Pathfinder Solutions

Platform Independent Action Language

Version 2.2

December 31, 2004

PathMATE™ Series

Pathfinder Solutions LLC
33 Commercial Drive, Suite
Foxboro, MA 02035 USA
www.PathfinderMDA.com
508-543-7222

Copyright © 2004 by Pathfinder Solutions

Table of Contents

Preface

Audience

Platform Independent Action Language is for software engineers who want to learn how to design embedded systems with PathMATE. It's helpful but not essential to have some familiarity with IBM's Rational Rose modeler.

Related Documents

These PathMATE documents are available at www.PathfinderMDA.com, or from your Pathfinder account manager:

- *PathMATE Installation Guide*
- *Accelerating Embedded Software Development with a Model Driven Architecture* (white paper)
- *PathMATE: Model Automation and Transformation Environment for Embedded Systems* (online brochure)

Conventions

The *Quick Start Guide* uses these conventions:

- **Bold** is for clickable buttons and menu selections.
- *Italics* is for screen text, path and file names, and other text that needs special emphasis.
- `Courier` denotes code, or text in a log or a batch file.
- A **Note** contains important information, or a tip that saves you time.
- The scissors icon marks text that you must copy from this document and paste elsewhere.

What You'll Need

To complete the steps in this guide, you'll need the following software on your computer:

- Microsoft XP Professional Edition
- Rational Rose
- PathMATE software development toolkit
- Microsoft Visual C++ 6.0 or 7.0
- Plain text editor

How to Use this Guide

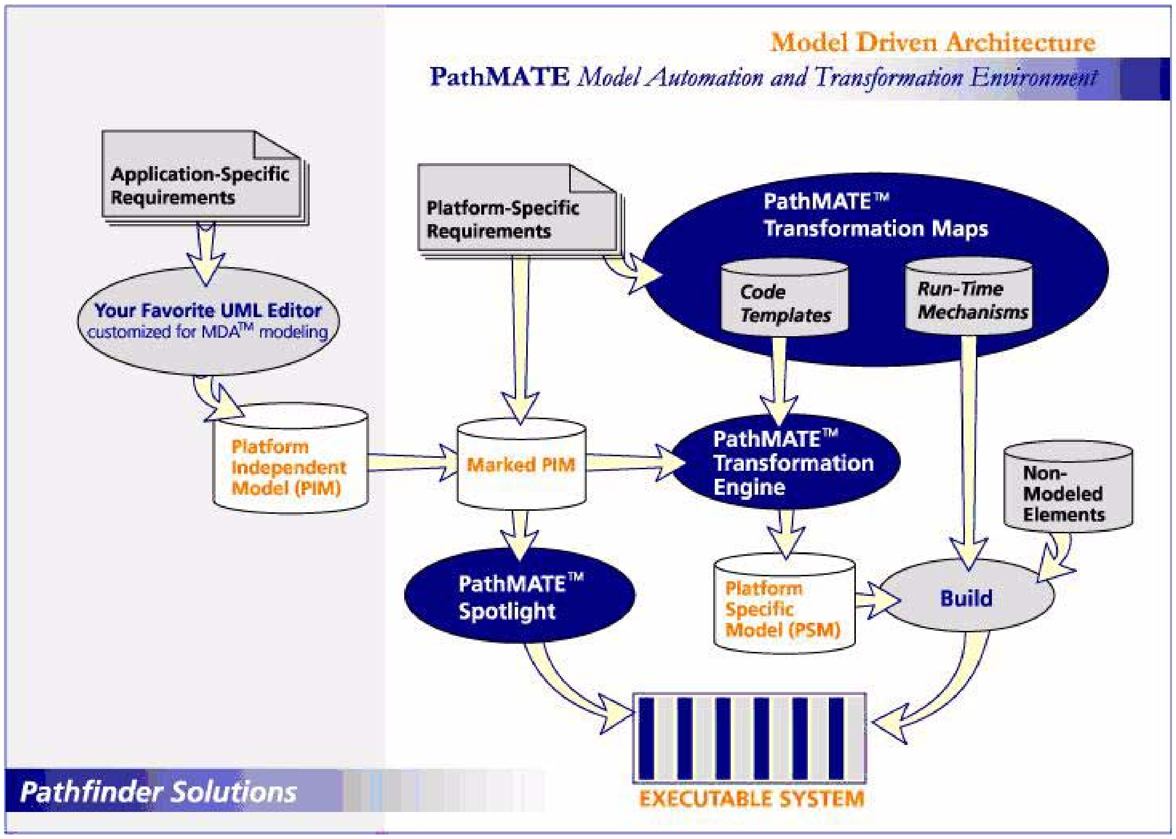
If you are not already familiar with the PathMATE toolset, read the overview in Section 1. If you have not installed the PathMATE toolset on your computer, turn to Section 2 to download the software from www.PathfinderMDA.com, then follow the installation instructions in Appendix A. After installation, carefully follow the steps in Section 3 to learn how PathMATE works.

PathMATE Overview

This overview introduces Model Driven Architecture (MDA) and the PathMATE™ tools that make MDA work. MDA and PathMATE move you from writing and debugging code to developing and testing the logic of a high performance system. Over years of rigorous refinement in several industries, PathMATE tools have proven their value in rapid and effective software systems development.

PathMATE Toolset

The PathMATE Model Automation and Transformation Environment includes all the tools required to transform your MDA models into high-performance systems. See the PathMATE workflow in the figure below.


Model Driven Architecture
PathMATE Model Automation and Transformation Environment
Application-Specific Requirements
Your Favorite UML Editor customized for MDA™ modeling
Platform-Specific Requirements
PathMATE™ Transformation Maps
Code Templates
Run-Time Mechanisms
Platform Independent Model (PIM)
Marked PIM
PathMATE™ Transformation Engine
Non-Modeled Elements
PathMATE™ Spotlight
Platform Specific Model (PSM)
Build
EXECUTABLE SYSTEM
Pathfinder Solutions

PathMATE Workflow

The three parts of the PathMATE toolset cooperate to turn your models into executable systems:

- *Transformation Maps* – Generate C, C++, or Java software with off-the-shelf Transformation Maps, or create custom maps to drive output for other languages or specific platforms.
- *Transformation Engine* – The Engine transforms platform-independent models into working, embedded software applications.
- *Spotlight* – Verify and debug your application logic with Spotlight, the most advanced model testing environment available.

No other MDA transformation environment offers a more open or configurable set of development tools, designed to meet the requirements of systems engineers.

How PathMATE Works

Use Model Driven Architecture to build complex embedded systems that meet rigorous standards for speed and reliability. MDA works because it separates what the system does from its deployment on a particular platform. PathMATE adds these advantages:

- *Greatest architectural control* – A highly configurable Transformation Engine enables you to optimize output for resource-constrained platforms.
- *Clean separation of model and code* – Conforming to the MDA paradigm, PathMATE models contain no implementation code. That gives you fast and flexible deployment and migration capabilities.
- *Configurable, target-based model execution and testing* – Preemptively eliminate platform-specific bugs, minimize quality assurance resources, and accelerate development.
- *Lowest cost of ownership* – Integrate PathMATE with your existing UML editor. Build on your previous investment in training and software.
- *Speed* – Even large transformations take just seconds with PathMATE. That enables highly iterative model development, and rapid transformation and test cycles.

Try the demonstration software available at *www.PathfinderMDA.com* to get started quickly and easily.

1. Introduction

This document provides a summary of the Platform Independent Action Language (PAL) as applied in Model Driven Architecture using the Unified Modeling Language. PAL is a platform independent form of behavioral expression: a programming language for MDA Platform Independent Models (PIMs) using the UML Standard Action Semantics standardized by the Object Management Group. Please see "UML Action Semantics Revised Final Submission," available from www.omg.org. Through its focus on platform independent model constructs, action language:

- Is concise and easy to learn, with a familiar, C++-like syntax.
- Provides the most convenient form of expression for PIM action procedures.
- Offers strategic agility through implementation platform independence and implementation language independence.
- Effectively enforces PIM separation from implementation code.
- Enables the highest degree of freedom to apply varying and project-specific implementation architecture and optimizations through transformation.

It is assumed that the reader is somewhat familiar with MDA/UML modeling conventions as specified in the Model Based Software Engineering (MBSE) approach for modeling with the UML, as introduced in:

> *Model Based Software Engineering: Rigorous Software Development with Domain Modeling*, Pathfinder Solutions. (This paper is available from www.pathfindermda.com.)

The *Action Language Quick Reference* syntax summary is provided for your convenience on page 20. Print it separately and keep it handy.

2. Action Overview

What Are Actions?

Actions are procedures in your UML models. More specifically, they enact the bodies of services and states specified in an analyzed domain. In addition, actions are provided to perform initialization at the system and domain levels. Briefly, then, actions define and implement:

- Domain services
- Class services
- State actions
- System and domain initialization

Domain and class services run when they are invoked, and state actions run when the instance enters the state.

What Can Actions Do?

Service, state, and domain initialization actions execute within the scope of their owning domain. These actions may access data atoms from various sources:

- Service parameters
- Event parameters
- Local variables
- Class attributes

Actions can read and write these data atoms, create class instances, generate events, invoke services, create and invoke service handles, and link, unlink and navigate associations. Actions associated with a domain can access any of the classes in the domain. To reach outside the domain, the action must call a service of the domain. An action in one domain cannot access classes from another domain.

What Makes Up an Action?

An action is very similar in execution semantics to a function from a procedural programming language. It is made up of blocks of statements. The action itself has a root block of statements, and certain statements have blocks nested within them.

Each statement is made up of expressions and keywords. Expressions are accessors to analysis elements, local variables, compound expressions (with operators), or literals.

How are Actions Used?

Actions, along with all other Analysis elements in your system, are fed into a translation step where they are mapped to executable implementation code.

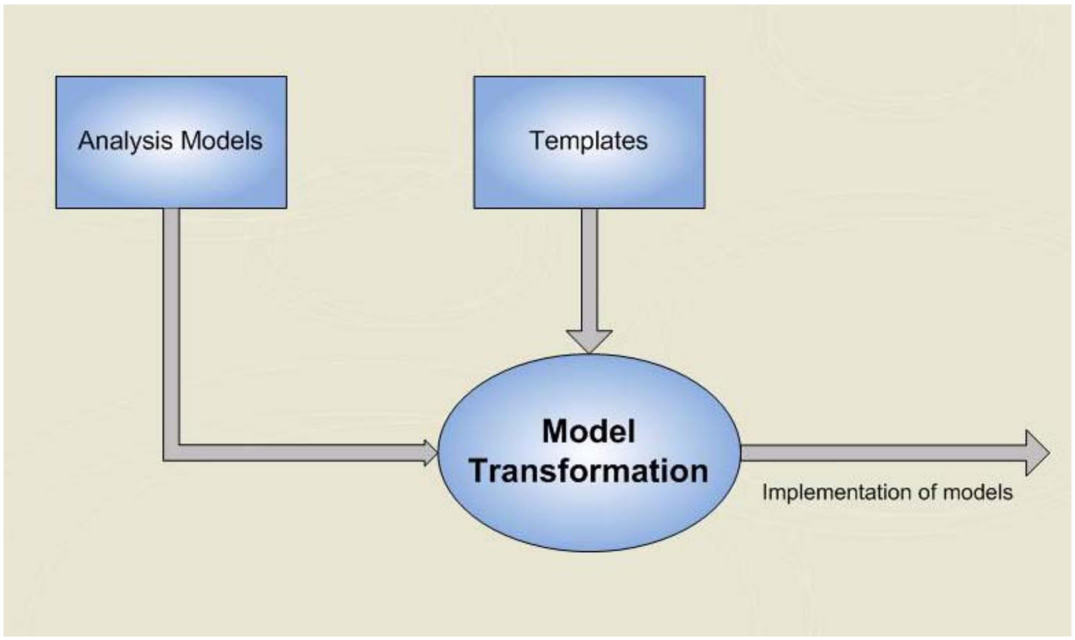


Figure 2-1. Model Transformation

A transformation rule maps each specific Action Language statement and expression type to the implementation language. Some statements may have more than one possible mapping, allowing for optimization. The implementation of an action is translated from the Action Language.

3. Action Semantics

Action Language (AL) is a programming language with specific primitives to support the manipulation of Analysis elements. To describe Action Language syntax, this document uses the following conventions:

- [*optional item*]{*either* | *or*} *0 or more iterations*, ...
- All bolded characters (such as **{|}, []**) indicate actual use of these characters in the action language.
- *Italic* items are substitution items or annotations.
- All action language keywords are case sensitive, and are shown in **BOLD**.
- // In action language, comments are as in C++.

Data Context

Each action has a varying set of data atoms that it reads and/or writes.

Explicit

Each domain or class service may have parameters available. Services may also have a return value. State actions have event parameters. All actions may declare and use local variables.

Implicit

Instance-based class services, or instance-based state actions have the “this” variable available as a Reference to the target instance. Literals may be used. FIND accessors over entire instance populations (FIND CLASS) imply the use of a domain-global collection of instance references.

Data Types

There is a fixed set of data atoms in Analysis: attribute, service parameter, event parameter, and action local variable. Each atom is of a specific data type. There is a core set of basic, built-in data types:

- Boolean: TRUE or FALSE
- Character: an ASCII character
- Integer: whole number (width is design dependent)
- Handle: generic reference (similar to void* in C)
- Real: floating point number (size is design dependent)
- String: a variable length ASCII string
- GenericValue: stores a String, Real, Handle, or Integer (similar to C union)

In addition there are advanced data types:

- Ref<class>: a reference to an instance of <class>, commonly used as a type for a local variable used to iterate over the results of a Find or Navigate
- Group<base type>: an ordered set of <base type> items, commonly used as a type for a service parameter to support passing sets of data items between domains
- GroupIter<base type>: an iterator over an ordered set of <base type> items, used to iterate over items in a group.

```
[ Boolean | Character | String | Real | Integer |
GenericValue | Handle | Group<base_type> |
GroupIter<base_type> | Ref<class_name> |
ServiceHandle]
```

The analyst can define new types:

- Enumerations
- Aliases base types (similar to typedef in C)

Statements

Statements combine expressions to accomplish specific tasks within actions.

Data Manipulation

Assignment - writes the value of the expression on the right of the equal side into the data atom on the left:

```
{ AttributeAccessor | Parameter | LocalVariable |
ServiceHandleParameter } = Expression ;
```

Local Variable Declaration – declares a local variable (scope limited to declaring action):

```
DataType variable_name { = initial_value };
```

Constant Declaration – in the system or domain initialization action declare a constant. Constants defined in the system initialization action are accessible to all domains. Constants defined in the domain initialization action are accessible only to the domain where they are defined.

```
CONST DataType variable_name  = initial_value;
```

External Constant Declaration – in the system or domain initialization action declare a constant that is defined in realized code. The action language parser will recognize an external constant but will not create a definition for it. External constants defined in the system initialization action are accessible to all domains. External constants defined in the domain initialization action are accessible only to the domain where they are defined.

```
EXTERN CONST DataType variable_name;
```

NOTE

Some designs such as the Pathfinder C++ Design components support an IncludeFile property that contains the name of a realized include file containing the definition of the external type.

Data Atom Ordering - sorts the specified list of data atoms based on their value. "/" indicates ascending order (default), or "\" indicates descending order:

```
ORDER GROUP [{ / | \ }] group;
```

Instance List Ordering – These statements sort the specified class instance population based on the specified attribute(s). The attributes can be preceded by "/" to indicate ascending order (default), or "\" to indicate descending order. The most significant key is specified first:

Class Population Ordering – sorts the specified class instance population:

```
ORDER CLASS class_name BY ([{ / | \ }]
attribute_name, …);
```

Association Population Ordering – sorts the associated instance population (Navigation is an association navigation expression):

```
ORDER Navigation BY ([{ / | \ }] attribute_name,
…);
```

Data ordering is not maintained when new elements are added to the group or instance population. For example, if an ORDER statement was executed and a subsequent action created a new instance of the class, the sort order specified by the ORDER statement would not be maintained. If you want a sort order to be maintained, use the Sort design properties.

Execution Flow Control

Action Language contains conditional and iterative execution flow control constructs.

Statement Block – A statement block is a sequence of statements.

Instance List Iteration – These statements declare a cursor variable, and then iterate over each class instance in the specified population. Each instance is assigned to the cursor variable, and the nested statement block is executed. An optional WHERE clause filters the instance set to only those that match the specified Boolean expression comparing attributes of the target class with any data atoms available in the action context.

Class Population Iteration - iterate over the entire class population:

```
FOREACH cursor_variable = CLASS class name [ WHERE
(Expression) ] { StatementBlock }
```

Association Population Iteration - iterate over the associated instance population (Navigation is an association navigation expression):

```
FOREACH cursor_variable =  Navigation [ WHERE
(Expression) ] { StatementBlock }
```

Conditional – execute the first statement block if the Boolean Expression is TRUE, otherwise execute the optional else statement block.

```
IF (Boolean Expression) { StatementBlock }
[ ELSE IF (Boolean Expression) {StatementBlock} ]
[ ELSE { StatementBlock } ]
```

Iterative – evaluate the specified Boolean expression – if it is TRUE, execute the statement block. Repeat until the expression is FALSE, or a BREAK statement is encountered.

```
WHILE (Expression)  { StatementBlock }
```

Break – interrupt execution of the enclosing iterative control structure (WHILE or FOREACH). Skip all remaining statements in the iterative statement block, and resume execution after the iterative statement block:

```
BREAK;
```

Continue - interrupt execution of only this iteration of the enclosing iterative control structure (WHILE or FOREACH). Skip all remaining statements in the iterative statement block, and resume execution at the top of the iterative statement block:

```
CONTINUE;
```

Function Ins and Outs

Invocation – call the specified service or built-in method (with no return value):

```
{ Service Accessor | BuiltIn method};
```

Service Value Return – return from this service with the specified return value:

```
RETURN [ Expression ];
```

Expressions

An expression is something that provides a data value, receives a data value (when it is written to), and/or performs some action. Expressions are used to create, store and access data values and Analysis elements. Expressions are also used to invoke services and access built-in capabilities.

Variables

Local Variable Reference – used after its declaration:

variable_name

Parameter – service parameters can be referenced in the service action; event parameters can be referenced in the state action:

parameter_name

Constants

Constant Reference – system or domain scoped constants defined in the initialization action may be used within the scope of the constant:

constant_name

Accessors

Accessors are expressions that read or write specific Analysis data atoms.

Class and Attribute Accessors

Attribute – read or write a class attribute value. The instance_ref can be a local variable, a service or event parameter, or the "this" reference in instance-based class services or states:

```
instance_ref . attribute_name
```

Class Instance Create – create an instance of the specified class and return a reference to it. A leaf subtype must be specified (no supertypes). Attribute values must be specified if there is no default. An initial state name must be specified if there is no default (a default state is specified by the initial state bullet on a state model):

```
CREATE class_name ( [ attribute_name = Expression,
… ] ) [ IN initial_state ]
```

Class Instance Delete – unlink the specified instance from all associations it participates in, and remove it:

```
DELETE instance_ref
```

NOTE

The design ensures that an instance is unlinked from all its associations before deletion.

Class-Based Find – Find the first (default) or last instance of the specified class. An optional WHERE clause filters the instance set to only those that match the specified Boolean expression comparing attributes of the target class with any data atoms available in the action context. If no matching instance is found, NULL is returned:

```
FIND [ { FIRST | LAST } ] CLASS class_name [ WHERE
(Expression) ]
```

Navigation-Based Find – Find the first (default) or last instance through the specified chain of association navigations. An optional WHERE clause filters the instance set to only those that match the specified Boolean expression comparing attributes of the target class with any data atoms available in the action context. If no matching instance is found, NULL is returned:

```
FIND [ { FIRST | LAST } ] Navigation [ WHERE
(Expression) ]
```

WHERE expressions are Boolean expressions that can include any data atom available from the action context: local variables, constants, and parameters. It also includes attributes of the target instance. However a WHERE expression cannot perform other instance-based accesses of the target instance, including instance-based operation invocations, or association accesses (navigation).

Relationship Accessors

Association Link – establish a connection between the specified class instances. If the association is reflexive (the same class at both ends), then at least one role phrase must be specified. If the association has an associated class, an associated class instance reference must be provided.

```
LINK [ @role_phrase1 ] instance1_ref
A<number>  [ @role_phrase2 ] instance2_ref
[ASSOCIATIVE assoc_ref ]
```

Unlink - break the connection between the specified class instances. . If the association is reflexive (the same class at both ends), then at least one role phrase must be specified. If the association has an associated class instance connected, this instance is deleted automatically by unlink. Do not delete the associated class instance prior to or after the unlink.

```
UNLINK [ @role_phrase1 ] instance1_ref
Anumber [ @role_phrase2 ] instance2_ref
```

Navigation Expressions – There are used to traverse associations and inheritance relationships (downward only). A Navigation Expression may return no instances, a single instance, or a collection of instances – all depending on the multiplicity of the associations in the Navigation Expression. A Navigation Expression may contain multiple individual navigations chained together with the across operator "->". A Navigation Expression cannot be used as a class instance reference expression itself – it is only used in the context of a FIND accessor, FOREACH statement, or ORDER statement.

SubSuper Navigation – "downcast" to get from a supertype to a specific subtype. Returns NULL if the actual subtype encountered at run-time subtype does not match specified subtype. Upcasting is performed automatically. A subtype can be used anywhere a supertype is expected.

```
supertype_reference
->Srelationship_number->subclass_name
```

Association Binary Navigation – Navigate from the start_ref class instance across the specified association to the instance(s) at the other end:

```
[ @role_phrase1 ] start_ref->A<number> [
->@role_phrase2 dest_class_name]
```

Association Navigation to Associated Class – Navigate to the instance of the class associated with a link between class instances start_ref_1 and start_ref_2:

```
[ @role_phrase1 ] start_ref_1 AND [ @role_phrase2
] start_ref_2 ->A<number>
```

Event

Generate – create an instance of the specified event, and queue it for dispatch to the specified instance. No destination is provided for create events. Destination is optional for self-directed events sent to self from an instance state. All event parameters must have a value provided. If a delay is specified, the event will be held in the delayed event mechanism for a minimum of the period specified, and then it will be queued for dispatch. The units of the delay are Design specific:

```
GENERATE event_name ( Expression, … ) [ AFTER
(delay) ] [ TO (destination_ref) ]
```

***Cancel* –** If an instance of this event destined to the specified destination is still held in the delayed event mechanism, then remove it before transmission. If more than one instance of this event is outstanding against the specified destination, delete the one with the shortest delay remaining. No indication is returned if this operation actually found an instance of the event.

CANCEL event_name [**TO (**destination_ref**)**]

***ReadTime* –** If an instance of this event destined to the specified destination is still held in the delayed event mechanism, return the amount of time remaining until it will be queued (the units are Design specific). Returns 0 if a matching event instance is not found.

TIME UNTIL event_name [**TO (**destination_ref**)**]

Service

Domain Service Invocation – may have a return value:

domain_prefix:service_name**(**Expression, …**)**

Class Service Invocation (class based) – may have a return value:

class_prefix**:**service_name**(**Expression, …**)**

Class Service Invocation (instance based) – may have a return value:

instance_ref **.** [class _prefix:
]service_name**(**Expression, …**)**

NOTE

Always omit the class prefix when calling a polymorphic service on a supertype instance.

ServiceHandle

Create ServiceHandle – create a ServiceHandle to a specific service (only valid to services in context domain). Input parameter values must be specified if there is no default:

CREATE ServiceHandle ([*parameter_name* **=**
Expression, …] **) TO** { *domain_prefix* |
class_prefix }**:***service_name*

***Invoke ServiceHandle* –** optionally specify input parameter values:

CALL *service_handle* **(** [*parameter_name* **=**
Expression, …] **)**

***ServiceHandle Parameter* –** a ServiceHandle parameter can be directly referenced for read or write using the parameter name as an index (error behavior is Design-specific):

service_handle **[** parameter_name **]**

Built-In Methods

Invocation – built-in methods are invoked as methods of their operand:

```
Operand.method_name(parameters…)
```

Group built-ins - Group data types have the following support methods:

Add an item in front of the first item in the group:

```
group_expression.addFront(item)
```

Add an item after the last item in the group:

```
group_expression.addBack(item)
```

Add an item after the current position (indicated by the iterator):

```
group_expression.insert(iter, item)
```

Return the first item in the group:

```
group_expression.front()
```

Return the first item in the group:

```
group_expression.back()
```

Remove the first item with the specified value from the group:

```
group_expression.remove(item)
```

Delete the item at the iterator location from the group:

```
group_expression.erase(iter)
```

Remove all items from the group:

```
group_expression.removeAll()
```

Return an integer specifying the number of items in the group:

```
group_expression.size()
```

Return the specified item in the group (0-based index; error behavior is Design-specific):

```
group[index]
```

***GroupIter Built-Ins* -** GroupIter data types have the following support methods:

Establish the base group for the iterator – required before any other iterator operations:

    group_iter_expression.**setGroup(***group***)**

Reset the iterator to the front of the list:

    group_iter_expression.**front()**

Return the iterator to the back of the list:

    group_iter_expression.**back()**

Return the current item in the group:

    group_iter_expression.**current()**

Increment the iterator's position in the group, and return the new current item in the group:

    group_iter_expression.**next()**

Decrement the iterator's position in the group, and return the new current item in the group:

    group_iter_expression.**previous()**

Return a boolean indicating if the last next() operation has advanced past the end of the list, or if the last previous() operation has advanced past the beginning of the list:

    group_iter_expression.**finished()**

Expression Mechanics

Binary Expression – has two operand expressions combined by an operator. Binary expressions can be nested, and grouped with parenthesis:

```
[ ( ] Expression Operator Expression [ ) ]
```

Arithmetic Binary Operators

+ plus
- minus
***** multiply
/ divide
% modulus

Bitwise Binary Operators

& and
^ exclusive or
| inclusive or
<< left shift
>> right shift

Boolean Binary Operators

< less than
<= less than or equal to
> greater than
>= greater than or equal to
&& and
|| or
== equal to
!= not equal to

Unary Expression –

```
[ ( ] UnaryOperator Expression [ ) ]
```

Unary Operators

+ arithmetic positive
- arithmetic negative
~ complement
! Boolean not

Literals

Boolean Literal

```
{ TRUE | FALSE }
```

Character Literal

A single character:

```
'character'
```

Integer Literal

One or more digits:

```
digit…
```

Invalid Class Instance Reference

```
NULL
```

Uninitialized or Invalid ServiceHandle Reference

```
EMPTY_SERVICE_HANDLE
```

Real Literal

For example, 3.45 or 3.45e-6:

```
IntegerLiteral . IntegerLiteral [ e [ - ]
IntegerLiteral]-
```

String Literal

Use \ to embed a double quote ":

```
"character…"
```

Attaching Design Properties to Statements

Design Properties – each statement may have one or more name value pairs accessible to the design templates. Properties are defined in curly braces after the statement semicolon or closing curly brace. The property name must begin with a letter followed by a letter, number or underscore.

```
statement ; [{ property_name = "property_value",
…}]
```

4. Examples

The best place to see Action Language examples is in a sample system. The fully executable Robochef system is available from Pathfinder Solutions.

The SystemInit service of the FoodPrep domain contains examples of object creation and association linking. Here is a portion of this service action:

```
Ref<ContainerCache>   cache;
Ref<Oven>             oven;
Ref<Dishwasher>       dishwasher;
Ref<Mixer>            mixer;
Ref<Dispenser>        dispenser;

// Set up the conveyance domain
CNV:Initialize();

// Create the appliances and start initialization
cache = CREATE ContainerCache
(type=RC_APPL_TYPE_CONTAINER_CACHE,
deviceHandle=201) IN Created;
GENERATE CC:Initialize() TO (cache);

oven = CREATE Oven(type=RC_APPL_TYPE_OVEN,
deviceHandle=202) IN Created;
GENERATE OVN:Initialize() TO (oven);

dishwasher = CREATE
Dishwasher(type=RC_APPL_TYPE_DISHWASHER,
deviceHandle=203) IN Created;
GENERATE DW:Initialize() TO (dishwasher);

mixer = CREATE Mixer(type=RC_APPL_TYPE_MIXER,
deviceHandle=204) IN Created;
GENERATE MIX:Initialize() TO (mixer);

dispenser = CREATE
Dispenser(type=RC_APPL_TYPE_DISPENSER,
deviceHandle=205) IN Created;
```

```
GENERATE DIS:Initialize() TO (dispenser);

// more …. – see Robochef example for full state
action
```

The DeterminingIfMoreOkSteps state action of the ActiveRecipe class in the FoodPrep domain contains examples of finding across a reflexive association, linking, unlinking, and event generation.

```
Ref<RecipeStepSpec> current_step;
Ref<RecipeStepSpec> next_step;
// find the current state being executed
current_step = FIND this -> A10;
// check to see what the next step is
next_step = FIND FIRST current_step -> A6 ->
@next_success_step RecipeStepSpec;
// if there is a next step
IF (next_step != NULL)
{
   // associate active recipe with its next step
   UNLINK this A10 current_step;
   LINK this A10 next_step;
   // perform the next step
   GENERATE AR:PerformNextRecipeStep();
}
ELSE
{
   // no more steps – recipe is complete
   GENERATE AR:Done();
   status = DONE_STATUS;
}
```

The performOperation service of the Appliance class demonstrates how to create service handles.

```
ServiceHandle  actionFailed;
ServiceHandle  actionSucceeded;

actionSucceeded = CREATE ServiceHandle(this =
this) TO APPL:opComplete;
actionFailed = CREATE ServiceHandle(this = this)
TO APPL:opFailed;
AI:ApplianceRequest(deviceHandle, action, data1,
data2, data2, data4, actionSucceeded,
actionFailed);
```

The following is an example of how to use groups:

```
Group<String>   names;
// adding elements
names.addBack(“Joe”);   // Joe
names.addFront(“Mary”); // Mary, Joe
names.addBack(“Sue”);   // Mary, Joe, Sue
// accessing elements
String name1 = names.front();  // Mary
String name2 = names[1];       // Joe
String name3 = names.back();   // Sue
```

The following is an example of how to use group iterators to iterate through a list:

```
Group<String>        names;
// iterate from end to beginning
GroupIter<String>    cursor;
cursor.setGroup(names);
cursor.back();
WHILE(!cursor.finished())
{
   // get name at current iterator position
   String current_name = cursor.current();
   // … do something with name
   // advance to the next item in the group
   cursor.previous();
}
```

A. Action Language Quick Reference

The last page contains a quick reference guide that you can print out and refer to while writing action language.

[optional item]{either | or}0 or more iterations, …
All bolded characters (such as **{|}**, **[]**) indicate actual use of these characters in AL.*Italic* items are substitution items or comments.
All AL keywords are case sensitive;// comments like C++; An Action Procedure has event or service Parameters and a StatementBlock.

Statement

Assignment	{ AttributeAccessor | Parameter | LocalVariable | ServiceHandleParameter } **=** Expression **;**	
Break	**BREAK;**	
Continue	**CONTINUE;**	
LocalVarDecl	DataType *variable_name* { **=** *initial_value* }**;**	
ConstantDecl	[**EXTERN**] **CONST** DataType *variable_name* [**=** *initial_value*] **;**	- valid only in system or domain init. action
Invocation	{ Accessor | BuiltIn method}**;**	- a procedure call with no return value
ForEach (*class*)	**FOREACH** *cursor_variable* **= CLASS** *class name* [**WHERE (**Expression**)**] **{** StatementBlock **}**	
ForEach (*nav*)	**FOREACH** *cursor_variable* **=** Navigation [**WHERE (**Expression**)**] **{** StatementBlock **}**	
If	**IF (**Boolean Expression**) {** StatementBlock **}** [**ELSE IF** **{** StatementBlock **}**] [**ELSE {** StatementBlock **}**]	
Return	**RETURN** [Expression]**;**	
StatementBlock	Statement…	
While	**WHILE (**Expression**)** **{** StatementBlock **}**	
Order (*class***)**	**ORDER CLASS** *class_name* **BY (**[{ **/ | \** }] attribute_name, …**);**	
Order (*nav*)	**ORDER** Navigation **BY (**[{ **/ | \** }] attribute_name, …**);**	
Order (*group*)	**ORDER GROUP** [{ **/ | \** }] *group***;**	

Accessors

AttributeAccessor	instance_ref **.** attribute_name	
CreateServiceHandle	**CREATE ServiceHandle (** [parameter_name **=** Expression**,** …] **) TO** { domain_prefix | class_prefix }**:**service_name - returns a service handle	
GroupItemIndex	group**[**index**]**	- 0-based index
InvokeServiceHandle	**CALL** service_handle **(** [parameter_name **=** Expression**,** …] **)**	- no return
DomainServiceInvocation	domain_prefix**:**service_name**(**Expression**,** …**)**	- may have a return value
ClassServiceInvocation	class_prefix**:**service_name**(**Expression**,** …**)**	- may have a return value
InstanceServiceInvocation	instance_ref **.** [class _prefix**:**]service_name**(**Expression**,** …**)**	- may have a return value
SubSuperAccessor	supertype_reference **->S**relationship_number**->**subclass_name	- performs a "downcast"
AssociationAccessor	[Link | Navigate | Unlink]	
Link	**LINK** [@role_phrase1] instance1_ref **A**<number> [@role_phrase2] instance2_ref [**ASSOCIATIVE** assoc_ref] - no return	
Navigation (binary)	[@role_phrase1] start_ref**->A**<number> [**->**@role_phrase2 dest_class_name]	
Navigation (to assoc class)	[@role_phrase1] start_ref_1 **AND** [@role_phrase2] start_ref_2 **->A**<number>	
Navigation (downcast)	supertype_instance_ref**->S**<number>**->**<subclass_name>	
Unlink	**UNLINK** [@role_phrase1] instance1_ref **A**number [@role_phrase2] instance2_ref	
EventAccessor	{ Cancel | Generate | ReadTime }	
Cancel	**CANCEL** event_name [**TO (**destination_ref**)**]	
Generate	**GENERATE** event_name **(** Expression**,** … **)** [**AFTER (**delay**)**] [**TO (**destination_ref**)**]	
ReadTime	**TIME UNTIL** event_name [**TO (**destination_ref**)**]	- returns time remaining for delayed ev.
ClassAccessor	{ Create | Delete | Find }	
Create	**CREATE** class_name **(** [attribute_name **=** Expression**,** …] **)** [**IN** initial_state]	- returns inst. reference
Delete	**DELETE** instance_ref	
Find (class)	**FIND** [{ **FIRST** | **LAST** }] **CLASS** class_name [**WHERE (**Expression**)**]	- returns an inst. reference or NULL
Find (nav)	**FIND** [{ **FIRST** | **LAST** }] Navigation [**WHERE (**Expression**)**]	- returns an inst. reference or NULL

Expression

accessors	AttributeReference, EventAccessor, ClassAccessor, AssociationAccessor, SubSuperAccessor BinaryOpExpression, Expression Operator Expression	
BooleanConstant	{ **TRUE** | **FALSE** }	
CharacterConstant	**'***character***'**	
IntegerConstant	*digit…*	*- one or more digits*
invalid reference	**NULL, EMPTY_SERVICE_HANDLE**	
LocalVariable	*variable_name*	
Parameter	*parameter_name*	
RealConstant	IntegerConstant **.** IntegerConstant [**e** [**-**] IntegerConstant]- *3.45 or 3.45e-6*	
ServiceHandleParameter	*service_handle* **[** *parameter_name* **]**	
StringConstant	**"***character…***"**	*- use \ to embed "*
UnaryOpExpression	UnaryOperator Expression	

Operators

Arithmetic	**+ - * /%**	Bitwise	**& ^ |**
Boolean	**< <= > >= && || == !=**	Unary	**+ - ~ !**

DataTypes

[Boolean | Character | String | Real | Integer | GenericValue | Handle | Group<base_type> | GroupIter<base_type> | Ref<*class_name*> | ServiceHandle | UserDefined enumeration | UserDefined typedef]

BuiltIn Methods

- invoke via <expression>.<method>(args)

Group	{ **addFront(***item***)** | **addBack(***item***) | insert(***iter, item***) | front() | back()** | **remove(***item***)** | **erase(***iter***)** | **removeAll()** | **size()** }
GroupIter	{ current() | next() | previous() | finished() | front() | back() | setGroup(*group*) }

Index

Page numbers in the PDF version of this index are hyperlinks. In Acrobat, place your cursor over the number and click to go directly to the page.