6
Action Language

All of the processing that goes on in a domain is stated in the actions of state models and the synchronous services for the domain. This processing appears in the OOA models in the form of ADFDs and SDFDs[1], a graphical form of a more general concept: process models. In this chapter we look at properties of process models as defined in this method and present, as an alternative to the graphical rendition currently defined, a text-based action language named SMALL (Shlaer-Mellor Action Language—one L is gratuitous).

6.1 Philosophy of the Graphical Process Models

In developing the concepts and notation for action data flow diagrams—the specification in OOA for all processing—we considered, among other things, the question of what it means to be a process. In particular, how much should we record about a process in an analysis model, and how much can we abstract away?

As an example, how can we account for the work required to get data to or from a data store? Is this a part of a process as shown on a DFD, or is it mere clutter? For a time we held the view that a process had a little shell around it, and the shell—which could be ignored for most purposes—held the work that was required to bring data into and out of the data store. Over time, this perspective evolved so that the little data-mover shell became a separate accessor process.

In due course, we constructed a list of desirable properties of processes:

- A process should be functionally cohesive in the classical sense: It should do one thing and one thing only. In other words, a process should express a single analysis thought.

- A process should be context-free (contain no internal data that survives from one invocation to another). This ensures that the process behaves exactly the same way every time it is executed. The reusability of UNIX filters is due to this property, and we were strongly influenced by this concept.

- A process should be free of side-effects.

The results of this analysis, followed by a number of years of experience, now appear in OOA in the form of rules about process types and what each type can and cannot do. In summary[2], there are five types of processes permitted:

---

[1] For brevity, we refer to ADFDs and/or SDFDs throughout this chapter simply as DFDs. The reader is cautioned not to interpret this shorthand term as a classical (ca.1980) data flow diagram, since the OOA DFDs have entirely different rules and properties. For example, data stores can only be used to represent current time or one of the conceptual entities defined on the Object Information Model. Another difference is that the OOA DFDs are single layered—there is no concept of decomposition, functional or otherwise.

[2] The definition of process types can be found in *Object Lifecycles: Modeling the World in States*. There are a few amendments to the process type definitions contained in *The OOA96 Report*.

Copyright 1997, Project Technology, Inc. All rights reserved

1. Accessors: An accessor can access the data of a single object only. Four types of accessors are defined: read, write, create, and delete accessors.

2. Event generators: An event generator can generate exactly one type of event.

3. Tests: A test process tests relationships between its inputs and produces exactly one control output on each execution.

4. Transformations: A transformation is a process whose purpose is one of computation or transformation of data: the process exists to convert its input data into new forms or results that are then output.

5. Wormholes: A wormhole is a process whose purpose is to transfer control to another domain.

The analyst is then required to provide separate definitions of the processing required within the tests and transformations. The method itself provides definitions of all accessors and event generators, and the processing provided by a wormhole is defined in another domain.

As a result of the seemingly innocuous process type definitions, together with the constraints inherent in the graphical representation, OOA process models, in their graphical form, have a number of powerful properties:

Fundamental Processes. The action being modeled is made up of indivisible atomic processes. We believe that these processes are in some way fundamental to the domain, in that they are frequently reused across a number of different actions.

Separation of Processing from Control. Processing is completely separated from control: The required processing is contained in the processes, while the control (sequencing of computation) is given in the lines that connect processes and data stores.

Reproducibility. Although you can express anything in an action, the way you can express it is actually highly constrained. We have observed repeatedly that two analysts, working independently, produce essentially identical models[3]. Because of this, little time is taken up in building and reviewing the mechanical aspects of the models.

Level of Abstraction. The graphical models are, by and large, at an appropriate level of abstraction for capturing the details of the required processing without introducing any implementation bias. This allows the model builders and reviewers to focus on the particulars of the *problem* and not be drawn into inappropriate distractions having to do with detailed design (how should the data be organized, what control structure to use where, etc.)

Executability. A set of execution rules is defined so that the models can be verified by simulation.

Multiple and Intelligent Translation. Because (a) the analyst specifies only constraints on the order of execution, (b) the models specify only analysis facts about the problem, and (c) each analysis fact is expressed in its entirety as a single unit, the models can be translated to a variety of architectures. In addition, it is possible to build intelligent translators that can optimize for performance, space constraints, or whatever is most appropriate for the system under construction.

---

[3] Leon Starr named this the "ham sandwich test", visualizing the analysts locked in separate offices, and fed by ham sandwiches being slid under the doors.

Copyright 1997, Project Technology, Inc. All rights reserved

Let us set these thoughts aside for the moment and study an alternative approach: specification of actions via textual action languages.

6.2 Background on Action Languages

By 1993, only one commercial tool supplier supported the graphical rendition of the OOA process models. Clients building their own automation tools, as well as other CASE vendors, had chosen instead to implement a textual action language. There were—and are—a number of reasons for this:

- Because of the state of the art of graphical user interfaces, many analysts found entering and maintaining graphical process models time-consuming and problematic.

- Because compiler technology is well understood, it was much easier to build automation to support a process model specified in text.

These action languages have, in general, successfully separated the processing from the sequencing of the processing. However, *to the extent that the approach taken has been based on classical third generation languages* (such as C and C++), these action languages leave much to be desired. The fundamental problems are (1) that they are too low-level and (2) provide too much power and choice. On one hand, they require the analyst to over-specify some aspects of an action (for example, statements in these languages are generally executed sequentially); on the other, constructs are provided for *for* loops, *if* statements, *switch* statements (not allowed in the graphical process models). Finally, there is little or nothing in these languages to enforce rules equivalent to those of the graphical models. As a result, some of the textual action languages require the analyst to violate the underlying rules contained in the graphical rendition, and so forego many of the benefits.

Consider, for example, the following third-generation style action language:

```
Select many dog from instances of object Dog 'owned by' myDogOwnerID;
For each dogID in dog
      generate D1: ComeToDinner( ) to dogID;
       dogId( Weight ) := ProportionalIncrement( dogID( Weight) );
End for;
If myDogOwnerId.FoodLeft < SafeAmount then
     generate DO4: BuyMoreFood to myDogOwnerID;
End if;
```

Traditional control structures of this nature make it difficult to translate the action into other organization—even if the primitives in the language correspond one-to-one with the basic processes in OOA96. Some of the problems here are:

- Because of the sequential nature of the language, control has been over-specified. There is no apparent reason why the *if* clause should be after (or before) the *for* loop. In short, the language fails the ham sandwich test.

- The *for* loop is a general structure into which we could place any number of statements. In the example, two unrelated analysis ideas have been related by being placed in the *for* loop. This obscures the fact that such statements may, in fact, constitute reusable processes. The *if* statement has the same problem—unrelated analysis thoughts can be placed in the body of the statement.

Copyright 1997, Project Technology, Inc. All rights reserved

By examining other code fragments one can find additional problems. Perhaps the best summary conclusion is this: Textual action languages based on third-generation languages tend to produce only a thinly disguised form of the implementation, and do not provide the level of abstraction needed for clear, complete analysis and intelligent code generation.

6.3 The Personality of SMALL

After reviewing a number of existing action languages, we decided that a fundamentally different approach was needed. Our goal in defining SMALL was to produce a textual language that retained the properties of the graphical action models.

The first decision was easy: SMALL should have the nature of a data flow language, using data flow as the primary sequencing mechanism. Hence we based the large-scale structure of SMALL on the most familiar data flow language: pipes and filters.

Some other properties followed from the properties of the OOA DFDs:

- Each chain of processes on a corresponding DFD is a statement in SMALL.
- All statements in the language execute concurrently except when constrained by a data or control dependency.
- Within each statement SMALL thinks of data as active and flowing. Data is assumed to be flowing in sets, and no distinction is made between sets and single data values.
- As with the DFDs, the details of transformations, tests and wormholes are defined separately from the body of the action.
- Execution of the action or synchronous service terminates when the last statement that can execute has completed.
- The execution rules for SMALL are equivalent to those of the DFDs: Execution proceeds in parallel[4] for all statements, except where constrained by data or control flow. Consequently, if two statements write the same variable and their order of execution has not been constrained, it is indeterminate which value will be used in subsequent processing.

To implement control flows that sometime appear on OOA DFDs, we defined guards that can be set and then used to initiate (or not initiate) the execution of a chain of processes.

SMALL can be written in a very terse form—which is used throughout this chapter. An alternative verbose form is easily defined, and we expect to do that in a subsequent publication. Finally, to give you a preview of the language as a whole, let us re-write the dog-feeding example in the terse form of SMALL.

```
// Select all dogs owned by the owner, flow them to a base process
// that generates an event to each
    myDogOwner -> [R1.'owned by'] Dog( all ) | Gen D1: ComeToDinner;

// Select all dogs owned by the owner, get the weight of each, flow that
// to a base process that computes the new weight, and write it back
    myDogOwner -> [R1.'owned by'] Dog( all ).Weight |
        ProportionalIncrement >Dog( ).Weight;

// Get the FoodLeft attribute of the owner, and the variable SafeAmount,
// flow the pair to a test process, TestLess, which has two guard conditions
    (myDogOwner.FoodLeft, ~SafeAmount ) | TestLess? !Less, !GreaterEqual;

// If the Less guard is set, flow the DogOwner to the event generator
    !Less: myDogOwner | Gen DO4: BuyMoreFood;
```

---

[4] The statements proceed "in parallel" from the analyst's perspective. The architecture can serialize the statements as desired for optimization or other purposes.

Copyright 1997, Project Technology, Inc. All rights reserved

```
// If the other guard is set, do nothing
    !GreaterEqual: ;
```

We describe the structure and detail of the language in subsequent sections.

6.4 Overall Syntax

An action language *segment* consists of a number of statements. Each *statement* can be either a simple statement, involving access to the attributes of an object, relationship traversals, and invocations of processes, or it may be a block. A *block* comprises a set of statements grouped together with square brackets [and].

An action language statement always terminates with a semicolon.

Comments are introduced with // at any point in the line. When this pair of characters is encountered, the remainder of the line is ignored.

Action language statements include names for:

- variables: data variables and reference variables
- OOA model elements: attributes, objects, relationships, events, process invocations, and the like
- language-supplied processes
- keywords

Variable names, names of OOA model elements, language-supplied processes and keywords are all case insensitive, so "Bench" is treated the same as "bENCH".

Variables names may contain only the characters, a-z, A-Z, 0-9 and _, and must not begin with a numeric character, 0-9.

White space (spaces and tabs) may be inserted at any point in a SMALL statement other than within a name, or between characters that make up a token (such as // or ->).

Names of OOA model elements can be written in a “whitespace-removed” form, as in InServiceIonChamber. Equivalently, these names can contain spaces, provided that they are delimited by tick marks. Hence the following names are all the same:

```
'In Service Ion Chamber'
InServiceIonChamber
inserviceionchamber
```

as are

```
'Inser Vice Ionch Amber'
InServiceIonChamber
inserviceionchamber
```

but **`'In Service Ion Chamber'`** and **`'Inser Vice Ionch Amber'`** are not.

6.5 Types, Literals, Data Variables and Composition

Types

Attributes, supplemental data items of an event, and synchronous service input and output parameters have domain-specific data types, as described in Chapter 3. These types are defined as part of the definition of the attribute, event or synchronous service.

Literals

A literal is a constant value that takes the form of

- a number: 3, 4.25, 3E-6,
- a string: "Therapy", "red", "11:30:00"
- a boolean: true, false

A literal does not itself have a type. However, when a literal is written to a model element—that is, written to an attribute, passed as a supplemental data item in an event, or passed as input to a process or wormole—the value of the literal is coerced to the domain-specific data type of the model element.

The coercion rules in SMALL are quite general.

- A number can be coerced to any domain-specific data type based on the numeric base type. It may also be coerced to a duration—in which case it is interpreted as a number of seconds.

- A string can be coerced to base type symbolic. It may also be coerced to a value of an enumerated data type: The string "Therapy" can become the enumerated value therapy.

- A string that has the pattern of a time or duration can be used to set the value of a data element so typed. Hence "97-06-18 9:35:00" can be used to establish the value of an attribute such as Gas Bottle.Date of Delivery.

Data Variables

A *data variable* is a locally defined variable that contains data values. A data variable is in scope throughout the action or synchronous service in which it appears.

The name of a data variable always begins with the special character ~. Reading aloud, the data variable **`~Temperature`** is “data Temperature.”

Like a literal, a data variable does not have a data type. However, when data is written to a data variable, the data variable takes on the type of the data written to it. As with a literal, when a data variable is written to an attribute, passed as a supplemental data item in an outgoing event, or passed as input to a process or wormhole, the value of the literal will be coerced to the domain-specific data type of the data variable.

Data variables may be created by writing to a name using the write operator >, read “write.” This operator takes the value on the left hand side and writes it to a (newly created) data variable on the right hand side, so the statements

```
"Therapy" > ~BenchUse;          21 >~CaveTemp;
```

cause the creation of new data variables **`~BenchUse`** with the value **`"Therapy"`**, and **`~CaveTemp`** with value 21.

When an action (or synchronous service) starts to execute, the supplemental data items supplied with the event (or the input parameters) become available as data variables. The name of the data variable is the same as the whitespace-removed form of the supplemental data item or input parameter. Hence, the event ISR2: Current Integrated(Color, Suffix, Ring Number; Integrated Current)[5] creates and initializes one data variable, **`~IntegratedCurrent`**.

Composition

When several data variables are required together, group them with the composition operator ','. The expression

```
( ~Temp, ~Pressure )
```

is read "data Temp with data Pressure," and it groups **`~Temp`** and **`~Pressure`** together in that order.

The statement

```
( ~Temp, ~Pressure ) > ~CaveProperties;
```

writes to the data variable **`~CaveProperties`**, which may now be used wherever **`~Temp`** and **`~Pressure`** flow together. This feature follows the classical concept of defining a flow as the composition of other elements.

6.6 References, Reference Variables and Attribute Access

References

A *reference* refers to an instance of an OOA object. Exactly what form a reference takes depends upon the architecture. The analyst can think of a reference as an abstraction of any reasonable reference scheme: a handle, a set of values for identifying attributes, a table name and a row number, etc.

The reference **`Self`** may be used—in an action only—to refer to the current instance.

Reference Variables

A *reference variable* is a locally defined variable that contains one or more references[6]. A reference variable is in scope throughout the action or synchronous service. The name of a reference variable—for example, **`myBench`**—does not start with ~, so one may easily distinguish a data variable from a reference variable. Read **`myBench`** as "reference myBench."

Attribute Access

---

[5] Color, Suffix and Ring Number form the identifier of an instance of an In-Service Ring, and are therefore not supplemental data items

[6] If a reference variable contains multiple references, all such references must be to instances of the same object.

Copyright 1997, Project Technology, Inc. All rights reserved

Access a non-referential attribute of an instance with the operator '.', read “dot.” Place the variable containing the reference to the instance on the left of the operator, and the attribute to be accessed on the right. Hence, to read the pressure in the cave referred to by the reference variable **myCave**, write

```
myCave.Pressure
```

When writing an attribute with the write operator >, use the same construction, as in

```
990 > myCave.Pressure;
```

Multiple attributes may be read or written at once by making a list of attributes inside parentheses:

```
( ~Temperature, ~Pressure ) >myCave.( Temperature, Pressure );
```

or

```
~CaveProperties > myCave.( Temperature, Pressure );
```

Note that **~Pressure** is a data variable, while **Pressure** is an attribute name. Note also that the composition **~CaveProperties** must comprise two values that have domain-specific data types matching, in order, those for the attributes Temperature and Pressure.

There are two kinds of attributes whose value is architecture-dependent:

- referential attributes
- identifying attributes of type arbitrary

The values of these attributes are therefore meaningless in the OOA models, and therefore may not be accessed directly as described here.

6.7 Selecting Instances

Selecting an Arbitrary Instance

To find an arbitrary instance of an OOA object, write the name of the object followed by the keyword **one** in parentheses. Hence, **Bench(one)** picks an arbitrary bench. The expression is read “one Bench.”

This expression produces a reference, which may then be written to a reference variable, as in

```
Bench( one ) > myBench;
```

The reference may be used to access attributes: **Bench(one).Name >~BenchName;**

Selecting an Instance with Specified Properties

To find an instance that has a specified property, express the required property as a comparison. The *comparison* names one attribute, a data value, and a comparison operator (>, <, =, ≠, ≤, ≥ —as defined in Chapter 3). The effect of the comparison is to examine each instance of an object, comparing the value of the specified attribute with the specified data value; this yields those instances that satisfy the criterion. For example,

```
Bench( one, BenchUse = "Therapy" )
```

Copyright 1997, Project Technology, Inc. All rights reserved

yields one arbitrarily selected instance of Bench whose use is equal to "Therapy". The comparison is read "where." Hence the example above is read "one Bench where Bench Use equals Therapy."

Arithmetic operators may be used in forming comparisons, as in [arithmetic precedence list in sidehead]

```
Cave( one, Temperature < (  ~TempInFahrenheit - 32 ) * 5 / 9 )
```

This expression yields an instance of Cave that has a temperature less than the specified temperature in degrees Fahrenheit.[7]

To select an instance based on the values of multiple attributes, you may use a *compound comparison* made up of comparisons and logical operators. Hence to select a cave with Temperature greater than 21° C and Pressure less than 990 millibars, write

```
Cave( one, ( Temperature > 21 ) ∧ ( Pressure < 990 ) )
```

The order of precedence for operators from highest to lowest is

arithmetic operators:

-	unary minus
**	exponentiation
*, /, %	multiply, divide and integer divide
+, -	addition and subtraction

logical operators:

¬	negation
∧, ∨	logical and, and logical or

comparative operators:

>, <, =, ≠, ≤, ≥

Having selected an instance, you may access its attributes[8] just as you would using a reference variable. Hence

```
// to read the Pressure of one cave hotter than 21°C
 Cave( one, Temperature > 21 ).Pressure
```

is equivalent to

```
Cave( one, Temperature > 21 ) > myCave;
myCave.Pressure
```

and

```
// to write Pressure of 990 millibars in the North Cave
    990 > Cave( one, Name = "North" ).Pressure
```

is equivalent to

```
Cave( one, Name = "North" ) > myCave;
```

---

[7] The domain-specific data type of attribute Cave.Temperature has units of degrees Centigrade. 0°C = 32°F and 100° C = 212°F, hence°C = (°F - 32) * 5 / 9.

[8] You can access only those attributes that are meaningful to the analyst: that is, non-referential attributes that are not of type arbitrary.

```
990 > myCave.Pressure;
```

Selecting Multiple Instances

To select multiple instances of an object, use the keyword **all**, instead of **one**. Examples are:

```
Cave( all )                              // finds all caves, read all Caves
Cave( all, Temperature > 21 )       // finds all Caves where temperature
                                    // is greater than 21°
Cave( all, Temperature > ~Temp)     // finds all caves hotter than ~Temp
Cave( all, ( Pressure < 1000 ) ∧ ( Temperature > 21 ) )
                                    // finds all Caves with pressures less
                                    // than 1000 millibars, and with
                                    // temperatures greater than 21°
```

Once some instances have been selected, you can access their attributes in the usual fashion:

```
Cave( all, Temperature > 21 ).Pressure         // reads pressures of all caves
                                              // hotter than 21°
Cave( all, Temperature > ~Temp).Pressure          // reads pressures of all caves
                                              // hotter than ~Temp
Cave( all ).( Temperature, Pressure)              // reads the temperature and
                                              // pressure of all caves
```

The first two expressions produce a set of pressures. The last expression yields a set of tuples.

The next statement writes the value 990 as the pressure of all caves that qualify.

```
990 > Cave( all, Temperature > 21 ).Pressure;
```

The following statement writes the same temperature and pressure for all caves.

```
( ~NominalTemperature, ~NominalPressure ) >Cave( all ).(Temperature, Pressure );
```

And, assuming **~Nominals** contains two values, one that matches the type for Temperature, and another that matches the type for Pressure, the following statement means the same as above:

```
~Nominals >Cave( all ).( Temperature, Pressure );
```

Selecting in Order

To select instances in a particular order, specify the order with the ascend operator / or the descend operator \ followed by the the name of the attribute on which to order. The operators are read “ascend” and “descend” respectively. The order specification can appear anywhere inside the parentheses, as in

```
Cave( Temperature > 21, /Name ).Pressure  >~PressuresOfCavesOrderedByName;
```

The ordering scheme is defined by the domain-specific data type of the attribute, hence the preceeding statement gets the pressures of all qualifying caves in ascending alphabetical order by name[9], and writes them to a data variable. Multiple orderings can be specified, so

```
Cave( all, \Pressure /Name).Pressure >~PressuresOfCavesOrderedTwice;
```

---

[9] Cave.Name has a base type of symbolic.

will read the pressure of all caves, ordered first by descending pressure, and then by ascending name for caves with the same pressure.

Ordering on a attribute whose domain-specific data type is based on a type that has no inherent ordering (booleans, for example) is meaningless, hence the instances are produced in an undefined order.

Object Expressions and Selectors

An *object expression* is any expression that yields references. Hence a reference variable, such as **myCave**, is an object expression, as is **Cave(all)** and **Cave(all, /Name)**.

A *selector* is a special form of an object expression. A selector comprises an optional quantifier, an optional comparison (possibly compound) and optional order specifications. The quantifier may be **one** or **all**.

Both the comparison and the quantifier are optional, but at least one must be specified. When no comparison is specified, the object expression yields one or all instances of the specified object, depending on the quantifier. When no quantifier is specified, the object expression yields *all* instances that qualify. Hence,

```
Cave( one )                         // yields one cave
Cave( one, Temperature > 21 )       // yields one cave hotter than 21°
Cave( all )                      // yields all Caves
Cave( all, Temperature > 21 )       // yields all caves hotter than 21°
Cave( Temperature > 21 )         // yields all caves hotter than 21°
```

In summary, then, to select an instance or instances, specify the object name followed by a selector in parentheses.

6.8 Instance Creation and Deletion

Creating Instances

To create an instance of an object, use the create operator >> with a literal, data variable or composition on the left, and the name of the object on the right. List the attributes separated by a period and enclosed with parentheses, after the object name.

```
// This statement creates an instance of cave with a name North, temperature
// of 21°, and a pressure of 990 millibars
    ( "North", 21, 990 ) >> Cave.( Name, Temperature, Pressure );
```

Read the statement as “North, 21 and 990 creates Cave with Name, Temperature and Pressure.”

A create operation initializes attributes: The first attribute is initialized by the first element in the data flow, the second attribute by the second element, and so forth.

An alternate form for the create operation looks like this

```
>> Cave.( "North" => Name, 21 => Temperature, 990 => Pressure );
```

where a value is placed to the left of the *explicit assignment operator* =>, and an attribute name is placed on the right. Read the explicit assignment operator as “assigned to.”

Copyright 1997, Project Technology, Inc. All rights reserved

The two forms can be intermixed, hence:

```
>> Cave.( "North" => Name, 21 => Temperature, 990 => Pressure );
```

and

```
21 >> Cave.( "North" => Name, Temperature, 990 => Pressure );
```

both mean the same.

To create an instance in a state, use the reserved word State, which is treated syntactically as though it were an attribute.

```
>> BeamPulse.( ~TimeOfTb => TimeOfTb, "Creating Following Pulse" => State );
```

Consistent with OOA96, the state "attribute" cannot be read.

The create operator produces a reference as a result. This can then be stored for future use, as in

```
( "North", 21, 990 ) >> Cave.( Name, Temperature, Pressure ) >myCave;
```

Rules about Initialization of Attributes

Various rules of OOA are reflected in SMALL in the following rules regarding the initialization of attributes in a create operation.

- It is illegal to initialize a referential attribute. Instances of relationships are established by another mechanism, as explained in section 6.13.

- The initial value of a non-referential, non-identifying attribute is established in one of the following ways:

  The analyst may initialize the value explicitly in the create operation.

  If the analyst does not initialize the value explicitly, and if a default value was defined for the domain-specific data type of the attribute, the attribute will take on that default value. If no default value was specified, the attribute will be initialized to UNDEFINED.

- If an identifying attribute is non-referential and is of type arbitrary, the initial value will be established by the architecture. Consequently, it is illegal to attempt to initialize such an attribute in the create operation.

- If an identifying, non-referential attribute is not of type arbitrary, the analyst is required to establish the initial value in the create operation.

These rules leave us with one more case to consider: The initialization of an identifying attribute that is not of type arbitrrary and is also referential. This case is discussed in section 6.13.

Deletion of Instances

Use the delete operator <<, read "delete," to delete instances, as in

```
<< myBench;
```

Copyright 1997, Project Technology, Inc. All rights reserved

which deletes those instances referred to by **myBench**. Similarly,

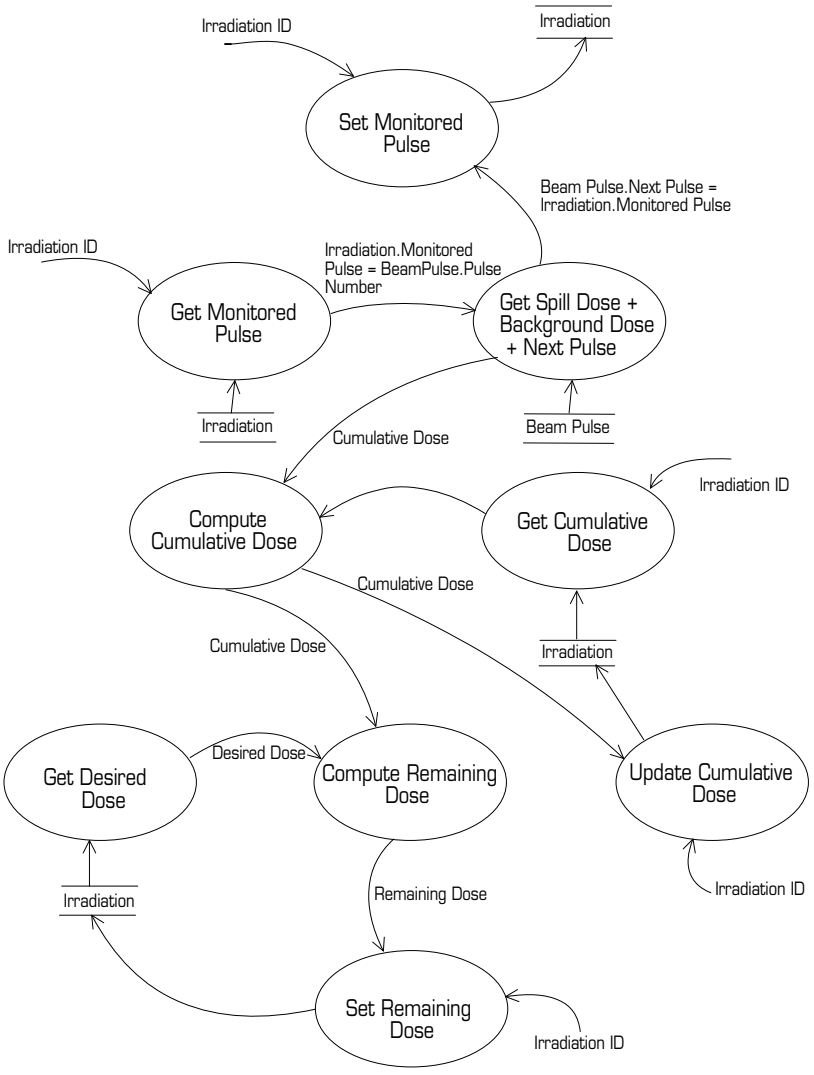
```
<< Bench( BenchUse = "Therapy" );
```

deletes all the Bench instances whose use is "Therapy". Read the latter example as “delete Bench where Bench Use equals Therapy.”

6.9 Flowing Data to Transformations

Transformations

To flow data to a transformation, use the flow operator |, read “flows to.” For example:




```
(   aBeamPulse.( SpillDose, BackgroundDose ),
    self.CumulativeDose
) | ComputeCumulativeDose >~CumulativeDose;
```

This example takes the **SpillDose**, the **BackgroundDose** and the **CumulativeDose** and flows them, in order, to the **ComputeCumulativeDose** transformation. The result is then written to a data variable.

The transformation **ComputeCumulativeDose** is defined, together with the types and order of its input and output parameters in a process specification. When a transformation is defined to have multiple input or output parameters, the parameters are represented in SMALL as a composition: a single data flow that contains several data elements.

Re-ordering Data Elements

The language-supplied process **shuffle** re-orders data elements in a flow. Provide the desired order of the data elements in parentheses, numbered from 1. For example:

```
(   self.CumulativeDose,
    aBeamPulse.( SpillDose, BackgroundDose )
) | shuffle( 2, 3, 1 ) | ComputeCumulativeDose >~CumulativeDose;
```

provides the data values to the transformation in the order **SpillDose**, **BackgroundDose** and **CumulativeDose**.

Dropping Data Elements

Use **shuffle** to drop data elements from a flow. Hence if we write

```
(   self.CumulativeDose,
    aBeamPulse.( SpillDose, BackgroundDose )
) | shuffle( 3, 1 ) | ComputeCumulativeDose >~CumulativeDose;
```

the second data element (**SpillDose**) is dropped from the input flow and the inputs provided to **ComputeCumulativeDose** are **BackgroundDose** and **CumulativeDose**, in that order.

Merging Data Elements

To merge data elements into a flow, provide the names of the data elements with the **shuffle** transformation. Hence:

```
    self.CumulativeDose >~CumulativeDose;
    (   aBeamPulse.( SpillDose, BackgroundDose )
    )   |   shuffle   (   1,   2,   ~CumulativeDose   )   |   ComputeCumulativeDose
>self.CumulativeDose;
```

appends the data variable **~CumulativeDose** after the other two data elements. Note that only named data variables may be added in this manner. References and literals cannot be merged directly.

6.10 Generating Events

Events

To generate an event, use the keyword **gen** followed by the event label and meaning. Flow the reference of the destination of the event to an event generator using the flow operator |. For example:

```
aBench | Gen BE6: GasNotOK;
```

Supplemental Data Items

An event specification provides names for the supplemental data items that are to be passed with an event. To provide supplemental data items with an event in SMALL, use the explicit assignment operator => to assign a value to the name of the supplemental data item. Place the explicit assignments in parentheses after the event label:

```
aMonitoredPulse | Gen BP6: SpillDoseUpdated( ~ComputedDose => SpillDose );
```

As a shorthand, you may omit the => <supplemental data item> if the data variable and the supplemental data item have the same name. Hence the following two statements are equivalent.

```
aMonitoredPulse | Gen BP6: SpillDoseUpdated( ~SpillDose );
aMonitoredPulse | Gen BP6: SpillDoseUpdated( ~SpillDose => SpillDose );
```

It is *not legal* to flow a data variable to an event generator:

```
aMonitoredPulse, ~SpillDose | Gen BP6: SpillDoseUpdated;  // NOT LEGAL
```

Delayed Events

To send a delayed event, add a duration in parentheses after the keyword **gen**.

```
myDog | Gen( ~StayTime ) D7: FinishStay;
myDog | Gen( "1:00:00" ) D7: FinishStay;   // 1 hour—after coercing the string to
a duration
```

If a numeric literal is used to specify the delay time, the units of the literal are presumed to be seconds.

```
myDog | Gen( 3E-6 ) D7: FinishStay;     // 3*10^-6 seconds, 3 microseconds—a number
```

Provide the supplemental data items, if any, just as you do for a normal undelayed event:

```
myDog | Gen( 3E-6 ) D7: FinishStay( "Sarzak" => Name );  // About the best he can
do
```

To cancel the most recent delayed event of a particular meaning, use the **cancel** keyword.

```
myDog | Cancel D7: FinishStay;
```

This example cancels the most recent D7: FinishStay to the instance(s) of Dog given by **myDog**.

To find the time remaining for an event, use the **timetill** keyword, which produces a single data output.

```
myDog | TimeTill D7: FinishStay > ~TimeRemaining;
```

6.11 Wormholes

Request Wormholes

Recall from Chapter 5 that when a request wormhole is invoked, all of the following must be supplied:

- Data items to be transmitted to the receiving domain. This includes identifiers (references) to instances used to make up transfer vectors.
- Data items output from the wormhole when control returns to the DFD from which the wormhole is invoked.

To invoke a wormhole, provide any reference(s) to an instance of an OOA object in the input data flow, and the non-reference data items, if any, to be transmitted to the recipient in parentheses after the name of the wormhole, just as you do for the supplemental data items of an event.

```
myRobot | Wormhole W3: RetractHand( 3 => distance );
```

where **`wormhole`** is a keyword in SMALL, **`W3`** is the identifier of the wormole, and **`RetractHand`** is the name of the wormhole.

The synchronous output data items, if any, appear on the data flow following the invocation, as in

```
myMagnet | Wormhole W4: GetCurrent > ~Current;
```

If more than one data item is input to (or output from) a wormhole, the data items that make up the input (output) data flow must be treated as a composition, as in

```
myRobot | Wormhole W16: WheresTheRobot > ( ~x, ~y, ~theta );
```

It is *not* legal to flow a data variable to a wormhole

```
~DesiredCurrent, myMagnet | Wormhole W5: SetCurrent;         //NOT LEGAL
```

nor is it legal to provide inputs that are references via explicit assignment:

```
Wormhole W5: SetCurrent ( ~DesiredCurrent => Current,
    myMagnet => whichMagnet )                              //NOT LEGAL
```

Returns

To return control to the client domain, flow the client return (transfer vector or return coordinate) to the language-defined process **`Return`**.

```
Self.ClientReturn | Return;
```

or

```
~ReturnCoordinate | Return;
```

Supply return data values as a part of the same flow, not as parameters:

```
( Self.ClientReturn, ~Current ) | Return;          //  To  return  the  value  of
Current
```

6.12 Traversing Relationships

Level of Abstraction

When tracing a chain of relationships, one is currently[10] required to provide a separate accessor for each link in the chain. While this is entirely consistent with the relational model of data, it is not at the level of abstraction preferred by most analysts, who generally like to cast a lengthy traversal as a single thought.

More to the point, it is not at a level of abstraction suitable for translation. Translation requires that each relationship traversal be treated as a single unit of computation because it must be translated as a single unit. SMALL, on the other hand, provides the desired level of abstraction—and in a very compact fashion.

Traversing a Single Relationship

To traverse a single relationship, begin with a reference and use the relationship traversal operator ->, read “cross” or “to,” followed by a relationship specification in square brackets and an object expression:

```
myBench -> [R2.IsEquippedWith] GasLine( one );
```

This example uses the reference **myBench** to traverse the relationship R2 to the (one) related instance of the Gas Line object.

The entire relationship traversal is an object expression—an expression that yields references. Hence, to read the flow alarm indication of the related gas line, you may write:

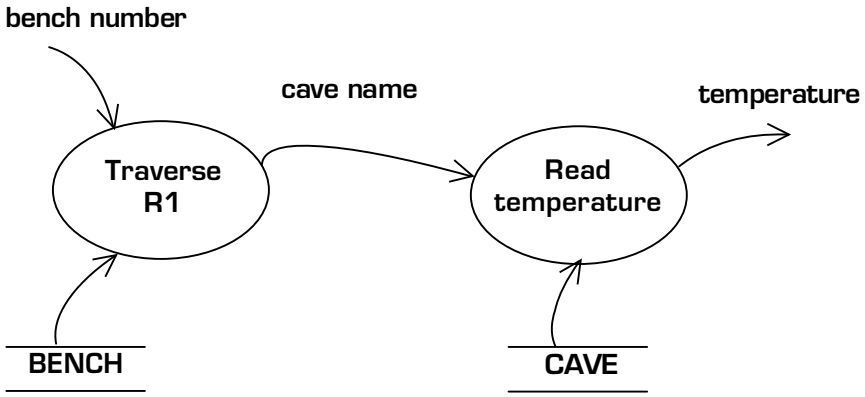
```
myBench -> [R2.IsEquippedWith] GasLine( one ).FlowAlarmIndication
```

Given a specified cave, to select one bench (1) that is housed therein and (2) whose use is "Therapy", and to return its bench number, write:

```
myCave -> [R1.Houses] Bench( one, BenchUse = "Therapy" ).BenchNumber
```

To write the cave temperature in which the bench is housed:

```
21 > ( myBench ->[R1.IsHousedBy] Cave( one ).Temperature );
```



And, to generate an event to the gas line with which the bench is equipped, write

```
myBench -> [R2.IsEquippedWith] GasLine( one ) | Gen GL1: GasFlowing;
```

---

[10] As defined in *Object Lifecycles* as well as the *OOA96 Report*.

Copyright 1997, Project Technology, Inc. All rights reserved

Relationship Specification

A relationship specification is a relationship number or verb phrase, or both, enclosed in square brackets. If a relationship phrase is supplied, precede it with a period. The following expressions all mean the same:

```
myBench -> [R2.IsEquippedWith] GasLine( one ).FlowAlarmIndication
myBench -> [R2.'Is Equipped With'] GasLine( one ).FlowAlarmIndication
myBench -> [R2] GasLine( one ).FlowAlarmIndication
myBench -> [.IsEquippedWith] GasLine( one ).FlowAlarmIndication
myBench -> [.'Is Equipped With'] GasLine( one ).FlowAlarmIndication
```

Traversing Multiple Relationships

To traverse multiple relationships, string together the relationships and intervening objects:

```
myBench -> [R2] GasLine( one ) ->
    [R4] GasBottle( one ) ->
        [R3] GasSpecification( one ).StoppingPower
```

This statement reads the stopping power of the gas that is in the bottle that supplies gas to the gas line that is installed on the bench.

Alternatively, you can string together the relationship specifications inside the square brackets, naming only the object at the end of the traversal, as in:

```
myBench -> [R2 -> R4 -> R3] GasSpecification( one ).StoppingPower
```

Reaching Multiple Instances

A relationship traversal can begin with any reference, either a reference variable or an object expression:

```
//starting with a reference variable
myCave -> [R1] Bench( all ).BenchUse
//starting with an object expression
Cave( all, Temperature > 21 ) -> [R1] Bench( all ).BenchUse
```

The second expression finds all caves with temperature hotter than 21°, then traverses from each such cave to find all related benches and then gets their uses.

The expression

```
Cave( all, Temperature > 21 ) -> [R1] Bench( one ).BenchUse
```

yields the use of *one* bench per cave whose temperature is greater than 21°, as shown in the shaded rows of the tables.

Cave		
Name	Temp	Pressure
North	22	990
Middle	21	985
South	24	980

Bench		
Number	Bench Use	Cave
1	Therapy	South
2	Dosimetry	Middle
3	Dosimetry	North
4	Biology	North

Copyright 1997, Project Technology, Inc. All rights reserved

The first part of the expression finds all caves hotter than 21° (North and South). The relationship traversal then picks one bench for each selected cave—Bench 4 for the North Cave (or it could be Bench 3), and Bench 1 for the South Cave—and reads the uses of those two benches.

To read attributes of just one bench, use **one** as the quantifier in *all* selectors.

Unique

When starting from a set of instances, it is possible to return data about the same instance more than once. Consider, for example

```
Bench( all ) -> [R1] Cave( one ).Temperature;
```

which takes the set of benches, finds the cave housing each bench, and then gets its temperature. The number of caves that you get is equal to the number of benches, so you may get the same temperature many times. Formally speaking, the traversal returns a bag[11] of cave temperatures. To reduce the bag to a set (i.e. to find only the unique values), flow the result to the language-supplied process **unique**.

```
Bench( all ) -> [R1] Cave( one ).Temperature | unique >~UniqueTemperatures;
```

Conditional Relationships

If a relationship is conditional, a traversal may fail to find an instance. This topic is discussed below in Section 6.17: *Failure to Find Instances*.

6.13 Relationship Creation, Deletion and Migration

Creating an Instance of a Relationship

To create an instance of a relationship, use **link** and a relationship specification together with references to the instances to be related:

```
( myBench, myCave ) | link [R1.Houses];
```

As usual, either of the two parts of the relationship specification is optional.

To create an instance of an associative object, combine the **link** and the **create** operators:

```
( myBench, myScintillator ) | link [R25] >> InServiceScintillator;
```

This statement creates an instance of the InServiceScintillator object with its referential attributes initialized, together with any attributes that have default initial values.

To initialize non-referential attributes of an associative object, use an explicit assignment:

```
( myBench, myScintillator ) | link [R25] >>
    InServiceScintillator.( ~Position => MeasuredBenchPosition );
```

---

[11] Recall that a bag is a collection which may contain duplicates, whereas a set contains only unique elements.

Use the same form for initializing identifying attributes that are not of type arbitrary in an associative object used to formalize a M-(M:M) relationship. For example:

```
(   PublicDomainDrug( Name = "Aspirin" ),
    CertifiedManufacturer( Name = "MyDrugCo" )
) |     link [R3] >> Batch.( BatchNo => "123" );
```

Deleting an Instance of a Relationship

To delete an instance of a relationship, use the **`unlink`** operation:

```
( myBench, myCave ) | unlink [R1.'Is Housed In'] ;
```

To delete a relationship formalized by an associative object, combine **`unlink`** and the delete operator:

```
( myBench, myScintillator ) | unlink [R1] << InServiceScintillator;
```

For a M-(M:M) relationship, you must clarify which instance is to be deleted:

```
(   PublicDomainDrug( Name = "Aspirin" ),
    CertifiedManufacturer( Name = "MyDrugCo" )
)   |  unlink [R3] << Batch( BatchNo = "123" );
```

Instance Migration

To migrate an instance of one subtype to another subtype of the same supertype, use the migrate operator ->>. Place an object expression referring to the "migrating from" subtype on the left of the migrate operator and the name of the "migrating to" subtype on the right. The expression yields a reference to the "migrated to" instance.

```
myOnlineDisk ->> OfflineDisk >myOfflineDisk;
```

This expression effectively deletes the instance of the old subtype and creates an instance of the new subtype[12]. Like a standard create operation, the **`migrate`** operator returns a reference.

Attributes—including the State "attribute"—may be initialized in the same way as in done with the create operator:

```
myOnlineDisk ->> OfflineDisk.( "In Library" => State ) >myOfflineDisk;
```

6.14 Guards, Blocks and Sequencing

Guards

All sequencing of processing in SMALL is accomplished with data flows and control flows. SMALL implements the concept of a control flow using the concept of a guard.

A *guard* is a gate that allows processing to take place only if the guard has been set, as in:

---

[12] Whether the instance of the old subtype is actually deleted, or simply modified to become an instance of the new supertype is dependent on the architecture's method for implementing sub/supertypes.

```
!OK: anIrradiation | Gen IR2:  BenchOK;
```

The guard—**!OK** in this example—must appear at the beginning of a statement. The remainder of the statement is executed only if the guard has been set. Otherwise the statement never executes. Read the statement "if guard OK is set . . ."

Note that all guards are automatically reset before the action or synchronous service begins to execute.

Setting a Guard

To set a guard, use the sequence operator !, read "sequence," followed by the name of the guard. This construct may be used only at the end of a statement, as in:

```
self.IrradiationID > ~IrradiationID !DataInHand;      // Read the ID
!DataInHand: << self;                                 // Now delete the instance
```

These two statements always execute in the same order, regardless of the order in which the statements appear in the action.

A guard may also be set by a test process, as discussed in Section 6.15.

Combining Guards

Multiple guards may be used to establish compound constraints on the execution of a statement. To specify that a statement can execute if either **!A** or **!B** (or both) are set, write

```
!A, B:  statement;
```

as in

```
!LessThan, Equal:   . . .    // executes if either LessThan or Equal is set
!GreaterThan:  . . .         // executes only if GreaterThan is set
```

To specify that a statement can execute only if **!A** and **!B** are *both* set, write

```
!A:  // line break not required, but recommended
!B:  statement;
```

as in

```
// Complete both unlinks before self-immolation
( self -> [R24] ScintillatorPowerSupply, self ) | unlink [R24]  !Done1;
( self -> [R25] Scintillator, self-> [R25] Bench )  | unlink [R25]  !Done2;
!Done1:
!Done2: << self;
```

Blocks

A set of statements may be guarded by surrounding them with square brackets [and] to make a block, as in:

```
!InProgress : [
    >> BeamPulse.( ~TimeOfTb, "Create Following Pulse" => State ) >newBeamPulse;
    ( self -> [R31] Irradiation, newBeamPulse ) | link [R31];
]
```

Copyright 1997, Project Technology, Inc. All rights reserved

6.15 Test Processes

A test process, in SMALL, is invoked by writing the name of the test process followed by a question mark that is, in turn, followed by a list of guards that may be set by the test process, as in

```
self -> [R2] GasLine( one ).FlowAlarmIndication | TestOK? !OK, !NotOK;
!OK: ...
!NotOK: ...
```

The construct is read as "TestOK sets OK or NotOK."

A number of rules apply to test processes and their use:

- There must be at least one guard in the list.
- A test process must be defined so that it sets exactly one of the guards when it is executed.
- The invocation of a test process must appear at the end of a SMALL statement. This rule is simply the result of the fact that new chains of processes must necessarily follow a test process.

Each guard that can be set by the test process must appear at the beginning of some SMALL statement in the same action or synchronous service in which the test process is invoked. This is required even if there is no work to be done associated with a guard.

```
myRobot.Operational | RobotStuck? !Stuck;
!Stuck:  ;     // haven't written the recovery operation logic yet,
               // but MUST have the guard
```

6.16 Establishing and Using Sets of References

Repeated Accesses

Whenever a statement reads attributes, it establishes a set of references to the instances whose attributes were read. For the rest of the action or synchronous service (or until a different set of references to the same object is established), the name of the object followed by an empty set of parentheses refers to that set of references. Hence:

```
Crate( all ).( Height, Width, Depth ) | ComputeVolume >Crate( ).Volume;
```

reads the dimensions of all crates, runs these values through a base process to compute the volume, and writes the result as an attribute of each crate that was included in the set of references. The statement is read: "all Crates dot Height, Width, Depth flows to ComputeVolume writes those Crates dot Volume."

Why is this important? Recall from *Object Lifecycles* that, under the simultaneous interpretation of time, multiple actions can be in execution at the same time. With an architecture that uses the simultaneous interpretation, it is possible that another action could add or delete a crate between the time we read the dimensions of the crates and the time we write back the volumes. This could have could have unfortunate consequences should we write

```
Crate( all ).( Height, Width, Depth ) | ComputeVolume >Crate( all ).Volume;
```

thereby causing the second access to be made independently of the first. Such consequences can be eliminated by reusing the established set of references for all accesses after the first.

Splitting a Data Flow

Consider the graphical model shown in Figure 6.16.1. The essence of the problem is to split a set of crates into two subsets—big crates and small crates—and then to treat all members of one set in one manner and all members of the other in a different manner. This is easily accomplished in SMALL:

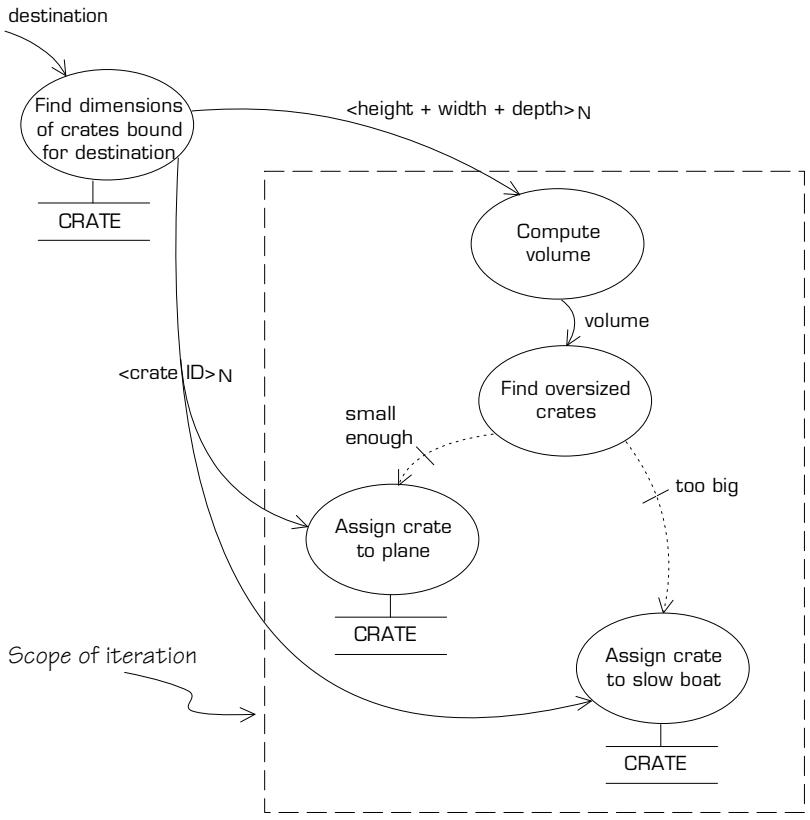


Figure 6.16.1

```
Crate( all ).( Height, Width, Depth ) | ComputeVolume | CheckSize? !Big, !Small;
!Big:      "Slow Boat" >Crate( ).Transport;
!Small:    "Plane" > Crate( ).Transport;
```

The test process, **`CheckSize`**, uses the computed volume to decide whether a crate is big or small, and then flows the reference to that crate to the appropriate guard. Hence, when there are multiple instances in a data flow, the test process divides the flow into two streams, each stream containing the appropriate references.

6.17 Failure to Find Instances

Failed Selection

The language-supplied test process **`None? !True, !False`** acts on references to determine whether an accessor failed, as in

Copyright 1997, Project Technology, Inc. All rights reserved

```
Cave( all , Temperature >21 ) | None? !isNone, !isSome;
!isNone:   ...
!isSome:   Cave( ).Pressure ....
```

Use the established reference to access data about an instance after checking for its existence:

```
!isSome:   Cave( ).Pressure     // gets the pressures of those caves known to be
hotter than 21°
```

Failed Traversal

The same notion can be used for failure to find an instance when traversing a relationship, as is possible when a relationship is conditional:

```
myBench -> [R30] Irradiation( one ) | None? !isNone, !isOne;
!isNone: ...
!isOne: ...
```

This example will execute the statements after the **isOne** guard when there is a related instance of Irradiation, and the statements after the **isNone** guard when there is not.

Failed Instance Creation

When creating an instance which has identifying attributes not of type arbitrary, the analyst must specify values for the identifying attributes. To ensure that an instance bearing the same identifier does not already exist, you can check prior to creation:

```
Bench( one, BenchNumber = ~BenchNumber ) | None?  !isNone, !isOne;
!isNone: ~BenchNumber >> Bench.( BenchNumber );
!isOne: Wormhole W3: ReportFailure( "You gave me the number of an existing Bench"
);
```

6.18 Current Time

A language-supplied process, named **CurrentTime**, produces a set of integer data values, one for each of year, month, day, hour, minute, second and a real number (less than one) for fractions of a second. Use **shuffle** to get the desired portions of the time, as in

```
CurrentTime | shuffle( 1, 2, 3 ) >~Date; or
CurrentTime | shuffle( 2, 3, 4, 5 ) >~ScheduleDate;
```

6.19 Summary

SMALL is a textual language for expressing processing that maintains the same high level of abstraction as the graphical form of the process model, the data flow diagram. Figure 6.19.1 shows a DFD, taken

from OL, that has an example of every DFD construct defined in the method. Here is the corresponding SMALL:

```
( Current Time, Self.EndTime ) | DetermineIfRampComplete? !Complete, !NotComplete;
!Complete: Self | Gen TR12: CompleteTemperatureRamp;
!NotComplete: [
    Self | Gen( 10 )TR13: RampTimerExpired;
    (CurrentTime, Self.(StartTime, EndTime, StartTemperature, EndTemperature) ) |
        ComputeDesiredTemperature > ~DesiredTemperature;
    (Self ->[R4->R3] CookingTank.ActualTemperature, ~DesiredTemperature ) |
        DetermineIfHotEnough? !HotEnough, !NotHotEnough;
    !HotEnough:       Self ->[R4->R3->R21] Heater | Gen H20: TurnOffHeater;
    !NotHotEnough:    Self ->[R4->R3->R21] Heater | Gen H21: TurnOnHeater;
]
```

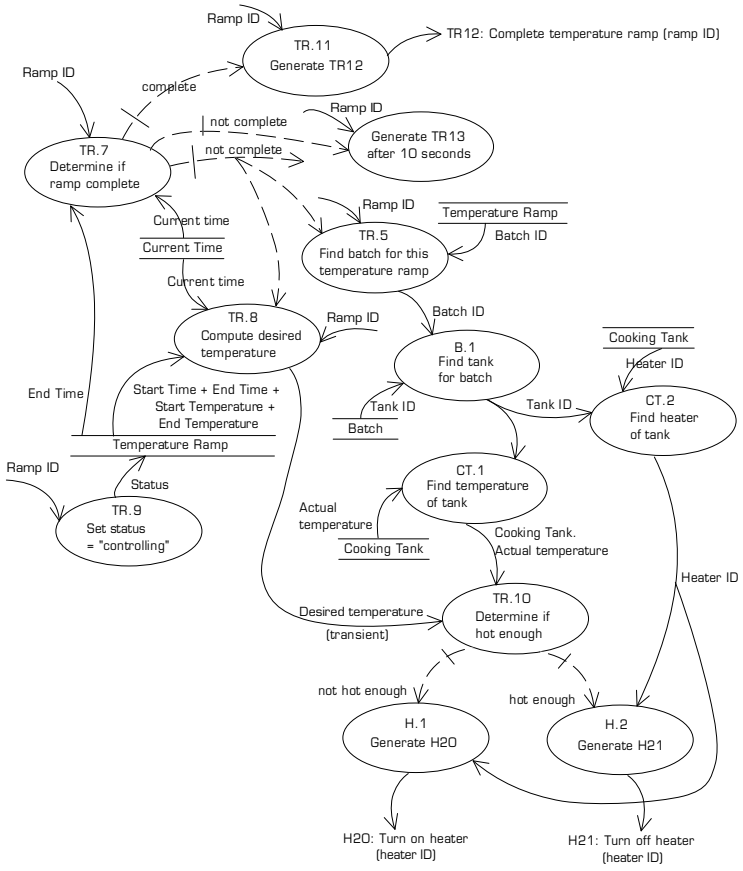


Figure 6.19.1

Acknowledgements. SMALL would not have been possible without *The OOA96 Report* produced with Neil Lang of Project Technology, and the contemporaneous discussions of merging and diverging dataflows—especially following a test process—with Howard Kradjel, now of Lucent Technologies, then of Project Technology. Neil also ensured that the language maintained consistency with *Object Lifecycles* and *The OOA96 Report*. He also provided valuable suggestions as well as continuous review.

The insight that 'chains' of processes always end in an event generator, a wormhole or a write is due to Peter Fontana and his colleagues at Pathfinder Solutions. Once it was clear that the data flow aspect of

the DFD could be understood textually, it became simpler to understand, and build, the remaining features. This was *the* breakthrough in the development of the language.

Thanks to Gregory Rochford and Cortland Starrett, both of Project Technology, for providing several parsers supporting the definition of SMALL.

Finally, thanks to John Wolfe and his team of Project Technology architects, and to John Yeager of Lucent Technologies, previously with Project Technology, for ensuring that the language could be translated and mapped to a wide range of architectures.

••••••••••••••••••