The first structured activity node has a control flow dependency on the mapping of the loop body and each subsequent node has a control flow dependency on the previous node. The result pin of each expression is connected by an object flow to a fork node which then has an object node to the output pin of the outer structured activity node corresponding to the variable being assigned. (If there is no assignment for a variable within the loop, then there is an object flow that connects directly from the fork node for the variable within the loop's structured activity node to the output pin for the variable.)
- The test expression maps to a structured activity node containing the mapping of the expression. There is a control flow from the structured activity node for the mapping of the last variable assignment expression to the structured activity node for the test expression. (If there are no variable assignments, the control flow comes from the mapping of the body.) The result pin of the test expression is connected by an object flow to an output pin of the outer structured activity node for the loop.
- If an assignment or test expression uses a variable previously assigned, then that use maps to an object flow from the fork node attached to the result pin of the assignment expression, rather than the fork node attached to the loop input pin.
- The test result output pin of the structured activity node for the loop is connected to a fork node outside the structured activity node, which is then connected by an object flow to the decision input flow of the loop control decision node. The output pin for each loop variable is connected by an object flow to a decision node. The decision input flow for the decision node is an object flow from the test result fork node. The “true” outgoing flow from the decision node connects back to the corresponding input pin and the false flow connects to a fork node, which is used as the source of the variable for all mappings of expressions after the while loop.

A.4.11 While Loop

Java

```
while (<test>) {
     <body>
     <var 1> = <expr 1>;
     …
}
```

- A while loop may contain variable assignments at the end of its body for local variables declared outside the loop.

Equivalent Java

```
if (<test>) {
     do {
          <body>
          <var 1> = <expr 1>;
          …
     } while (<test>)
}
```

- A while loop maps as if it was coded as a do-while loop (see A.4.10) nested in an if statement (see A.4.9), as shown above.

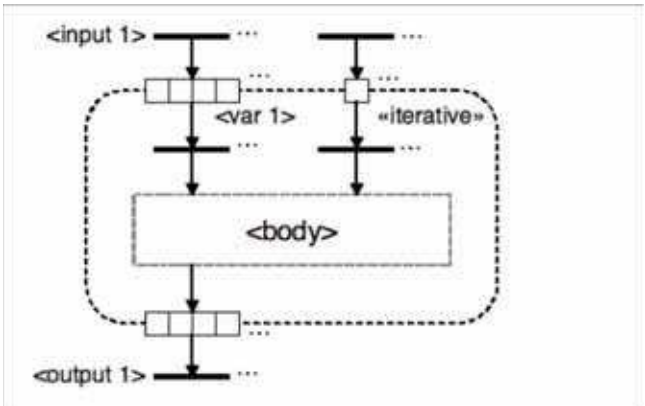
A.4.12 For Loop (iterative)

Java

```
<output type 1> <output 1> = new <output type 1>();
…
for (int <index> = 0;
      <index> < <input 1>.size(); i++) {
   <type 1> <var 1> = <input 1>.getValue(<index>);
   …
   <body>
}
```

- An iterative for loop must have a locally declared index variable of type “int” that is sequentially incremented (see also A.4.12).
- The body of the for loop must begin with one or more loop variable declarations, each initialized by an access to a different list variable with the loop index variable. The list variables must be declared outside the loop and all have list types (see A.6). The loop index variable may not otherwise be used in the body of the loop.
- The for loop must be indexed based on the size of the first list variable, as shown above.
- The for loop must not have any assignment statements at the end of its body.
- The body of the for loop may include nested statements of the form “<output n>.addValue(…),” where “<output n>” is a variable of a list type declared outside the loop and initialized to an empty list of the appropriate type.

UML



- A for loop with the structure given above is mapped to an iterative expansion region.
- The local loop variables map to input expansion nodes on the expansion region. The expansion node for the variable is connected outside the expansion region by an object flow to the mapping for the corresponding list variable from before the loop. It is connected inside the expansion region to a fork node. A reference to a loop variable within the loop body maps to an object flow from the fork node connected to the corresponding expansion node.

- For any variable declared outside the loop and referenced within the body of the loop, other than the local loop variables as defined above, there is a corresponding input pin on the expansion region. The input pin is connected outside the expansion region by an object flow to the fork node corresponding to the variable. The input pin is connected inside the expansion region to a fork node, which is then used as the source for references to the variable within the mapping of the body of the loop.
- The body of the loop maps as a sequence of statements (see A.4.13) nested in the expansion region.
- If there are any "addValue" statements within the body of the loop, then there is an output expansion node for each referenced output list variable. Each "addValue" statement maps to an object flow from the result of the argument expression of the "addValue" call to the appropriate output expansion node. Each output expansion node is connected by an object flow to a fork node that is used as the source for references to the corresponding output list variable in any subsequent statements.

Notes

- The mapping for the element variables is an exception to the normal rules for list indexing (see A.6.7) and for variable use (see A.5.1).

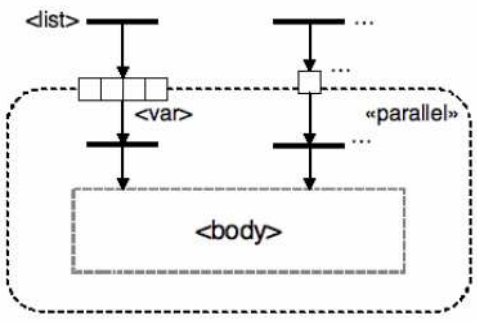
A.4.13 For Loop (parallel)

Java

```
for (Iterator <iter> = <list>.iterator();
      <iter>.hasNext();) {
   <type> <var> = (<type>)(<list>.next());
   …
   <body>
}
```

- A parallel for loop must be indexed by an iterator based on a list variable (see A.5.6). The list variable must be declared outside the loop and have a list type (see A.6).
- The body of the for loop must begin with exactly one variable declaration, initialized by an access to the loop iterator.
- The for loop must not have any assignment statements at the end of its body.
- The behavior of the body must not depend on the specific order in which list items are returned.

UML



- A for loop with the structure given above is mapped to a parallel expansion region.
- The local loop variable maps to a single input expansion node on the expansion region. (There are no output expansion nodes.) The expansion node for the variable is connected outside the expansion region by an object flow to the mapping for the corresponding list variable from before the loop. It is connected inside the expansion region to a fork node. A reference to a loop variable within the loop body maps to an object flow from the fork node connected to the corresponding expansion node.
- For any variable declared outside the loop and referenced within the body of the loop, other than the local loop variables as defined above, there is a corresponding input pin on the expansion region. The input pin is connected outside the expansion region by an object flow to the fork node corresponding to the variable. The input pin is connected inside the expansion region to a fork node, which is then used as the source for references to the variable within the mapping of the body of the loop.
- The body of the loop maps as a sequence of statements (see A.4.1).

Notes

- The Java code will execute the body iterations in a specific sequential order, but the behavior of the Java is not allowed to depend on what that order actually is.
- The mapping for the element variables is an exception to the normal rules for list indexing (see A.6.7) and for variable use (see A.5.6).

A.5 Expressions

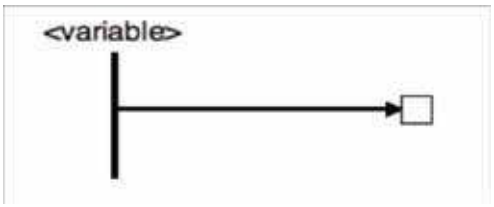
The following mappings are for expressions that are embedded within statements. Each expression maps to a fragment of an activity model that has a distinguished “result pin” (with the exception of the mapping of sub clause A.5.1. It is this result pin to which an object flow may be connected to obtain the output of the expression.

A.5.1 Local Variable or Method Parameter Use

Java

```
<variable>
```

UML



The use of a local variable or method parameter in an expression maps to an object flow from the fork node corresponding to the variable or parameter to an input pin of the mapping of the remainder of the enclosing expression.

Notes

- The fork node may result from the mapping of a method parameter, directly from the mapping of the declaration of the variable (see above), from the mapping of the output of an if statement or a loop (see A.4.1 and A.4.2) or from the mapping of the loop variable of a fork node.

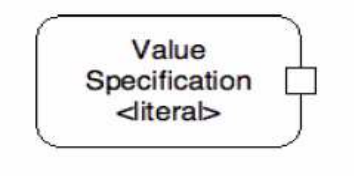
A.5.2 Literal

Java

```
<literal>
```

- The literal must be an integer, a boolean, a string, or an UnlimitedNatural.
- An UnlimitedNatural literal is created using a constructor expression of the form “new fUML.Syntax.UnlimitedNatural(n),” where n is a non-negative integer or -1 (used to represent "*").
- The integer value of an UnlimitedNatural value “x” is obtained by an expression of the form “x.value.”

UML



- A literal is mapped to a value specification action with a corresponding literal value. The result output pin of the value specification action becomes the result pin of the mapping.

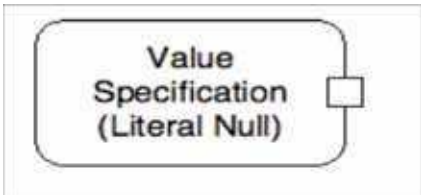
A.5.3 Null

Java

```
null
```

- A null value may not be used for a list type.

UML



- A null value maps to a value specification action for a literal null. The result output pin of the value specification action becomes the result pin of the mapping.

Notes

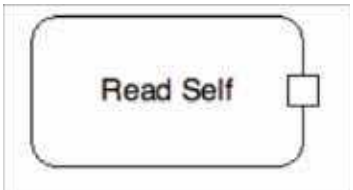
- All class types in Java allow “null” values. Such types map to types with “optional” multiplicity [0..1] in UML (see A.2). Java “null” is used to represent the case of “no value” (0 cardinality) allowed by this multiplicity. A value specification for a literal null places no values on its output pin when it executes, corresponding to the 0 cardinality case.
- Since a Java “null” maps to “no value” in UML, testing for a null value requires a special mapping (see A.5.3).
- For the mapping of an empty list, see A.6.4.

A.5.4 This

Java

```
this
```

UML



- A use of “this” maps to a read self action whose result pin is the result pin for the expression mapping.

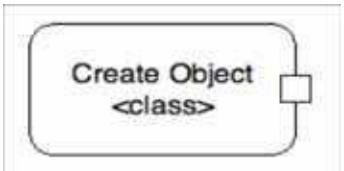
A.5.5 Constructor Call

Java

```
new <class>()
```

- Constructor calls are not allowed to have arguments.

UML



- A constructor call maps to a create object action for the named class. The result output pin of the create object action becomes the result pin for the mapping.

Notes

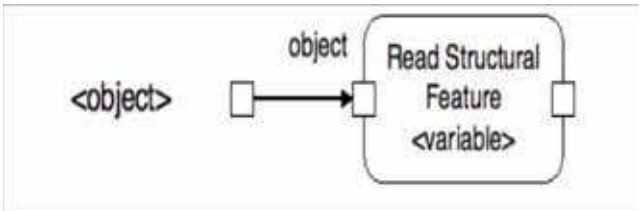
- This mapping does not apply to the case of the class being a list type (see A.6.4 for the construction of an empty list).
- This mapping does not apply to the case of creating an UnlimitedNatural value (see A.5.5).

A.5.6 Instance Variable Use

Java

```
<object>.<variable>
```

UML



- The use of an instance variable within an expression maps to a read structural feature action for the attribute corresponding to the instance variable. The result output pin of the read structural feature action becomes the result pin for the mapping.
- The object input pin of the read structural feature action is connected by an object flow to the result pin of the mapping of the expression evaluating to the target object.

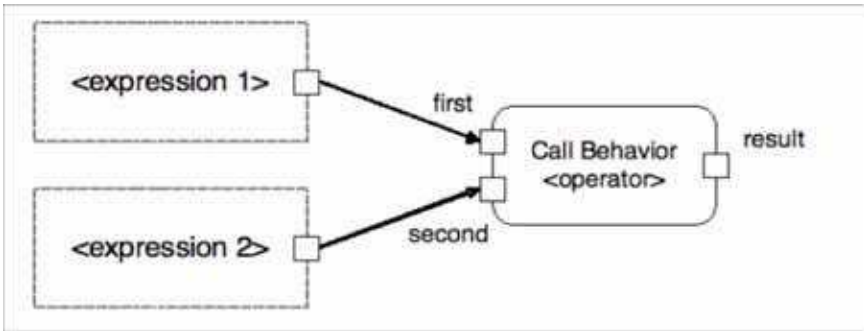
A.5.7 Operator Expression

Java

```
<expression 1> <operator> <expression 2>
```

- The operator must be an integer arithmetic operator or a boolean relational operator other than equals or not equals (for testing equality, see A.5.7 and A.5.8).

UML



- An infix operator expression maps to a call behavior action for the primitive behavior corresponding to the operator (chosen from the Foundational Model Library, see 9.3). The result output pin of the call behavior action becomes the result pin of the mapping.
- The first argument input pin of the call behavior action is connected by an object flow to the result pin of the mapping of the left sub-expression. The second argument input pin of the call behavior action is connected by an object flow to the result pin of the mapping of the right sub-expression.
- A prefix operator is mapped similarly, except that there is only one sub-expression and only one argument input pin to the call behavior action.

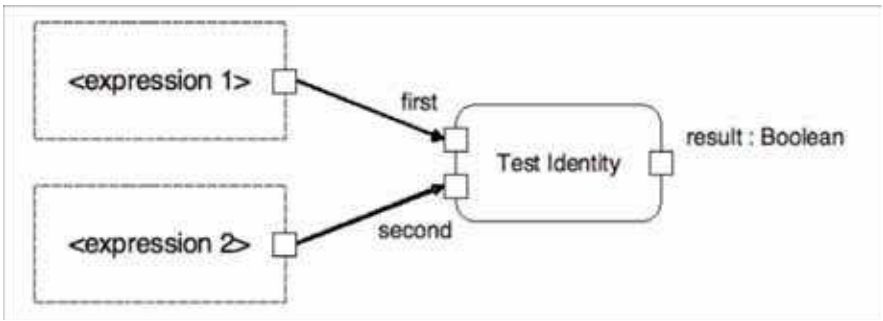
A.5.8 Testing For Equality

Java

```
<expression 1> == <expression 2>
```

- Neither expression may evaluate to null (for testing for null, see A.5.3).
- For UnlimitedNatural values, their integer values must be compared, not the object themselves (see also A.5.7).
- The expressions may not be of type String (for testing string equality, see A.5.8).
- The expressions may not have list types (for more on lists, see A.6).

UML



- An equality test maps to a test identity action. The result output pin of the test identity action becomes the result pin of the mapping.
- The first argument input pin of the test identity action is connected by an object flow to the result pin of the mapping of the left sub-expression. The second argument input pin of the test identity action is connected by an object flow to the result pin of the mapping of the right sub-expression.
- The expression “<expression 1> != <expression 2>” is mapped as if it was “!(<expression 1> == <expression 2>).”

Notes

- For primitive values, the test identity action tests for equality of value. For object references, it tests the identity of the referent objects.

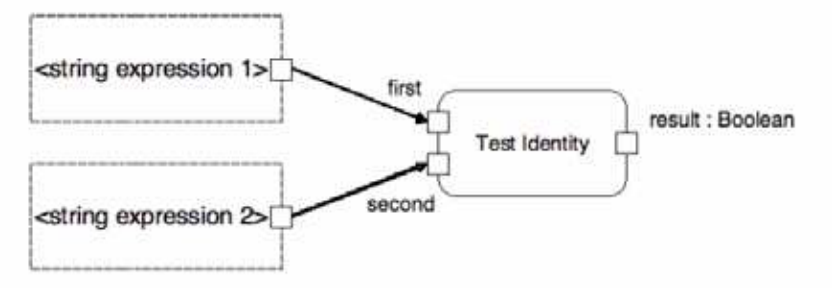
A.5.9 Testing String Equality

Java

```
<string expression 1>.equals(<string expression 2>)
```

- Strings are never tested for equality using "==".

UML



- A string equality test maps to a test identity action. The result output pin of the test identity action becomes the result pin of the mapping.
- The first argument input pin of the test identity action is connected by an object flow to the result pin of the mapping of the left sub-expression. The second argument input pin of the test identity action is connected by an object flow to the result pin of the mapping of the right sub-expression.

Notes

- In Java String is a class, and testing string values using "==" tests the identity of the string objects being tested, not equality of their values. In UML String is a primitive type, and the test identity action tests for equality of value for strings.

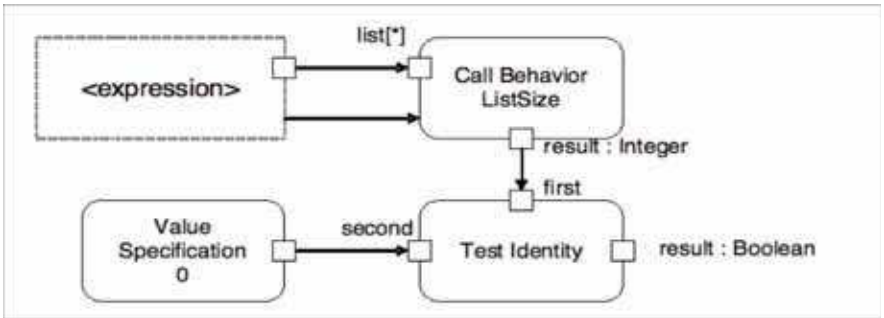
A.5.10 Testing For Null

Java

```
<expression> == null
```

The expression being tested may not have a list type (see A.6).

UML



- A test for null is mapped to a test for whether the result of the mapping of the expression has a list size of zero.
- The result pin of the mapping of the expression is connected by an object flow to the argument pin of a call behavior action for the ListSize behavior (with multiplicity *).
- The call behavior action has a control flow from the action owning the result pin of the mapping of the list expression.
- The result output pin of the call behavior action is connected by an object flow to the first argument pin of a test identity action. The second argument pin of the test identity action is connected by an object flow to the result pin of a value specification action for the integer value "0". The result output pin of the test identity action becomes the result pin for the mapping.
- The expression “<expression> != null” is mapped as if it was “!(<expression> == null)”.

Notes

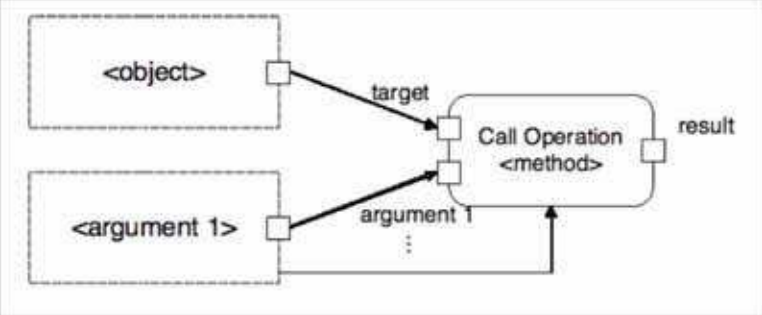
- Java null is used to represent the case of “no value” for a class type with multiplicity [0..1] (see A.5.3).
- The ListSize behavior is provided in the Foundational Model Library (see 9.3; see also A.6.6).
- Since the input pin to the call behavior action has a multiplicity lower bound of 0, the control flow is necessary to ensure that the call does not happen before the completion of execution of the mapping of the list expression.

A.5.11 Method Call

Java

```
<object>.<method>(<argument 1>,…)
```

UML



- A method call maps to a call operation action for the operation corresponding to the named method. The result output pin of the call operation action becomes the result pin of mapping. (If the method has a void return type, then there is no result pin.)
- The target input pin of the call operation action is connected by an object flow to the result pin of the mapping of the object expression.
- Each argument input pin (if any) of the call operation action is connected by an object flow to the result pin of the mapping of the corresponding argument expression (in order).
- Unless an argument is of a primitive type, the call operation action has a control flow from the action that owns the result pin of the mapping of the argument expression.

Notes

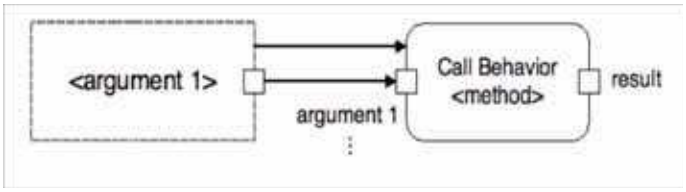
- Since all non-primitive types map to UML types with multiplicity [0..1] or [*] (see A.2), the control flows are necessary to ensure that the call operation action does not start executing before the arguments are computed.

A.5.12 Super Call

Java

```
super.<method>(<argument 1>,…)
```

UML



- A super call maps to a call behavior action for the UML method (the behavior, not the operation) that implements the UML operation corresponding to the Java method in the superclass. The result output pin of the call behavior action becomes the result pin of mapping. (If the method has a void return type, then there is no result pin.)
- Each argument input pin (if any) of the call behavior action is connected by an object flow to the result pin of the mapping of the corresponding argument expression (in order).
- Unless an argument is of a primitive type, the call operation action has a control flow from the action that owns the result pin of the mapping of the argument expression.

Notes

- Since all non-primitive types map to UML types with multiplicity [0..1] or [*] (see A.2), the control flows are necessary to ensure that the call operation action does not start executing before the arguments are computed.
- This is different than the normal mapping of a method call (see A.5.1), but it is not really an exception, since “super” is not actually a proper expression in Java.

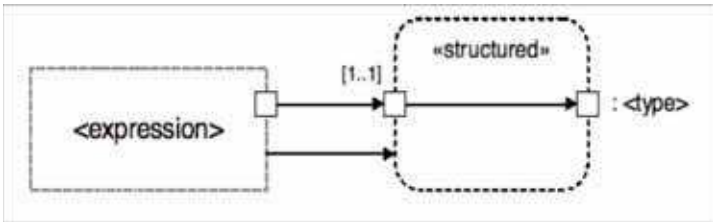
A.5.13 Type Cast (non-primitive)

Java

```
(<type>)<expression>
```

- The expression being cast cannot be of a primitive type.
- The expression being cast cannot be a list.

UML



- A type cast is mapped to a structured activity node that simply copies its input to its output. The input pin of the node is un-typed. The output pin of the node is given the result type of the cast, and it becomes the result pin of the mapping.
- The input pin of the structured activity node is connected by an object flow to the result pin of the mapping of the expression being cast.
- The action that owns the result pin of the mapping of the expression is connected by a control flow to the structured activity node.

Note

- This mapping presumes that the cast is legal. Its behavior is not defined if the result of the expression cannot be cast to the given type.

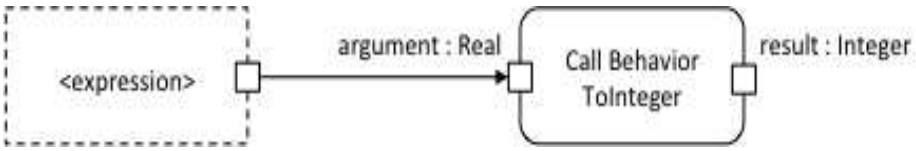
A.5.14 Type Case (numeric)

Java

```
(<type>)<expression>
```

- The type must be int or float.
- The expression being cast must be of type int or float.

UML



- A type cast from int to int, float to float, or int to float is mapped as the expression being cast. The cast itself is ignored, other than that the result pin for the expression being mapped is always given the UML type corresponding to the type of the cast.
- A type cast from float to int maps to a call behavior action for the ToInteger behavior. The result output pin becomes the result pin for the mapping. The argument input pin of the call behavior action is connected by an object flow to the result pin of the mapping of the expression being cast.

Note

- In the base semantics, an integers is a kind of real number (see 10.3.1.2), so no actual operation is needed to cast an integer to a real.

A.6 Lists

Classes with names of the form <base type>List are used to represent lists of values of the type <base type>. List classes are mapped to UML multiplicity elements of the form <base type>[*]{ordered, non-unique} (see A.2). Lists of lists are not allowed.

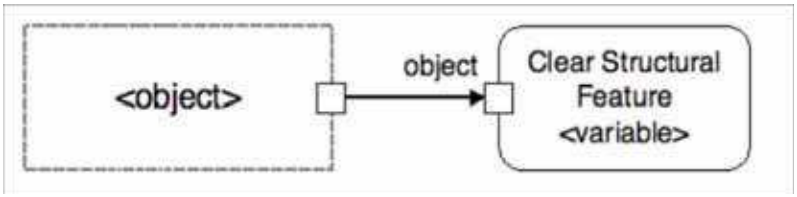
Calls to methods on list classes have special mappings. Calls to the clear, addValue, and removeValue methods map as statements. These methods can only be used on instance variables. A list constructor and calls to the size and get methods map as expressions.

A.6.1 List Clear

Java

```
<object>.<variable>.clear();
```

UML



- A call to the list clear method maps to a clear structural feature action on the attribute corresponding to the list variable.
- The object input pin of the clear structural feature action has an object flow connection to the result pin of the mapping of the object expression.

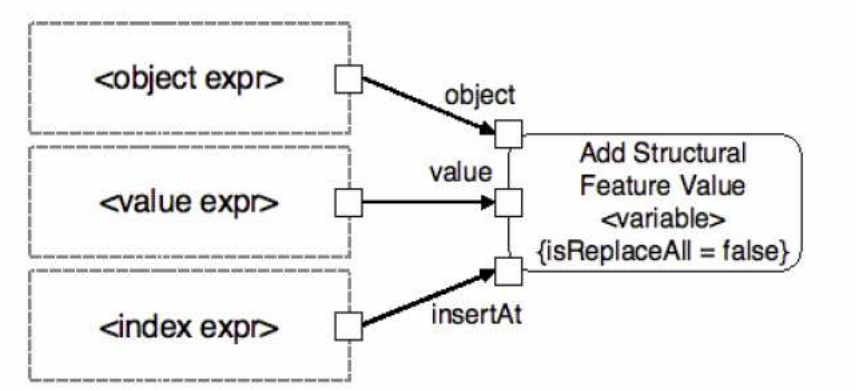
A.6.2 List Add

Java

```
<object expr>.<variable>.addValue(<index expr> - 1, <value expr>)
```

- The value expression must not evaluate to null.

UML



- A call to the list add method maps to an add structural feature value action with isReplaceAll = false.
- The object input pin of the add structural feature value action is connected to the result pin of the mapping of the object expression.
- The value input pin of the add structural feature value action is connected to the result pin of the mapping of the value expression.
- The insertAt input pin of the add structural feature value is connected to the result pin of the mapping of the index expression. If the call does not include an index expression, then the insertAt pin is connected to the result output pin of a value specification action for the UnlimitedNatural value "*".

Note

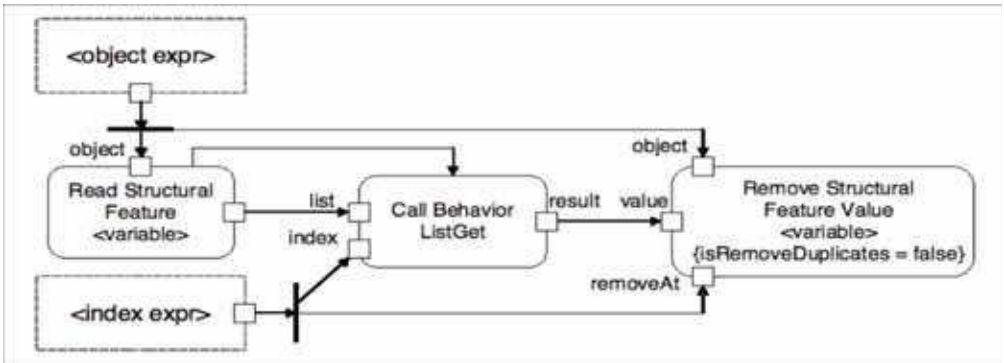
- The Java method indexes from 0, but the add structural feature action indexes from 1.
- Adding a single value to an empty list is an exception to this mapping (see A.6.4).

A.6.3 List Remove

Java

```
<object expr>.<variable>.removeValue(<index expr> - 1)
```

UML



- A call to the list add method maps to a remove structural feature value action with isRemoveDuplicates = false.
- The result pin of the mapping of the object expression is connected by an object flow to a fork node, which, in turn, is connected to the object input pin of the remove structural feature value action and the object input pin of a read structural feature action.
- The result output pin of the read structural feature action is connected to the list input pin of a call behavior action calling the ListGet behavior (see A.6.6). There is also a control flow from the read structural feature action to the call behavior action.
- The result pin of the mapping of the index expression is connected by an object flow to a fork node, which, in turn, is connected to the index input pin of the call behavior action and the removeAt pin of the remove structural feature value action.
- The result output pin of the call behavior action is connected by an object flow to the value input pin of the remove structural feature value action.

Note

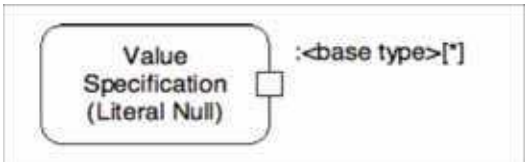
- The Java method indexes from 0, but the remove structural feature action indexes from 1.

A.6.4 Empty List

Java

```
new <base type>List();
```

UML



- A constructor expression for a list type maps to a value specification action for a literal null.

Notes

- A value specification for a literal null places no values on its output pin when it executes, corresponding to the 0 cardinality case of the multiplicity [*].
- This is an exception to the normal rule for mapping “addValue” calls (see A.6.2).

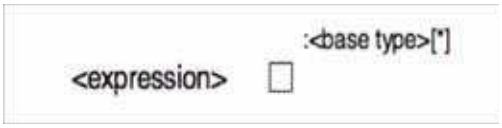
A.6.5 List of One Element

Java

```
<base type>List <var> = new <base type>List();

<var>.addValue(<expression>);
```

UML



A list variable initialized by an empty list, immediately followed by adding a single value to that list, maps to the mapping for the expression that is the argument to the “addValue,” but with the output pin given multiplicity [*].

Notes

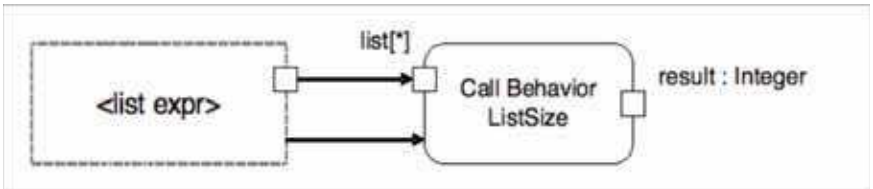
- Since in UML a single element (cardinality 1) conforms to the multiplicity "*", it is not necessary to use an explicit add structural feature value in this case to create the effective mapping of a “list of one element.”
- This is an exception to the normal rule for mapping constructor calls (see A.5.5).

A.6.6 List Size

Java

```
<list expr>.size()
```

UML



- A call to the list size method maps to a call behavior action for the ListSize behavior. The result output pin becomes the result pin for the mapping.

- The argument input pin of the call behavior action (with multiplicity *) is connected by an object flow to the result pin of the mapping of the list expression.
- The call behavior action has a control flow from the action owning the result pin of the mapping of the list expression.

Notes

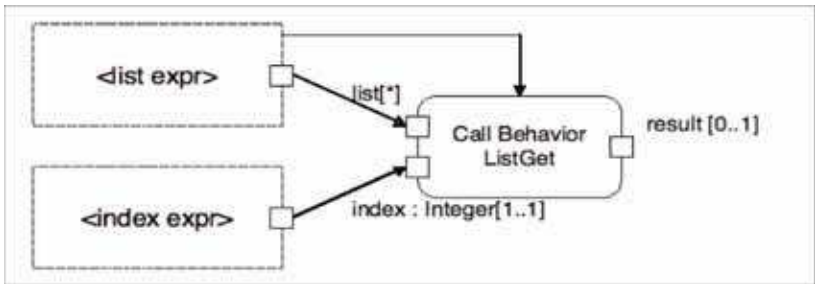
- The ListSize behavior is provided as part of the Foundational Model Library (see 9.3). (It can also be defined as an activity, so it does not have to be primitive.)
- Since the input pin to the call behavior action has a multiplicity lower bound of 0, the control flow is necessary to ensure that the call does not happen before the completion of execution of the mapping of the list expression.

A.6.7 List Indexing

Java

```
<list expr>.getValue(<index expr> - 1)
```

UML



- A call to the list get operation maps to a call behavior action for the ListGet behavior. The result output pin of the call behavior action becomes the result pin of the mapping.
- The list argument input pin of the call behavior action is connected to the result pin for the mapping of the list expression.

- The index argument input pin of the call behavior action is connected by an object flow to the result pin for the mapping of the index expression.
- The call behavior action has a control flow from the action that owns the result pin of the mapping of the list expression.

Notes

- The ListGet behavior is provided as part of the Foundational Model Library (see 9.3). (It can be defined as an activity and so does not have to be primitive.)
- Since the input pin to the call behavior action has a multiplicity lower bound of 0, the control flow is necessary to ensure that the call does not happen before the completion of execution of the mapping of the list expression.
- The Java method indexes from 0, but the ListGet behavior indexes from 1. If the input index value is less than 1 or greater than the size of the input list, no result is generated.