Action Language Examples B

This appendix shows mappings from fragments of specifications in existing action languages to UML Action Semantics models. The intent is to demonstrate by example that the Action Semantics model contains concepts and has a structure required to support real action languages.

The examples also aid the reader in understanding the models. By providing concrete examples in the familiar form of a textual language the reader can more readily see what some of the concepts actually mean. The reader should be aware that the mappings are not definitive and that the Action Semantics specification is normative.

B.1 The Action Languages

The Action Languages used for this mapping have been in use in system development for a number of years. All are Action Languages targeted at object modeling techniques and have primitives built in to support, for example, the creation and deletion of objects and links as well as the sending of signals and the invocation of operations. The example languages are:

- The Action Specification Language (ASL). A public domain language of which there have been several implementations. See "The Action Specification Language Reference Manual" available at www.kc.com.
- The BridgePoint Action Language (AL). An action language supported by the BridgePoint modeling tool. More information is available from www.projtech.com.
- The Kabira Action Semantics (Kabira AS). An action language for the ObjectSwitch middle-tier server suite. More information is available at www.kabira.com.
- The action language subset of SDL. An international standard widely used in the telecom industry. More information is available in ITU-T Recommendations Z.100 (SDL) and Z.109 (SDL UML profile).

The specification allows both for pure data and control flow oriented approaches and for traditional imperative languages. The languages used in this appendix are all examples of the latter style, but there are a number of features that map to individual actions connected by data and control flow. The mappings thus also provide examples of flow connections between actions.

The languages provide, neither individually nor collectively, constructs that make use of all of the Actions described in this document. However, they do cover most of the key features.

A mapping from the action language SDL to these action semantics is described in OMG document http://cgi.omg.org/cgi-bin/doc?ad/00-08-01, which is on the OMG web server.

B.2 Presentation of the Examples

This appendix starts with a series of examples, each usually of one source statement in size, of the ASL, AL and Kabira AS languages and concludes with a complete example of an SDL procedure.

The ASL, AL, and Kabira AS examples consist of:

- source text formulated in one (or more) of these languages exhibiting a fragment of a specification,
- an object diagram showing an instance of the Action Semantics model.

Each object diagram gives the meaning of the corresponding specification fragment in terms of the Action Semantics.

These object diagrams show M1 artifacts (i.e. actual user models) by displaying them as instances of classes at the M2 level (the UML Metamodel level). These object diagrams are not M0 objects.

These object diagrams may include model elements from the Core package of UML. These are included to aid in understanding the connection between the diagrams and the surface syntax. Where no confusion is likely to arise, model elements from the Core package are omitted.

This first series of examples illustrates many of the individual model elements of the Action Semantics: control structures (see Section B.3), object manipulation (see Section B.4), and messaging actions (see Section B.5).

For some of the examples there may be no direct equivalent construct in one or two of these languages. In such cases, this does not mean that the operation achieved by the example cannot be achieved in the other language(s), rather it means that it must be achieved by a more explicit and verbose model where the user strings several source language statements together to achieve the desired effect. For example, some language constructs can act directly on collections in some languages but not in others. In this case, the language without the direct support must employ an explicit loop to achieve the same effect. In such examples the resulting object diagram will look very different and so separate examples have been created.

Unlike ASL and AL that have implicit variable declarations and typing, Kabira AS requires explicit declaration that is, for the most part, not shown in the examples here. In the specification, local variables have an association with GroupAction providing for scoping within an action language. The examples do not show this association. Unless otherwise shown, all local variables in these examples have a multiplicity of 1..1.

B.3 Control Structures

These examples are of the traditional control structures and sequential logic present in the action languages. These map to the actions found in Section 2.20, “Composite Actions,” on page 2-228.

If-then-else Logic

The example in Figure B-1 shows a simple if-then-else involving a logical comparison. The semantics of the action language construct is that if and only if the value of `factor` is equal to (the integer) 2, then `some action 1` will be executed. In all other cases, `some action 2` will be executed.

.

ASL:	AL:	Kabira AS:
if factor = 2 then	if (factor == 2)	if (factor == 2) {
# Some action 1	// Some action 1	// Some action 1

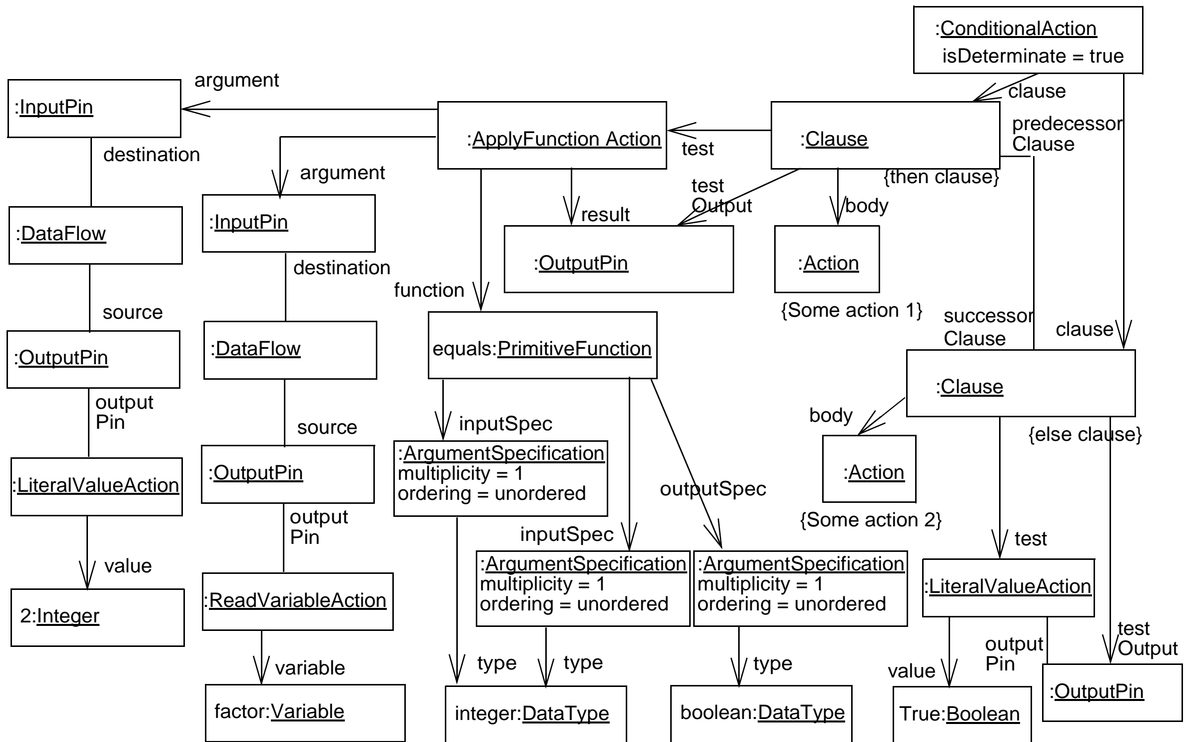


Figure B-1 If-then-else Logic

This maps to a general ConditionalAction within the action semantics. The conditional action has two clauses corresponding to the “then” and “else” branches of the if. These clauses are shown at the top right of the instance diagram. They have dummy actions as bodies for the purposes of this example.

The `then` clause has the actual logical comparison from the source action language attached to it as a `test`. The test action is an ApplyFunctionAction that applies the `equals` function to the two operands of the logical expression. The two operands come from a constants (LiteralValueAction) and a local variable (ReadVariableAction).

The `else` clause has a test action (LiteralValueAction) that always returns the value `TRUE`. The `else` clause is the “successor” of the `then` clause and so will be tested and hence executed only if the `then` clause failed. This arrangement preserves the `if-then-else` behavior of the source languages.

Multi-way Decision

All of the action languages support multi-way decisions in a single statement. ASL supports this through the use of a switch statement where a variable is compared against constant values whereas AL and the Kabira AS support the more general else-if construct.

The example shows a decision based on the value of a local variable realized both as a `switch` statement and as a `else-if` statement.

ASL:	AL:	Kabira AS:
switch factor	if (factor == 1)	if (factor == 1) {
case 1	// Some action 1	// Some action 1
# Some action 1	elif (factor == 2)	} else if (factor ==1) {

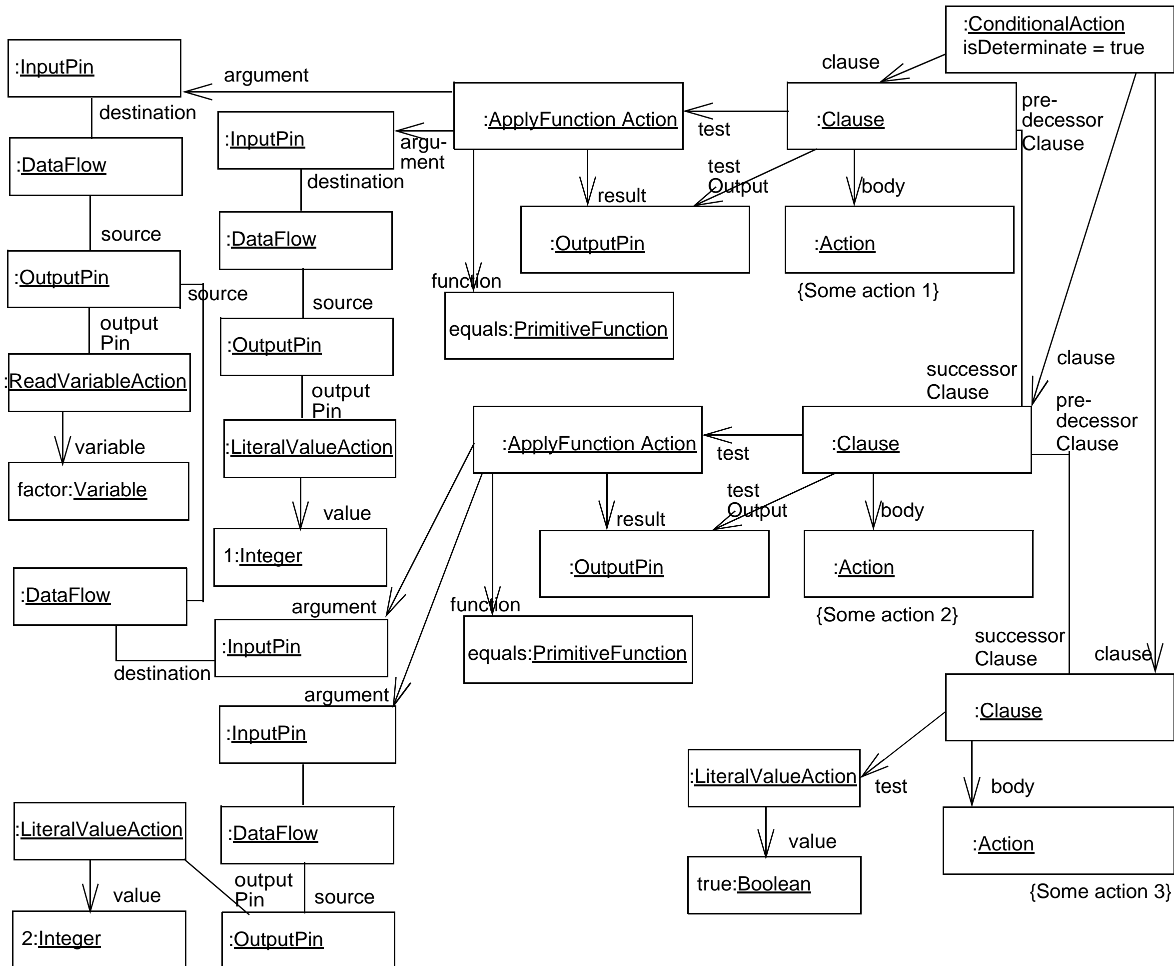


Figure B-2 Multi-way Decision

Note – In ASL, the switch statement has an implicit "break" at the end of every clause so that there is no "fall through" into the next clause. In addition, the execution of the switch terminates once any clause has executed and so the example shown is semantically identical to the else-if examples in AL and Kabira AS.

The details of the substructure of the PrimitiveFunction (input and output types) have been omitted to make the example clearer. The previous example shows what this would have looked like.

Note – The predecessor-successor control flows between the clauses that provide the correct semantics for both the ASL switch and the explicit if-then-else construct. However, an alternative mapping could have been shown in which the test on the third clause is replaced by a logical expression of the form "(factor != 1) && (factor !=2)" in which case the predecessor-successor control flows could have been omitted. This would have allowed all three tests to execute concurrently. However, the logic of the test specification would have meant that only one of the branches could ever execute. This would provide the same semantics as the example as shown.

B.4 Object Manipulation

These examples show the basic creation and manipulation of objects. The actions used by these language constructs are for the most part those described in the chapter on Read and Write Actions. Being local variable oriented languages, all of these operations use local variables of type object reference, or sometimes collections of object references.

Simple Object Creation

Figure B-3 shows an example of creating an object of the class `Customer`. The reference to the newly created object is then assigned to the local variable `new_customer`.

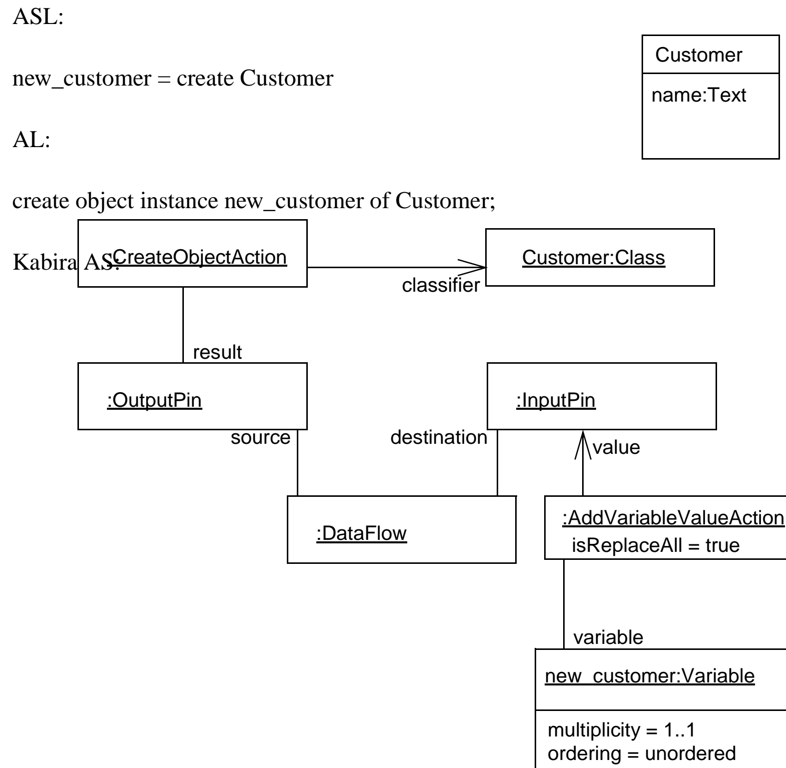


Figure B-3 Simple Object Creation

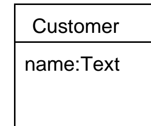
Note – AL actually uses a second form of the class name (the “key letter” captured as a UML tag) to identify the class. This is a minor syntactic detail and to avoid confusion the examples in this Appendix use the class name instead.

Object Creation with Attribute Assignment

Figure B-4 shows an example of creating an object of the class Customer with assignment of the value of the attribute `name` from the local variable `new_name`.

ASL:

new_customer = create Customer with name = new_name



AL:

create object instance new_customer of Customer;

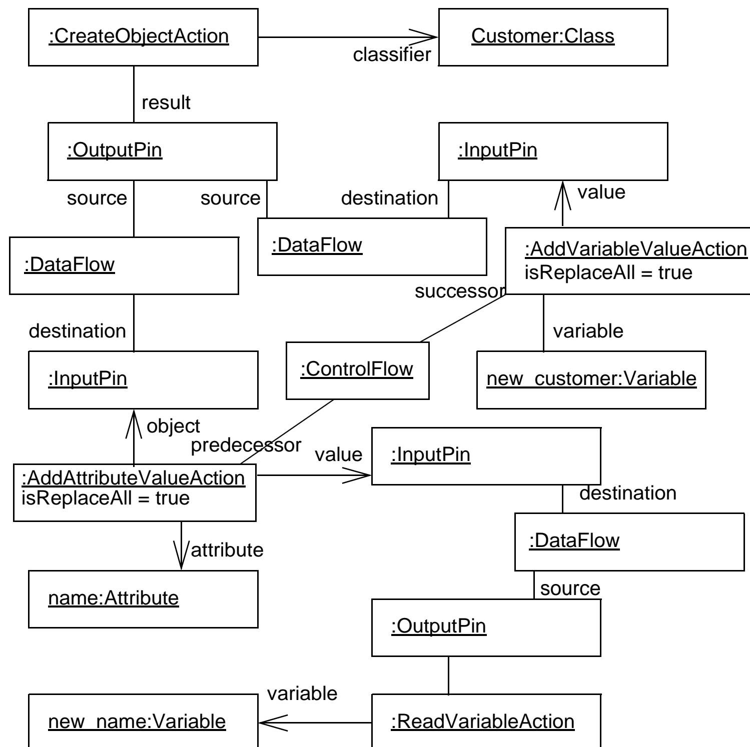


Figure B-4 Object Creation with Attribute Assignment

Note – With the AL example, a strict interpretation would have the attribute assignment via the reading of a local variable rather than through a data flow from the CreateObjectAction as shown. The example is semantically identical to the AL.

Object Destruction

Figure B-5 shows an example of destruction of an object. In the examples, the object is identified by a reference `my_customer`.

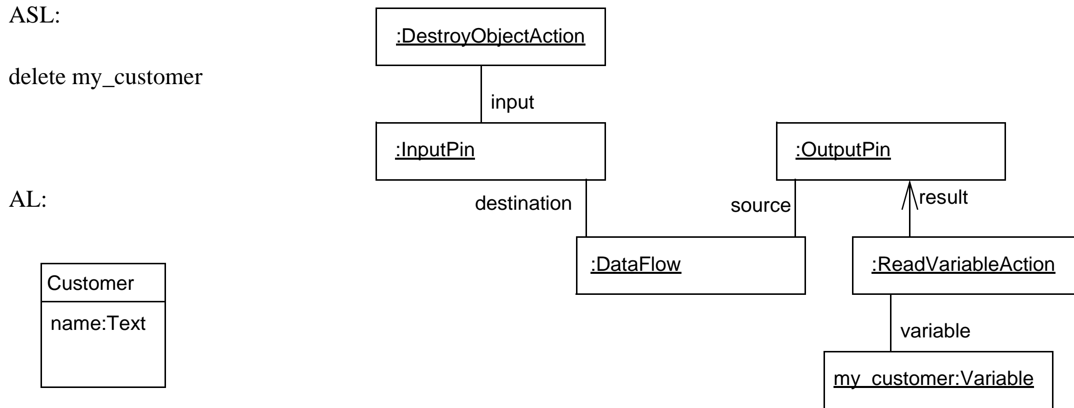


Figure B-5 Object Destruction

Writing of Attributes: Single Attribute, SIngle Object

In this example, a value is written to a single attribute of a single object instance. The value comes from a local variable (`new_balance`), and the object is identified by an object reference local variable (`current_account`).

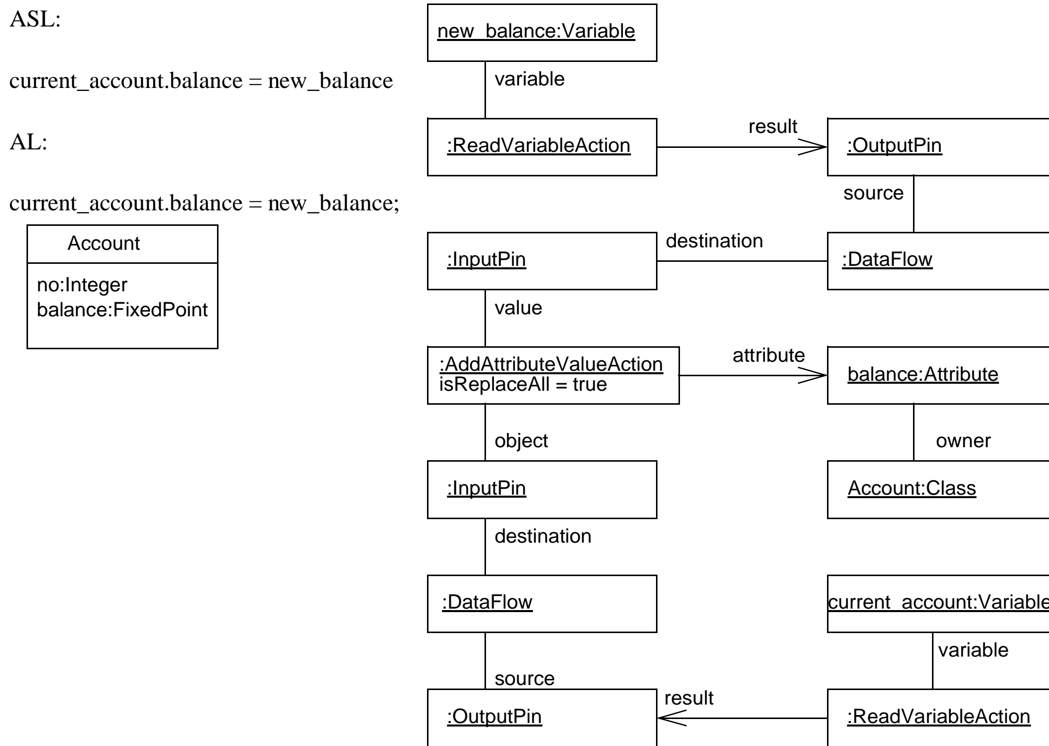


Figure B-6 Writing of Attributes (Single Attribute, Single Object)

Writing of Attributes: Multiple Attributes, Single Object

In this example, a values are written to a multiple attributes of a single object instance. The values come from local variables (`new_balance`, `todays_date`), and the object is identified by an object reference local variable (`current_account`). In ASL this can be achieved through a single language statement, but in AL and Kabira AS this must be achieved through multiple statements, each operating from the same object instance. This example shows ASL only.

.

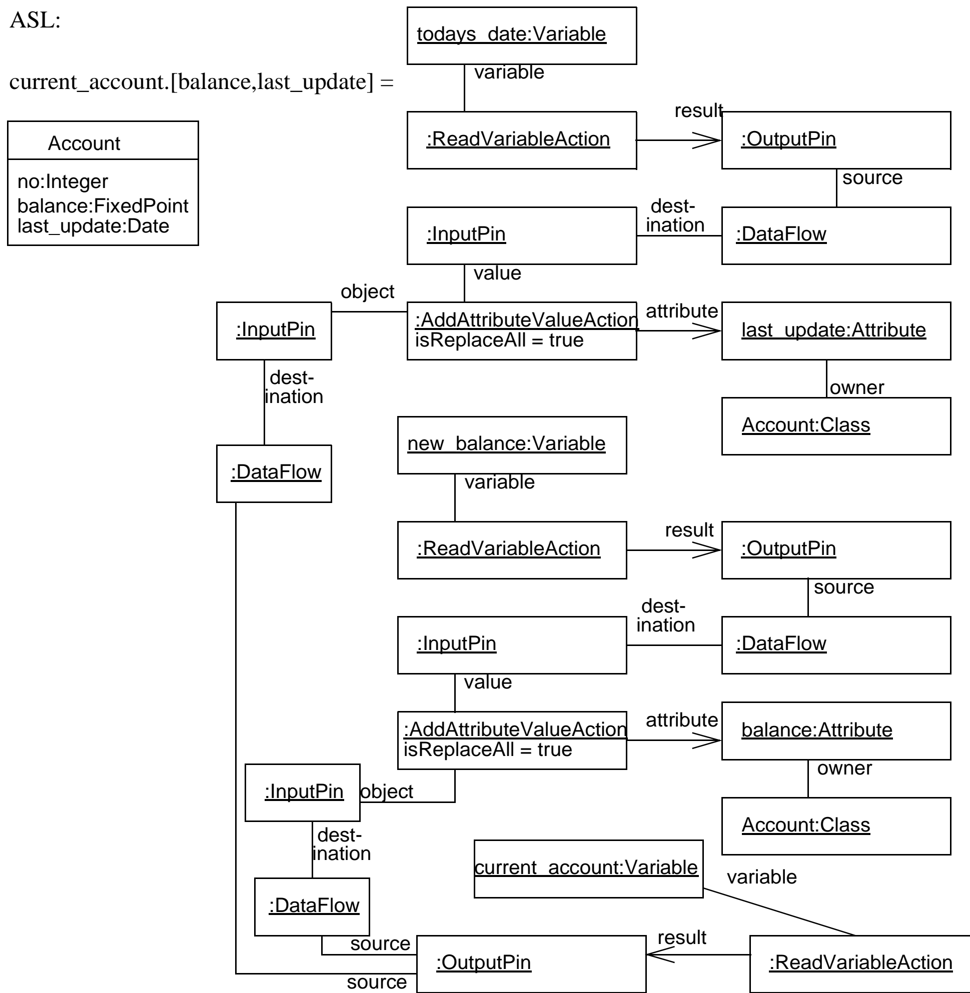


Figure B-7 Writing of Attributes (Multiple Attributes, Single Object)

Writing of Attributes: Single Attribute, Multiple Object

In this example, a value is written to an attribute of a multiple object instances. The value comes from a local variable (`todays_date`), and the objects are identified by an object reference collection local variable (`{reviewed_accts}`). In ASL this can be

achieved through a single language statement, but in AL and Kabira AS, this must be achieved through an explicit loop. The example shows ASL only and explicit loops are shown in other examples

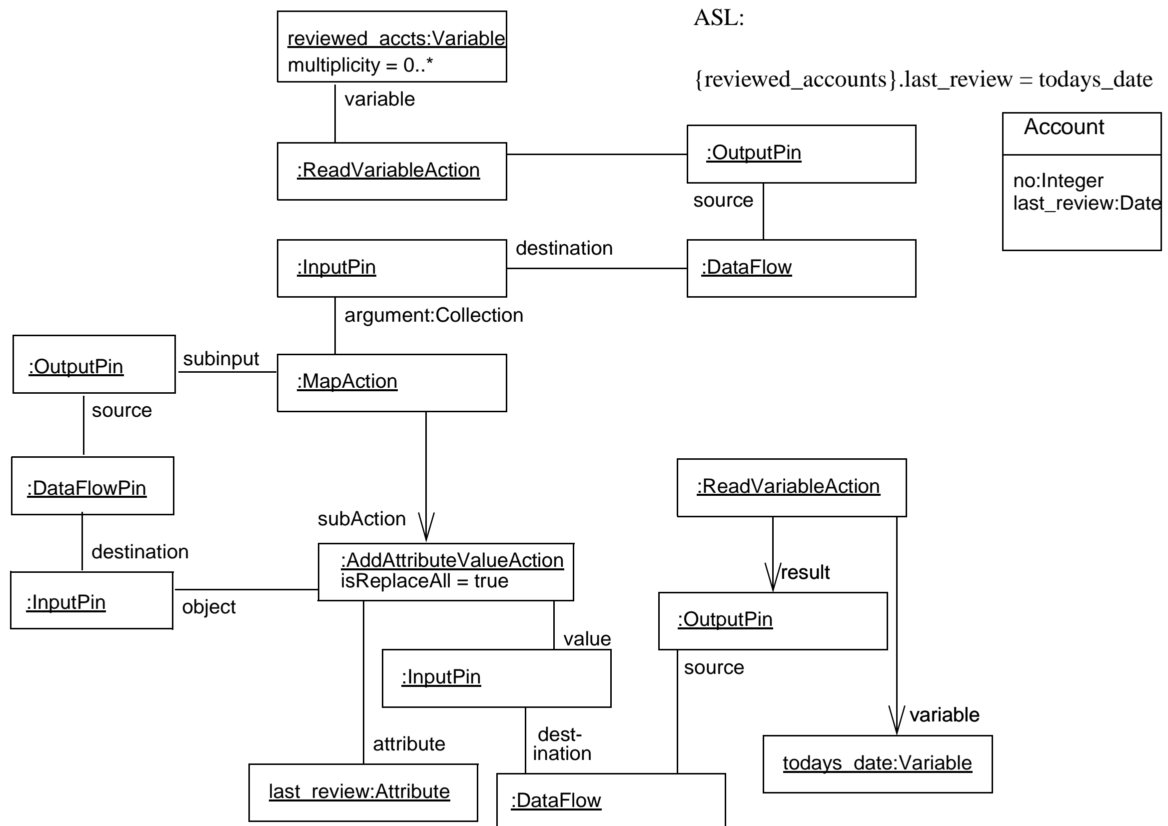


Figure B-8 Writing of Attributes (Single Attribute, Multiple Objects)

Obtaining a Selection of Objects

The ASL, AL, and Kabira AS languages provide facilities to select the extent of a class and store the result in a local variable that can then be used in other language constructs. In ASL and AL, the local variables can be singletons or collections.

In the first example, a selection is made from the `Account` class through to a simple logical condition. The condition results in a single object instance being selected and the reference assigned to a local variable `my_account`.

The top part of the instance diagram concerns the reading of the extent of the `Account` class and flowing the resulting collection into a filter action. The filter action produces an output written to the `my_account` local variable, shown on the right hand middle of the diagram. The remainder of the diagram concerns the subAction of the FilterAction. This is an ApplyFunctionAction that invokes the logical comparison of the two items

in the logical expression. One of the items is a constant (supplied by the LiteralValueAction) and the other is obtained by reading the value of the attribute of the instance being tested.

The second example shows a similar selection, but one that results in a collection that is assigned to a local variable. In this case, because the Kabira AS does not support local variables that are collections, we have not shown an example from this language. In Kabira AS such selections can be used directly as the inputs to loops. In that case the assignment to the output local variable would be replaced by a flow into a LoopAction.

ASL:

my_account = find-only Account where no = 42

AL:

select one my_account from instances of

Account where selected.no = 42;

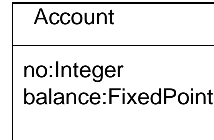


Figure B-9 Single Object Selection

ASL:

{neg_accts} = find-all Account where

balance < 0

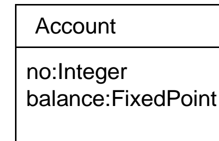


Figure B-10 Obtaining a Collection of Objects

Creating a Link

This example creates a link between two objects. The link is an instance of the binary association shown between the `Account` class and the `Customer` class. The objects are identified by the local variable object references (`the_customer`, `the_account`). In ASL and AL the syntax of the language is such that users must give all associations a

unique name (in this example, `R1`) to provide a unique reference to the association being manipulated. Kabira AS uses the association role for this purpose. Semantically, each of the three languages achieves the same effect.

In a reflexive association, it is necessary to know in which direction the link is intended. All three languages use role names to clarify this.

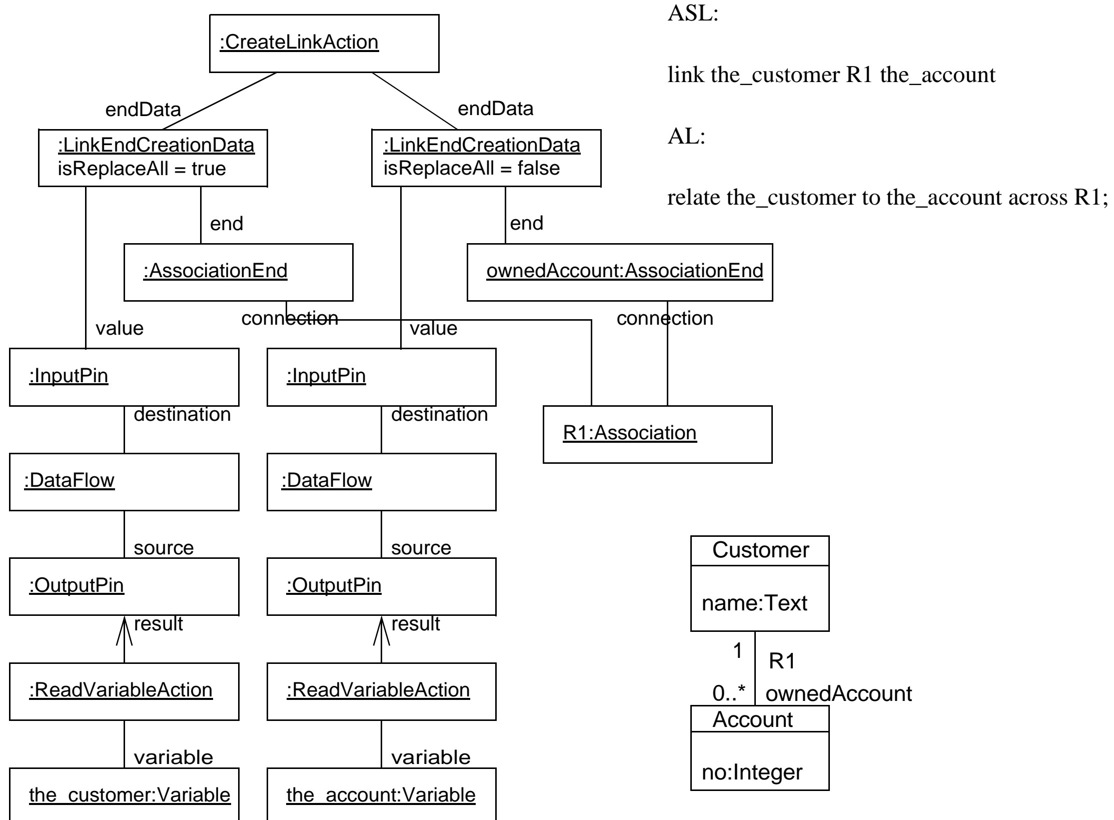


Figure B-11 Creating a Link

Note – This example shows a 1:0..* association. This means that there can be only one linked Customer for a given Account. In the Kabira AS, the semantics are such that if a Customer object was previously linked when the “relate” statement was executed, then an implicit “unrelate” (DestroyLinkAction) is performed prior to the CreateLinkAction. There can, however, be many Account objects for a given Customer object. The pattern of “isReplaceAll” values in the two LinkEndCreationData objects achieves this effect. In ASL, the semantics are slightly different. In that language, the modeler must ensure that any existing instance of Customer is explicitly unlinked from the instance of Account before the link is executed. Any failure to do this is regarded as an exception condition due to the violation of the multiplicity of the association. For ASL, therefore, a more accurate mapping of the semantics of “link” would have “isReplaceAll” false in both the instances of LinkEndCreationData.

Destroying a Link

This example destroys a link between two objects. The link is an instance of the binary association shown between the `Account` class and the `Customer` class. The link to be deleted is identified by the objects at either end that are identified by local variable object references (`the_customer`, `the_account`). In ASL and AL the syntax of the language is such that users must give all associations a unique name (in this example, `R1`) to provide a unique reference to the association being manipulated. Kabira AS uses the association role for this purpose. Semantically, each of the three languages achieves the same effect.

In a reflexive association, it is necessary to know in which direction the link is intended. All three languages use role names to clarify this.

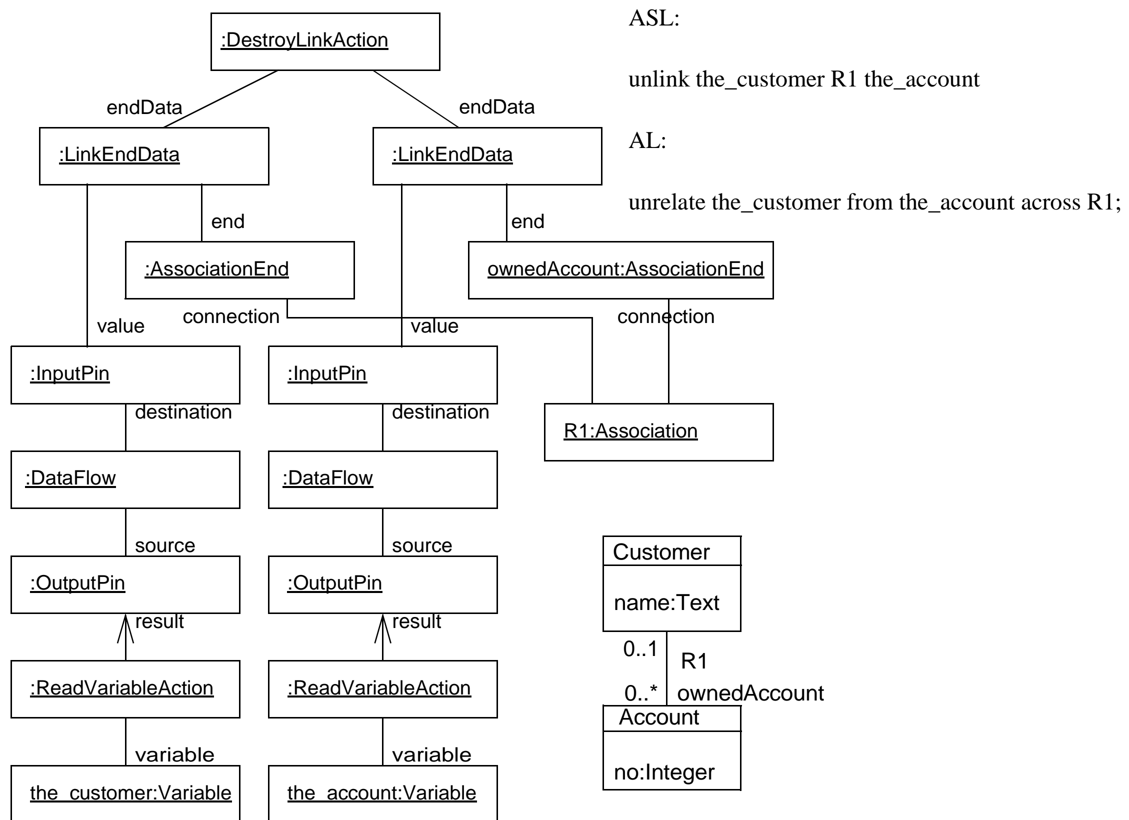


Figure B-12 Destroying a Link

Navigating an Association to a Single Object

These action languages navigate an association to obtain the object references of linked objects, which are then used to perform some manipulation on the linked objects, such as calling an operation provided by the object.

In this example, a navigation starts from an instance of `Account` (specified by the object reference `the_account`) to obtain a reference to the Customer that owns it. The resulting reference is in a local variable called `owner`. Both ends of the association involved in the navigation are specified, but only one of them has an input pin. This pin takes the identity of the object at the starting end of the navigation.

In the model fragment shown, due to the multiplicity of the association, there will be only one instance of Customer obtained by the navigation.

ASL:

owner = the_account -> R1

AL:

select one owner related by

the_account -> Customer[R1];

Figure B-13 Navigating an Association

Navigating an Association to Multiple Objects

These action languages navigate an association to obtain the object references of linked objects, which are then used to perform some manipulation on the linked objects, such as calling an operation provided by the object.

In this example, a navigation starts from an instance of `Customer` (specified by the object reference `the_customer`) to obtain references to all the Accounts owned by Customer. Due to the multiplicity of the association in the example, the result will be a collection.

In ASL and AL, local variables can be collections and so the example and mapping is very similar to those in the previous example (of navigation to a single instance). In Kabira AS, local variables cannot be collections and so the result of the navigation

must be used directly in an explicit loop. However, the effect of this loop is exactly that of a MapAction in the action semantics. The action enclosed in the Kabira AS loop is repeatedly executed with the local variable (the_account) being successively given a value corresponding to each instance in the collection of object references obtained by the navigation. This mapping is shown in the second diagram.

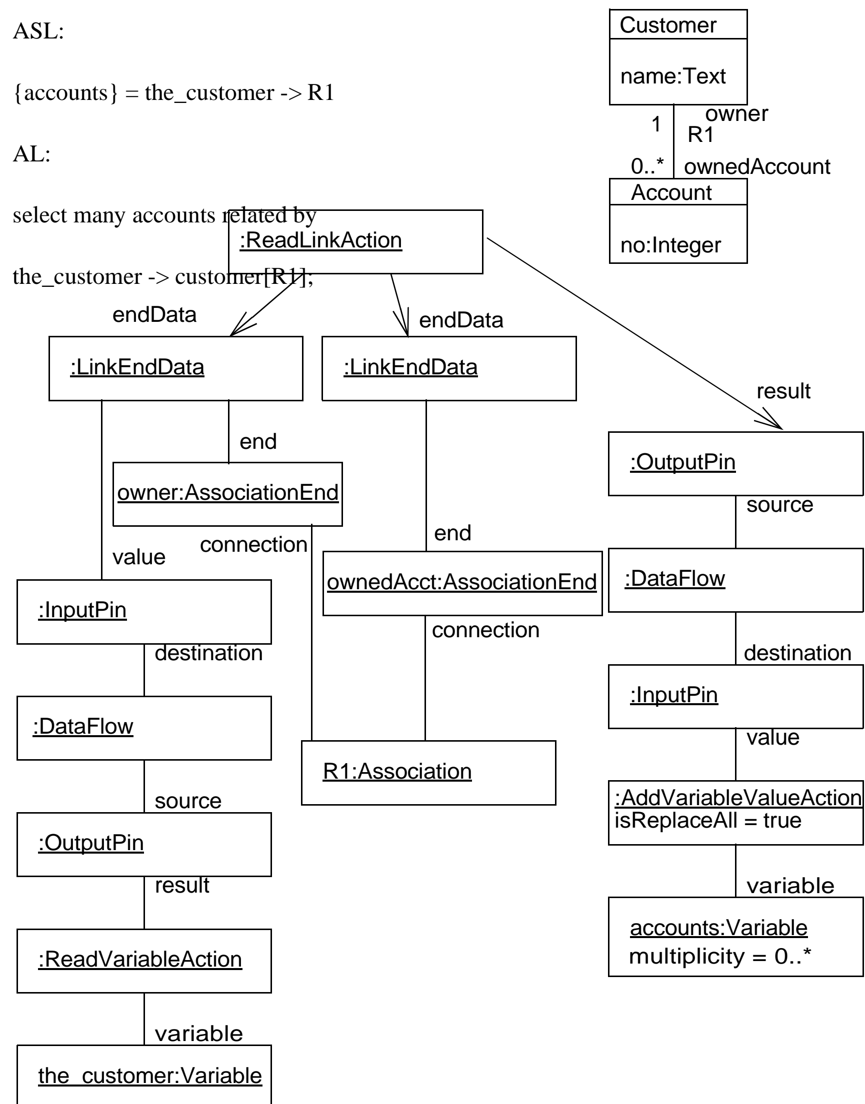


Figure B-14 Obtaining a Collection by Navigation

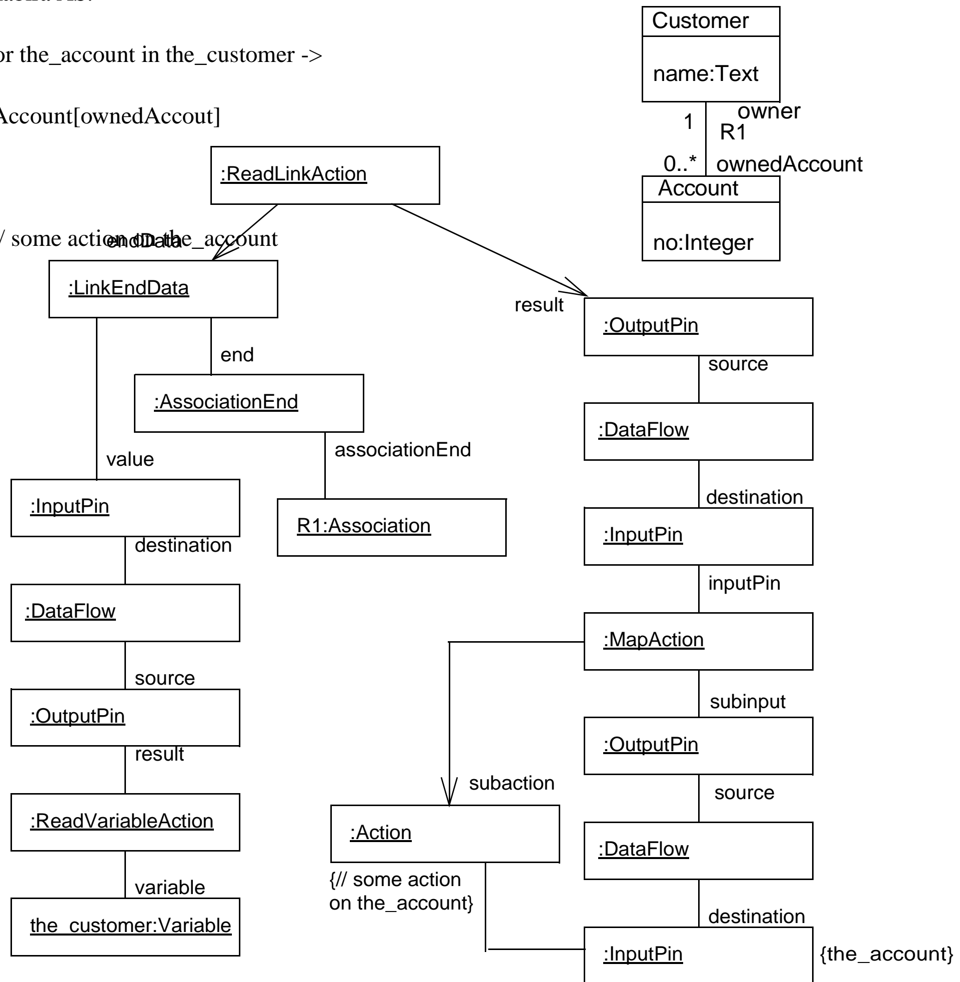


Figure B-15 Loop over a Navigation

B.5 Messaging Actions

This section covers the invocation of operations and the sending of signal events.

Invocation of an Instance Operation with no Parameters

In this example, an operation (validate) provided by the `Customer` class is invoked. The operation has no parameters, but applies to a specific instance of Customer (identified by the object reference local variable `my_customer`).

Note – The ASL syntax supports a particular syntax for the names of operations that makes their association with the class of the target object clear. This has not been used here to make the example straightforward.

ASL:

[]= validate[] on the_customer

AL:

the_customer.validate();

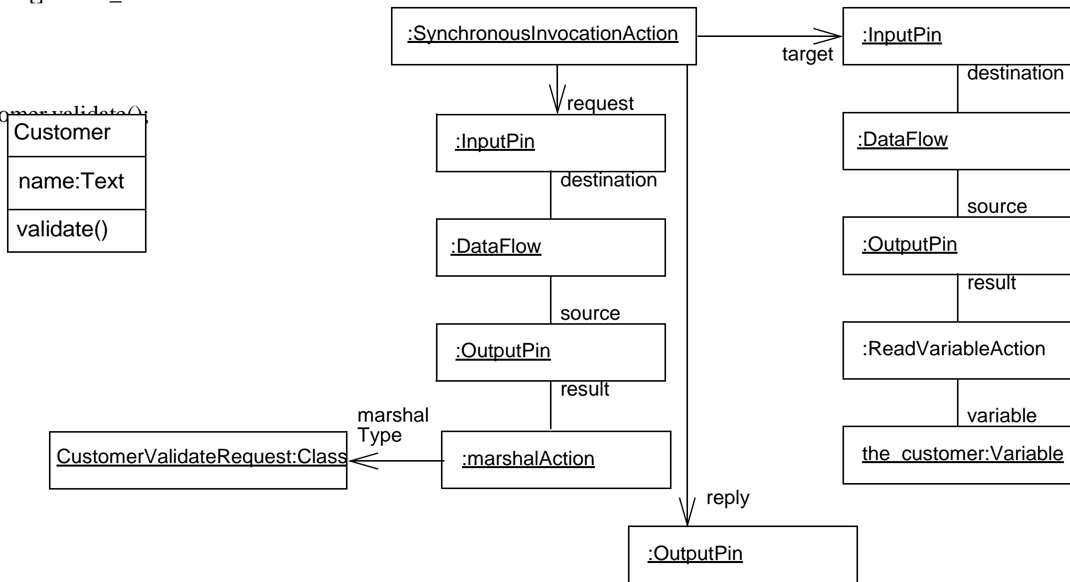


Figure B-16 Simple Operation Invocation using SynchronousInvocationAction

The example shows only the invocation actions and not the Effect Resolution. However, in all three languages the resolution is a simple operation lookup.

Note – In this example, there are no return parameters from the validate operation and so the output pin can be ignored, or it could be used as an input to a subsequent action in lieu of a control link.

The above mapping uses the SynchronousInvocationAction. As is discussed in Section 2.24, "Messaging Actions," on page 2-311; however, an alternative action, the CallOperationAction is also available. With this action, the semantics of marshalling data are implicit in the action. This alternative mapping is shown in Figure B-17.

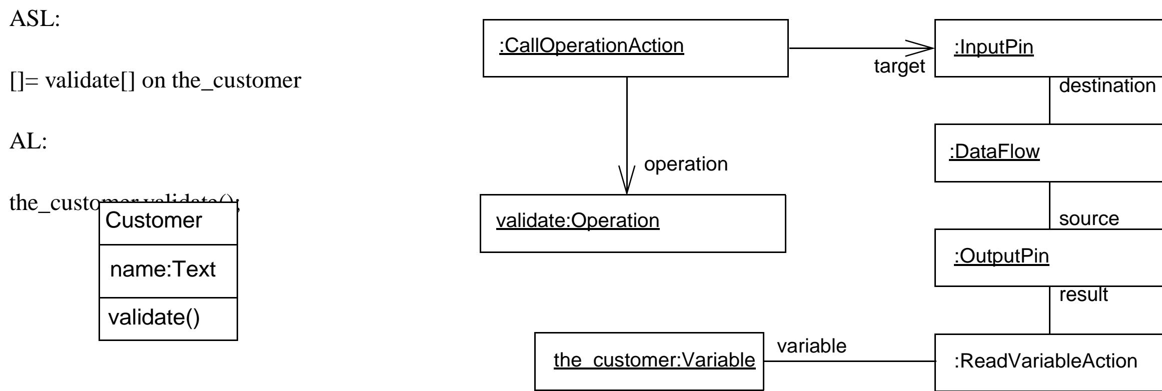


Figure B-17 Simple Operation Invocation using CallOperationAction

Invocation of an Instance Operation with Parameters

ASL:

[new_balance]= update_balance[amount] on the_account

AL:

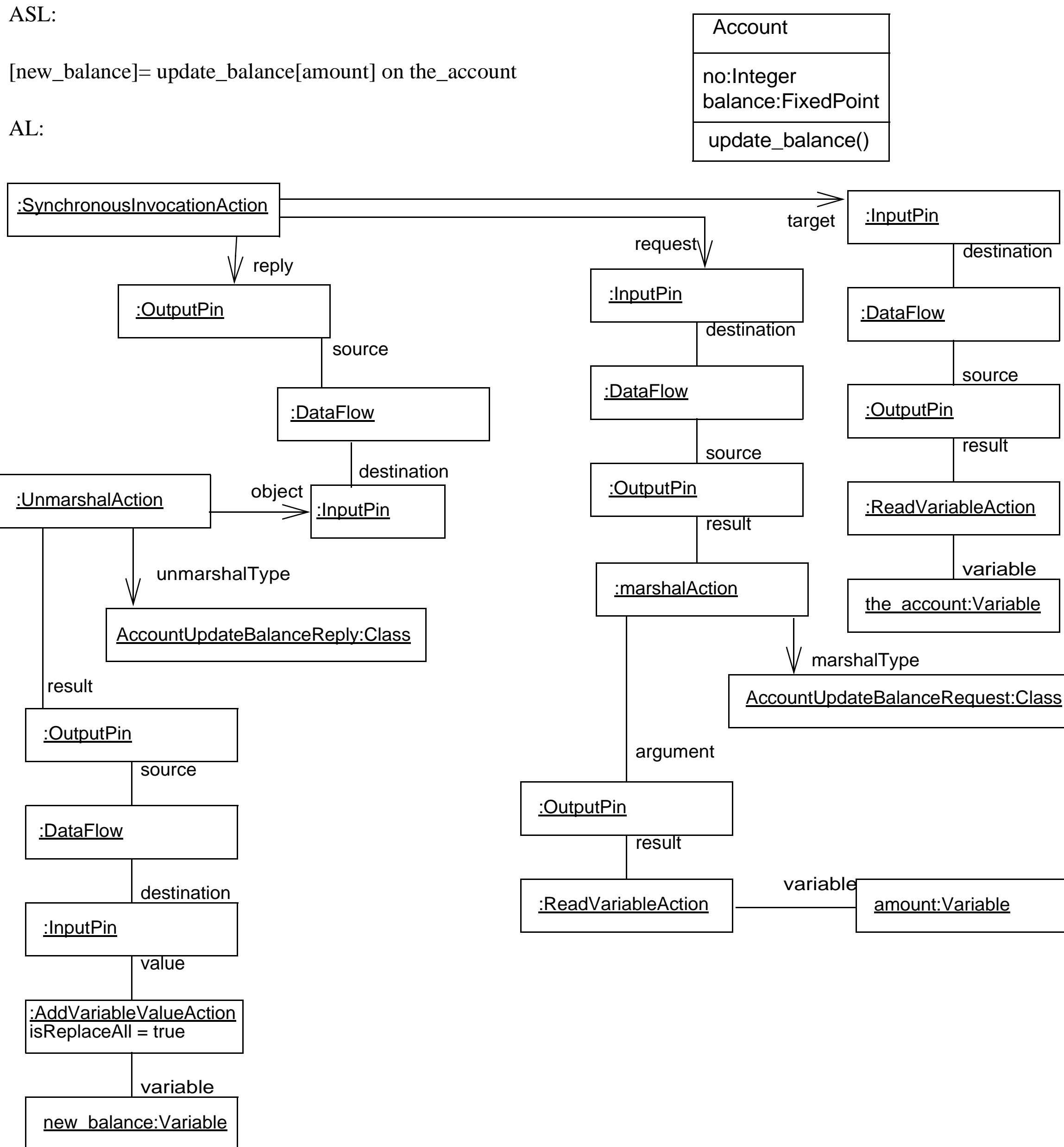


Figure B-18 Operation with Parameters using SynchronousInvocationAction

This example shows the invocation of an operation that takes an input parameter (amount) and updates the balance of an Account. The resulting balance is returned as an output parameter. In Figure B-18 this is shown using the SynchronousInvocationAction with explicit marshalling and unmarshalling of parameters. The equivalent mapping with CallOperationAction is shown in Figure B-19.

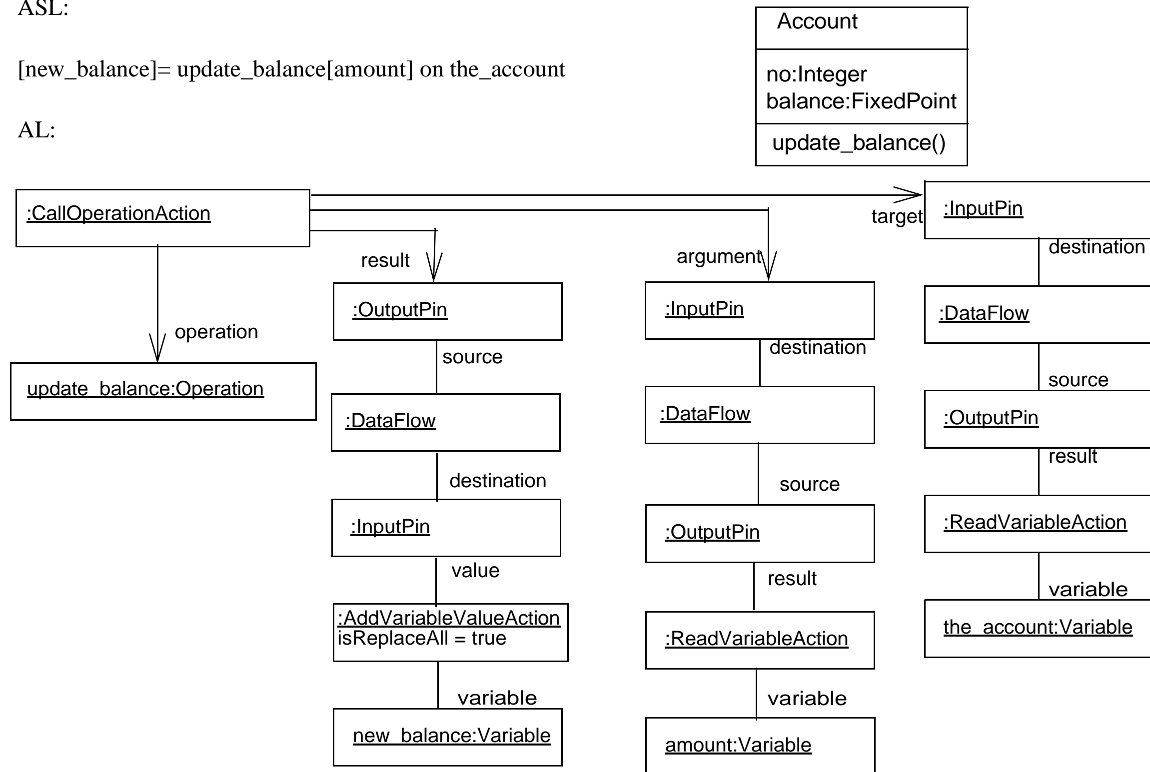


Figure B-19 Operation with Parameters using CallOperationAction

Sending of a Signal Event with no Parameters

This example shows the dispatching of a Signal Event to an object of the Account class.

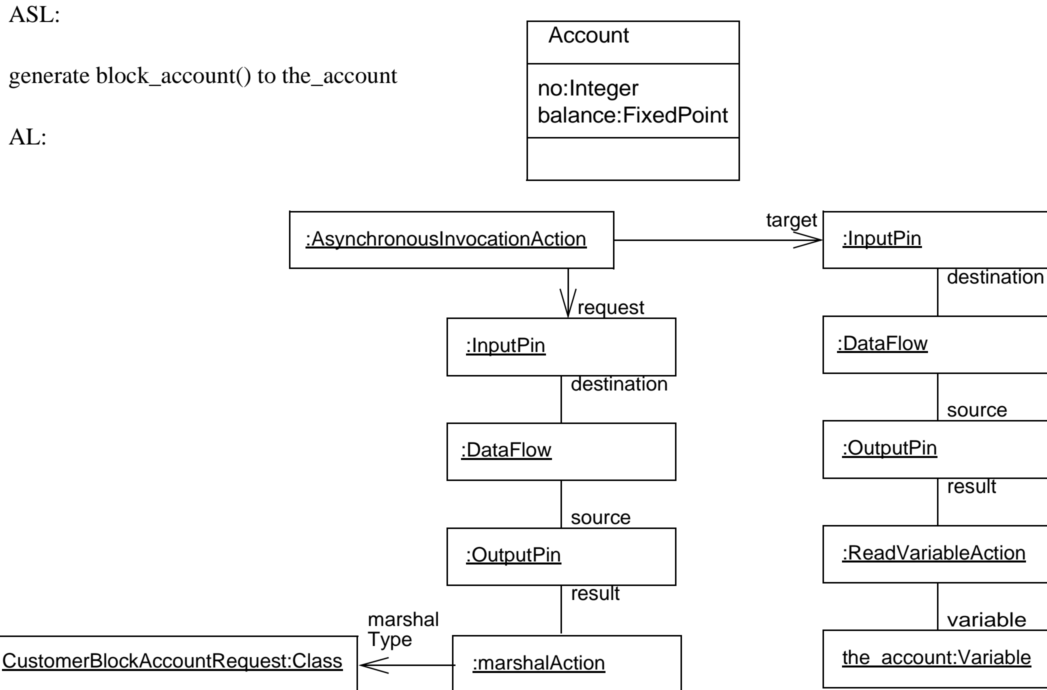


Figure B-20 Sending of Signal with no parameter using AsynchronousInvocationAction

In a similar way to the mapping of operation invocations discussed in the previous section, an alternative mapping for this uses the SendSignalAction, where the marshalling of data is implicit. This mapping is show in Figure B-21.

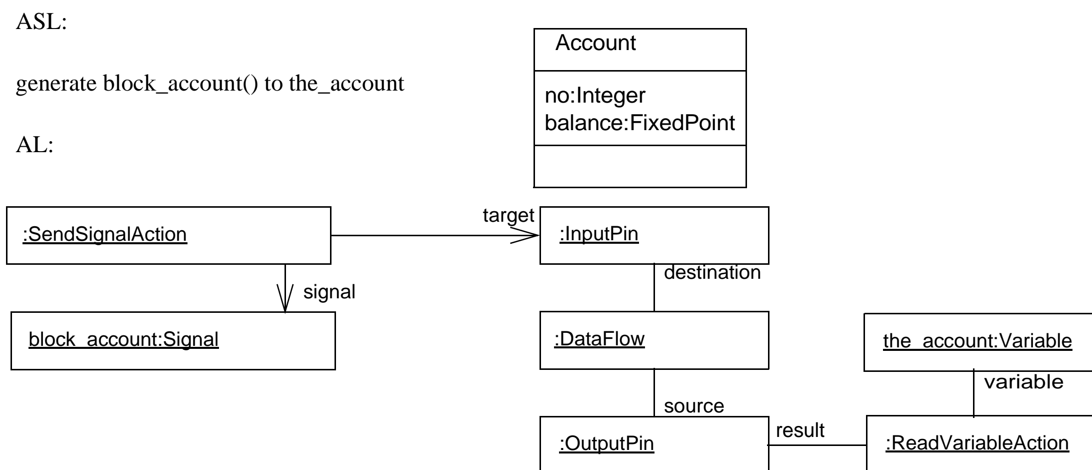


Figure B-21 Sending of Signal with no parameter using SendSignalAction

Note – The ASL and AL syntax actually requires a particular syntax for the names of signals that makes their association with the class of the target object clear. This syntax has not been used here to make the example more straightforward.

Sending of a Signal Event with Parameters

In a similar way to the previous example, this sends a signal to an object. In this case there is an additional parameter (period) that is sent with the signal.

ASL:

generate update_interest(period) to the_account

AL:

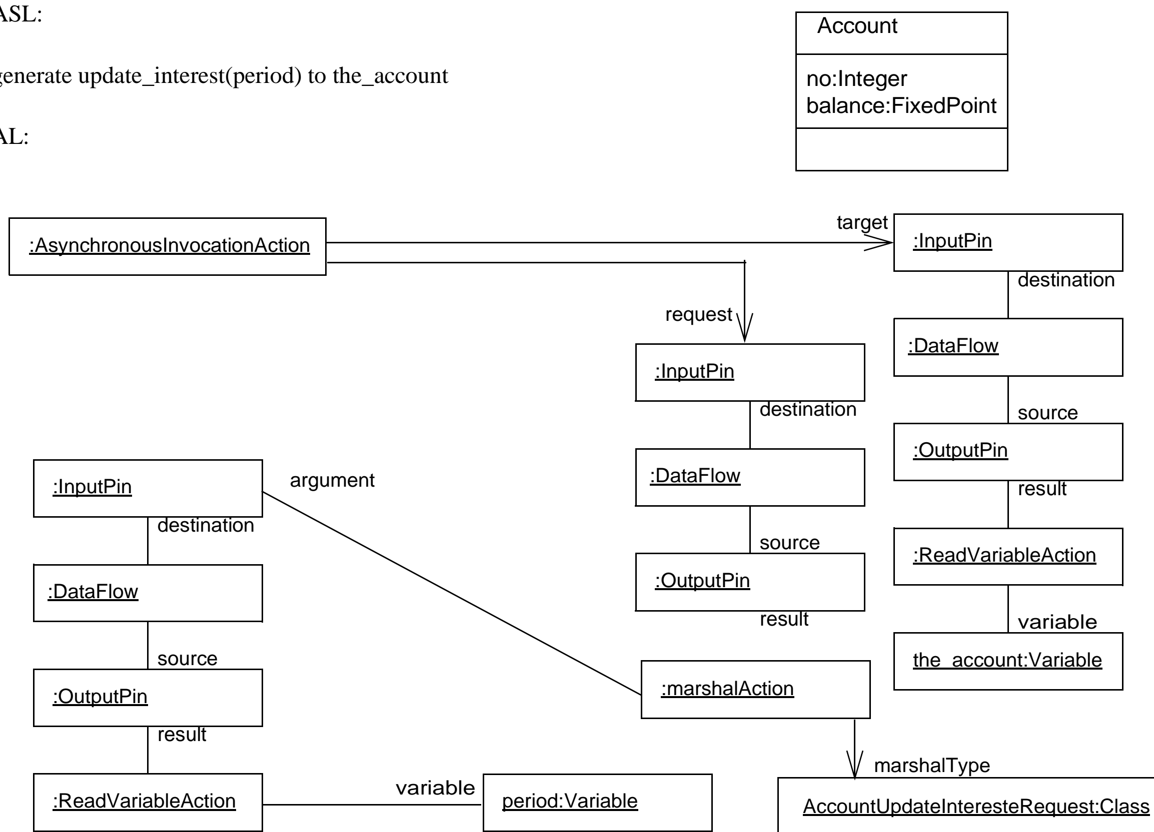


Figure B-22 Sending of Signal with Parameters using AsynchronousInvocationAction

Finally, the same example using the SendSignalAction is shown in Figure B-23.

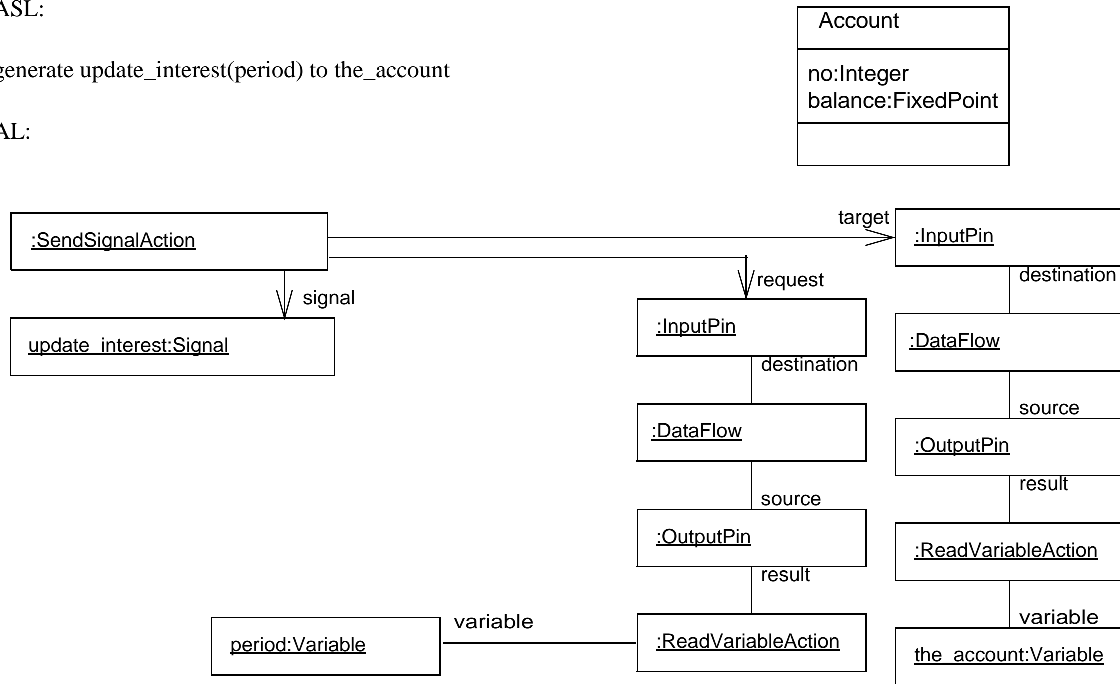


Figure B-23 Sending of Signal with Parameters using SendSignalAction

B.6 Complete Example: The FFT

This section shows a complete example of the specification of a procedure using actions. The purpose is to show how the various actions fit together to carry out a complete algorithm, as well as to illustrate some of the complicated actions through actual use. An informal notation is used to illustrate the examples, emphasizing the connectivity of the actions while suppressing minor mechanical details. This notation is not complete and is not intended to be a normative syntax, but merely to help get an intuitive feel for the action semantic constructs without becoming immersed in UML minutiae.

The example describes the Fast Fourier Transform (FFT), one of the most important algorithms discovered to date, both for its theoretical and practical consequences. This algorithm has revolutionized signal processing applications, and it has consequences for polynomial multiplication and other areas. Theoretically, its discovery showed that many algorithms could be much more efficient than intuition would suggest and led to major advances in complexity theory. The algorithm itself is also elegant both in theory and in implementation, with extreme malleability that allows it to be adapted in many different ways. In particular, it lends itself to a highly parallel implementation, which can be captured using these action semantics. The ability to preserve inherent concurrency is a major contribution of these action semantics.

This section does not attempt to explain the theory or derivation of the FFT. The reader is advised to consult a modern book on numerical algorithms for more information. Note also that the form of the algorithm presented here is not the most efficient form

used for computation. Even in algorithm books, the algorithm is usually presented in a more transparent form first, because the highly optimized forms can be tricky to understand. Such optimizations could, of course, be expressed in the action semantics.

B.6.1 The Fast Fourier Transform

The Discrete Fourier Transform (DFT) performs a complex-number transformation from the time or spatial domain into the frequency domain, according to the following formula:

$$B_j = \sum_{k=0}^{n-1} A_k W_n^k \qquad j = 0, 1, \ldots, n-1 \tag{EQ 1}$$

where n is a power of 2 and W_n is the primary complex root of unity of order n:

$$W_n = e^{-2\pi i/n} \tag{EQ 2}$$

Intuitively, this would seem to require $O(n^2)$ arithmetic operations (multiplications and additions) to compute. Remarkably, the Fast Fourier Transform (FFT) algorithm can perform this transformation in $log_2\, n$ stages of $O(n)$ operations, for a total complexity of $O(n \log n)$. Among many forms, the algorithm can be expressed in the following form:

$$S_{0,j} = A_j \tag{EQ 3}$$

$$S_{m+1,2j} = S_{m,j} + S_{m,j+n/2} \qquad S_{m+1,2j+1} = S_{m,j} - S_{m,j+n/2} V_{m,j} \qquad j = 0, 1, 2 \ldots, n/2 - 1 \tag{EQ 4}$$

$$V_{m,j} = W_n^{\lfloor j/2^m \rfloor 2^m} \tag{EQ 5}$$

$$B_j = S_{\log n,\, rev(j,n)} \qquad j = 0, 1, 2 \ldots, n-1 \tag{EQ 6}$$

where $rev(j,n)$ is the integer obtained by reversing the $log\ n$-bit binary representation of the integer j; for example, rev(6,8) is 3.

This formula can be understood as follows: There are $log\ n$ stages, each of which transforms a complex vector of n elements into another complex vector of n elements (Equation 4). The starting vector is the original vector to be transformed (Equation 3). During each stage, the pattern of computation is the same, except the multiplication factors V are “thinned” by a factor of 2 on each successive stage (Equation 5). The final complex vector is rearranged by swapping each element to its “bit-reversed” position to obtain the final complex vector result (Equation 6). These $log\ n$ stages must be performed sequentially; there is no concurrency across them.

Looking within each stage, Equation 4 indicates that two elements from one stage are used to compute two elements of the next stage. This pair of formulas is called a butterfly operation, after the shape of its data flow graph, which uses two input values to produce two output values. All $n/2$ butterfly operations can be performed in parallel, because there is no interaction among the input or output values of different ones. The FFT algorithm permits a high degree of concurrency. Examining the formula closely,

we see that each butterfly operation takes an element from the first half of the input vector and the corresponding element from the second half of the input vector, producing two successive values in the output vector for the next stage. There is no interaction between the n/2 butterfly operations in a stage, either on input values or output values. A stage can be viewed in vector terms as follows: A vector of *n* elements is cut in half (like a pack of cards) into two vectors of n/2 elements each. In parallel, each element of one half is combined with the corresponding element of the other half in a butterfly operation, yielding two values. If we form two half-vectors from the results, they must be shuffled together by alternating elements from each of the half-vectors (again, like a pack of cards) to form a full-size vector for the next stage of the computation. To summarize a stage: cut a full vector into two half vectors, map the butterfly operation concurrently onto each pair of half vectors to form a new pair of half vectors, and shuffle the two half vectors together to produce a new full vector.

There is one final detail. The multiplication factors change each stage. On the first pass, they are the full set of roots of unity. After each pass, the number of distinct values is reduced to form runs of length 2^m, where m is the pass. We call this a vector "thinning" operation, which forms runs by duplicating the first value of each sequence.

B.6.2 Illustrative Notation

Figure B-24 shows the action semantics constructs representing the FFT algorithm.

The notation represents an object diagram with some structural details suppressed or expressed as text. Actions are shown as rectangles; the kind of action is shown by a label (such as "LoopAction"), with an optional name and colon to label individual action instances that are part of other actions (for example, the Clause contains a GroupAction with the rolename "body"). Rectangles with names but no action names are ApplyFunctionActions; the name is the name of the applied function. Rectangles with values in them are LiteralValueActions. Nesting of actions indicates ownership of the contents by the containing action. Small squares represent pins. An input pin is a hollow square and an output pin is a filled square. Input and output pins of an action are placed on its outer border. Argument and result pins of a procedure are placed on its border (because they are viewed from the viewpoint of the procedure contents). Pins that are lined up inside it are meant to be an array of pins; a text label applies to the entire list. For example, "loopVariable" represents the array of 4 output pins near the top of LoopAction; the array of pins is a part of LoopAction with the rolename "loopVariable". (This obviously would not be precise enough for a complete notation.) Arrows from output pins to input pins represent data flow relationships.

There is one important action semantics construct that is not based on containment. A clause references an output pin within its embedded test action and an array of output pins within its embedded body action. These pins are owned by the embedded actions, not the clause. This reference is shown in the diagram by a hashed square within the clause connected by a solid line to the appropriate pins within the embedded actions. The square is hashed to indicate that it is not an actual pin but merely a reference. For example, the hashed pin in Clause labeled "testOutput" represents an association from Clause to the test Action with the rolename "testOutput." Clause does not duplicate the pin or own it directly.

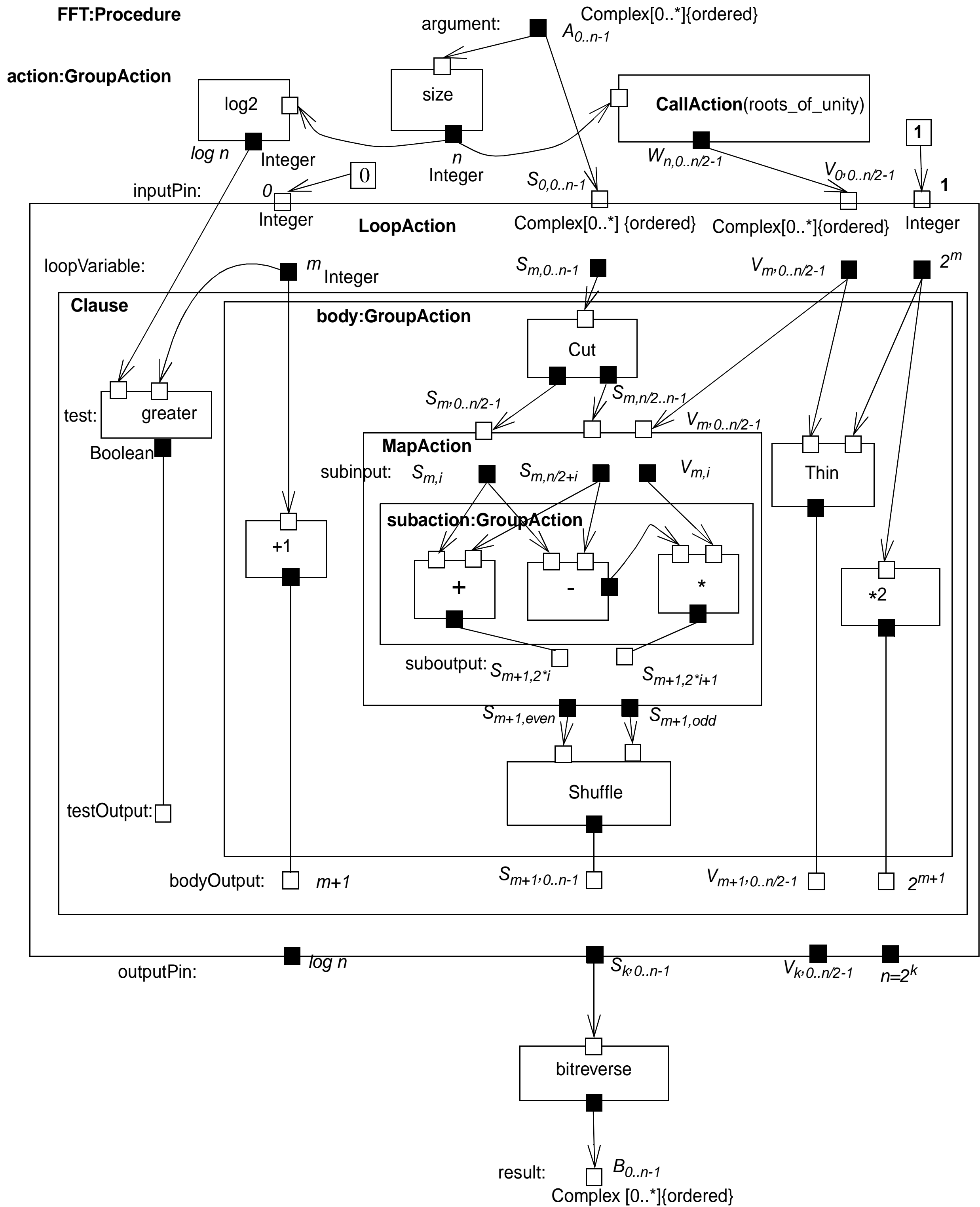


Figure B-24 FFT Algorithm

The consequences of various elements in a construct is explained in informal text. For a precise definition, consult the model and class descriptions. The purpose of this section is to show how the constructs might be used to build larger structures. The notation is suggestive, not precise, and is not meant to be normative.

Many pins have names in italics next to them, such as $A_{0..n-1}$. These are not UML constructs and do not appear in the model. They are merely labels corresponding to the mathematical equations to permit describing the pins in this discussion.

B.6.3 Discussion

The overall construct in the diagram is the procedure FFT. In a UML model, this procedure would be attached to an operation on a class as a method. This procedure takes one argument, a complex vector, and produces one result, another complex vector. The argument is modeled in the procedure as an output pin. This might seem strange; after all, the value is an input to the procedure. The mystery is explained because the argument is viewed from inside the procedure, where it appears as a value available to be used. From the viewpoint of the outside of the procedure, the value is an input, but from the inside, it is an output. Crossing the boundary changes the polarity, and we describe procedures from the inside.

The size of the vector must be a power of two. This could be expressed as a UML constraint, but it is not shown in the diagram. What if it is not a power of 2? Then the model is in error, and its semantics are undefined. Eventually exceptions will be added to the action semantics. When they are present, the procedure could contain a conditional action that would raise an exception on failure of the condition. However, it is permitted to have procedures that do not verify their preconditions, and such verification would be the responsibility of the entire model.

The argument value is used in two ways, as shown by the two arrows leaving the output pin *A*. Both arrows represent the same data value. One copy is used as an input to the loop action (the initial value of the loop variable *S*). The other copy is an input to the `size` PrimitiveFunction action. This action outputs the size of a collection. In some implementations, collections may be resolvable into simpler objects, but the implementations vary widely, and actions on collections can be mathematically defined, so they can be treated as primitive functions.

The size of the array, *n*, is also used in two ways. It is an input to the `log2` primitive function that outputs the base-2 logarithm of the size (remember that n must be a power of two). Why do we treat `log2` as a primitive function and not as an operation? It is easily defined mathematically. Moreover, it might well be directly implemented in hardware (square root is a built-in instruction in most floating point chips today). In any case, the algorithms to compute it are purely implementation and should not be included in a semantic specification of behavior. The purpose of specifying a behavior with action semantics is to understand its inherent constraints, not to make arbitrary implementation decisions that overspecify the behavior.

The other copy of the integer *n* is an argument to a call to the operation `roots_of_unity`. This operation returns the *n* complex roots of unity as a complex vector. The specification of this operation is given by Equation 2.

The main work of the algorithm is performed by a LoopAction. Two of the inputs to this action are the original argument vector and the vector of roots of unity. Within the loop, these vectors are loop variables that are recomputed each iteration. The other two inputs are the integers 0 and 1. These values initialize the loop variables m and 2^m. The loop variables are initialized by the inputs, but the loop variables are distinct from the pins that initialize them. The values of the loop variables are recomputed each iteration. If a value from outside the loop is used directly inside a loop, then its value is fixed over all the iterations of the loop. For example, the test action within the loop clause has *log n* and *m* as inputs. The former value is fixed during the loop. The latter value is a loop variable and it changes during each iteration. Obviously, at least one input to a loop test must be a loop variable or the loop will never terminate!

The test and body actions of a loop are sequential. The test must succeed before the body can be executed, and the body must complete before the test can be performed again on the updated loop variables. Within the loop body, however, there is a lot of concurrency. The actions `+1, thin, *2,` and the `cut-map-shuffle` sequence can all be performed concurrently. Expressing such concurrency is one of the main purposes of the actions semantics. This does not mean that an implementation must implement the concurrency using parallelism. It merely means that the implementation does not contain arbitrary constraints.

The primitive actions `+1` and `*2` perform simple unary arithmetic operations. These could be defined as primitive functions. The primitive action `thin` is a primitive vector function. It takes a vector and an integer *t* and copies every *t*'th value from the input array into a run of length *t*, so that the output vector is the same size as the input vector but has fewer unique values in it. This action could obviously be defined as a procedure, but any particular implementation is arbitrary.

`Cut` and `shuffle` are also operations on vectors. `Cut` takes a vector as input and produces two vectors as output, one of them the first half and the other the record half of the input vector. `Shuffle` takes two vectors as input and interleaves their values to produce a single vector containing all the values. These operations could be defined as loops, but by defining these operations as primitive functions we preserve the possibility of a highly parallel implementation (which depends heavily on the exact implementation).

The map action is probably the most complex action in the action semantics. In this example, it takes 3 input vectors, each of size *n/2* elements. All the vectors must be the same size, but they could contain values of different types (in this case, however, they all contain complex vectors). A tuple comprising one value from the same position in each vector is called a *slice*. The input vectors contain *n/2* slices (remember, they must all be the same size). Each slice is the input to a separate execution of the subaction GroupAction. The executions of the subaction are all concurrent. Note the 3 output pins at the top of the subaction. On each execution, each pin gets a value from the corresponding input vector. For example, $S_{m,i}$ is a value from the vector $S_{m,0..n/2-1}$. There is no explicit data flow from the inputs to the map variables. The connection is implicit, part of the definition of the map action, and the pins are of different rank in any case (one is a collection, the other a scalar). In any case, it does not represent a

single data flow. Rather, a loop represents an unbounded sequence of repetitions of its contents, each implicitly connected to the previous repetition by data flows. A loop is a finite representation of this infinite graph.

Each execution of the map subaction GroupAction performs 3 arithmetic operations to yield two complex values as output. Note the two hatched pins within the MapAction labeled `suboutput`. These are not actual pins owned by the MapAction. In the notation, they represent associations from the MapAction to output pins owned by the embedded subaction. They designate the outputs of the map action, but there is no need to physically copy the values to new output pins.

Each execution of the map subaction yields a pair of output values, one for each slice from the inputs. The output values are assembled concurrently into two output vectors, equal in size to the input vectors. The map action applies a subaction concurrently to each of the slices of the input to produce slices of the output. The output need not have the same number of vectors as the input, but each vector must be the same size.

The two output vectors produced by the map action are then shuffled together to produce a single vector of size n. This vector and the other three outputs of the loop action update the loop variables for the next iteration.

When the loop variable m is finally equal to $\log n$, the loop test fails and the values of the loop variables are copied to the output pins of the loop action. Most of the output pins of the loop are ignored (they have served their purpose inside the loop), but the vector S serves as input to the `bitreverse` operation. This is another primitive function on a vector, defined by Equation 6. The output of this action, a complex vector of size n, becomes the result of the overall procedure.

Most published programs to compute FFT pay a lot of attention to rewriting vector values in place. It is possible to compute the function using the space of the input vector, but this requires some careful bookkeeping that obscures the basic algorithm itself. The inherent concurrency of the algorithm is obscured once a particular scan order is adopted. Specification of algorithms such as the FFT using the actions semantics avoids making implementation decisions not inherent in the basic algorithm. If parallel hardware is available (as in some signal processors), the specification can be implemented in parallel. If a sequential implementation is necessary, the actions semantics specification can be transformed in a straightforward manner to use read and write actions, expand the primitive vector functions, and so on. The point is not to perform such optimizations prematurely in the specification of the basic algorithm.

This example did not touch on every kind of action (for example, the conditional action did not occur, but it has many similarities to loop action), but it has illustrated how actions are connected together to define algorithms. It also did not touch on read and write actions and explicit control dependencies. These are more similar to traditional forms, however, and so less in need of explanation.

B.6.4 Implementation Using Memory Writes

Figure B-25 shows a possible implementation of the `bitreverse` function on an array in which the output values are written into the same array object, replacing the input values. This is obviously not a data flow operation. It is one of many possible implementations, although one that would often be used in an implementation of the FFT, because it is simple to do and does not consume extra memory space.

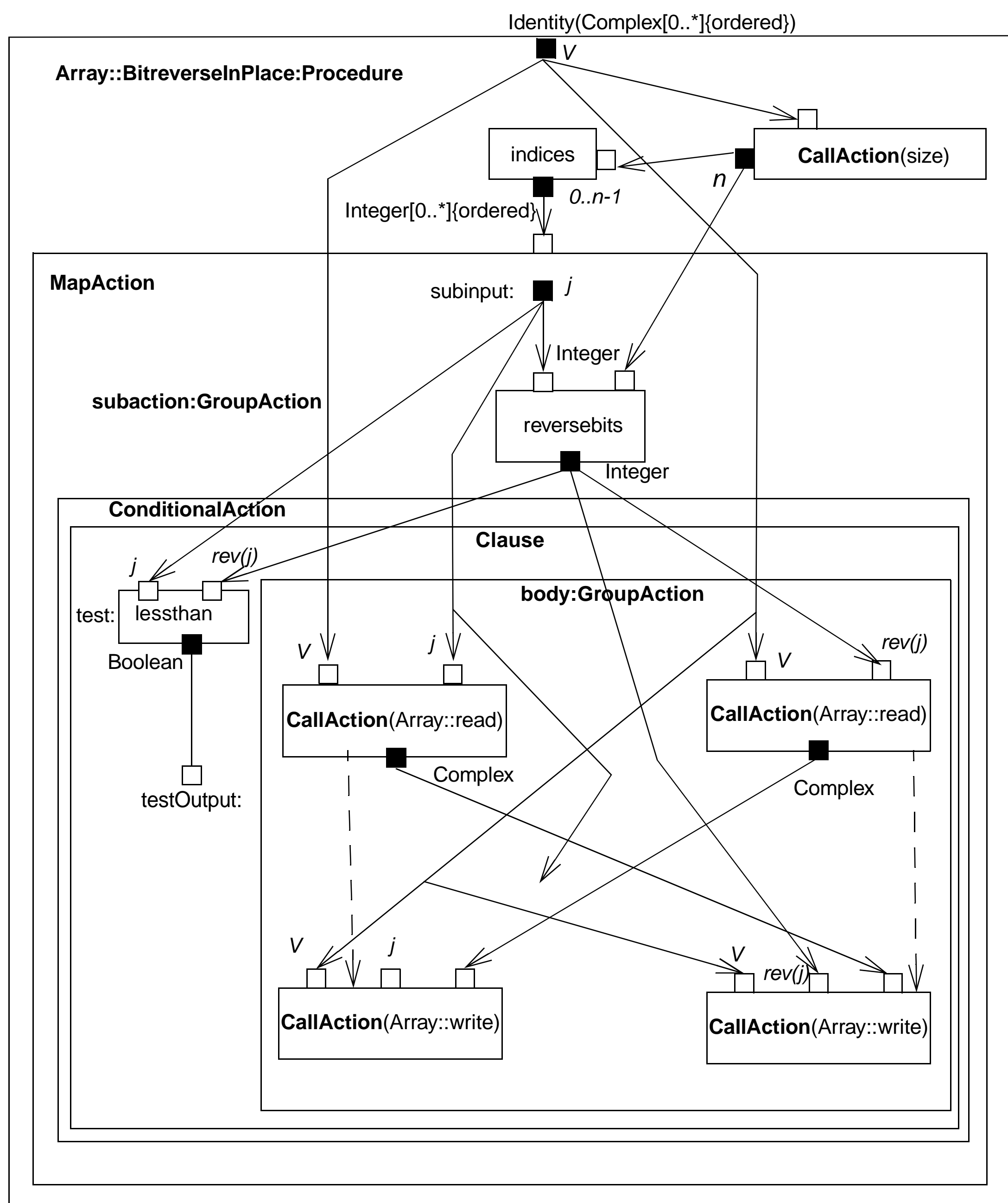


Figure B-25 Implementation of bitreverse using read and write operations

This procedure would be a method on the class `Array`. It would work for an array containing any kind of elements, including complex elements. `Carray` would be a subclass of `Array`.

This procedure has one input pin and no output pin. There are no outputs, because the operation does not generate a result. Rather, it operates on the existing array object whose identity is passed as input. In contrast to Figure B-24 showing the FFT algorithm in a fully data flow manner, the implementation of this procedure operates by side effects, that is, by modifying the state of existing objects.

The input to the procedure is an array *V*, containing elements of arbitrary type. The operation *size* obtains the size of the array, *n*. The primitive function *indices* generates an array containing the indices of array *V*, in this case the integers from 0 to *n-1*. The array of indices is the input to a map action. This means that the subaction within the map action is executed n times, once for each integer from 0 to *n-1*. The value for each execution of the subaction is available on the pin called subaction. Each execution is concurrent with the others.

The subaction comprises the *reversebits* primitive function and an embedded conditional action with a single clause. The *reversebits* primitive computes the function *rev(j,n)*, that is, it reverses the binary bits in an integer to obtain a new integer. One input to the *reversebits* primitive is the value *j*, that is, a value from the array of indices that was the input to the map action. This value is different for each concurrent execution of the subaction. The other input to the reversebits primitive is the value n, which is constant during all executions of the subaction, because it comes from outside the map action.

The conditional action contains a single clause. Its test condition tests whether the map variable *j* is less than its bit-reversed value. If so, the body is executed. If not, the subaction execution is complete—otherwise the same value would be swapped twice. This conditional action does not have clauses to cover every case. If the test fails, there is no other clause that succeeds. If a conditional action produces a data flow output, then at least one clause must be true in all circumstances; otherwise, the data flow output of the conditional would sometimes be undefined. However, in this case the conditional action has no outputs, so it is allowable to not cover all possible situations. In effect, the else clause is a null operation.

The body of the clause merely swaps the values in two cells of the array, those in position *j* and *rev(j,n)*. It does this by reading both values and then writing them back into the opposite positions. An array read operation takes an array *V* and an index *j* and extracts the value at the given position. This is very similar to a direct attribute read action, although a read attribute action has a single input, the identity of the object; the attribute designator is predefined as part of the action, not passed on an input pin. An array write operation takes 3 input values: the array, the index, and the value to write. It has no outputs. Write operations do not have results. They operate by side effects and represent dead ends for data flow values.

The same value of *V* is used 6 times within the overall procedure. At different times, the array may contain different element values, but it is the same array each time with the same identity. It is the identity of an object that is needed for a read or write operation. The identity is unaffected by the operations.

There is a complication. It is permissible to read or write two elements of an array concurrently, but it is not acceptable to write a new value in a cell before the old value has been read. Therefore it is necessary to add a control flow dependency between a

read operation and a write operation at the same position in the array, so that a value is not overwritten prematurely. This is shown in Figur eB-25 by dashed arrows from each read operation to the write operation on the same index value. Algorithms that use read and write operations often need explicit control flow dependencies, as opposed to algorithms in which values pass by data flow only. Note that there is an implicit control flow dependency from the contents of a procedure to the procedure itself—the procedure will not return to the caller until all internal executions have completed.