Transformation Engine User's Guide

Version 4.0

April 29, 2004

PathMATE™ Series

Pathfinder Solutions LLC
90 Oak Point
Wrentham, MA 02093 USA

www.PathfinderMDA.com
508-384-1392

Table Of Contents

Preface

Audience

The *Transformation Engine User's Guide* is for software engineers who want to use PathMATE to create high performance systems.

Related Documents

These PathMATE documents are available at www.PathfinderMDA.com:

- *Accelerating Embedded Software Development with a Model Driven Architecture* (white paper)
- *PathMATE: Model Automation and Transformation Environment for Embedded Systems* (online brochure)
- *PathMATE MDA Mentor Services* (data sheet)

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.

For New Users

If you are not familiar with the PathMATE toolset, read the overview that begins on page iv. If you have not installed the PathMATE toolset on your computer, obtain a password from your account manager and download the software from www.PathfinderMDA.com. After installation, use the *PathMATE Quick Start Guide* to become familiar with the software tools.

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 an embedded system. Over years of rigorous refinement in several industries, PathMATE tools have proven their value in rapid and effective embedded systems development.

PathMATE Toolset

The PathMATE Model Automation and Transformation Environment includes all the tools required to transform your MDA models into high-performance embedded systems (Figure 1).

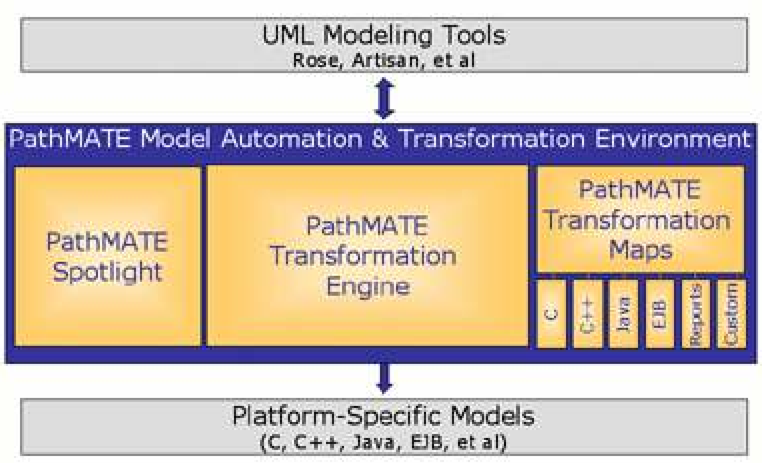
UML Modeling Tools
Rose, Artisan, et al
PathMATE Model Automation & Transformation Environment
PathMATE Spotlight
PathMATE Transformation Engine
PathMATE Transformation Maps
C
C++
Java
EJB
Reports
Custom
Platform-Specific Models
(C, C++, Java, EJB, et al)

Figure 1. PathMATE Toolset

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

- *Transformation Engine* – The Engine transforms platform-independent models into working, embedded software applications.
- *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.
- *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 embedded 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.

1. Introduction

The PathMATE Transformation Engine extracts the semantic information contained in your UML analysis models and presents it in textual form via a flexible and simple template notation. The Engine gives you the ability to completely navigate your model information and shape it into code, documentation, and reports according to your unique needs. This power and flexibility makes the Engine a development environment component of cornerstone importance for any projects applying the Unified Modeling Language (UML™).

This *User's Guide* is designed to help the analyst understand how to use the Engine to transform analysis information through templates. The reader is assumed to have a working knowledge of model based software engineering. They are also expected to understand their model editor, such as Rational Rose, and to be familiar with the Unified Modeling Language.

"Model Based Software Engineering: Rigorous Software Development with Domain Modeling," introduces you to the terminology, concepts and conventions you need to know to understand the contents of this guide. The paper is available at www.PathfinderMDA.com.

2. Overview Of Model Transformation

Considering the task of transformation at the highest level, the goal is to establish a set of instances of target documents with a set of UML analysis data according to specified templates. The templates specify the specific analysis information required, how it is arranged, and in what context.

The above paragraph helps us identify the major tasks and components of a transformation:

- Develop a set of templates to define our target document set.
- Employ the Engine to validate the correctness of the new templates.
- Apply the proven templates with the Engine to produce the Target Document Set.

Establishing Templates

This step focuses on the form of the result - the target documents. This form is specified in the Engine template notation. The tasks are to identify the set of target documents required, establish a typical prototype instance for each element of the set, and then convert the prototype into an template.

Identify Target Document Set

A Target Document set can be as simple as a custom report identifying all object attributes with blank definitions, or as intricate as the complete set of application-specific source code for your system. To develop an effective set of templates, you must first know what they are for.

- Establish the mission of the Target document set
- Gather any relevant requirements, such as coding standards and base class headers, report format guidelines, etc.
- Establish the overall structure of the set via a class hierarchy diagram, report topology description, etc.

Establish Target Prototype

Constructing a template from scratch using referential field specifiers without an example to work from can be difficult. It is recommended that a typical instance for each individual type of Target Document in your Set first be constructed by hand.

- For each Target with a different form, select a single instance of that Target based on your real analysis information.
- Construct the corresponding document by hand.
- Test it – for instance for a code template you could compile, link against the implementation elements, and run it (then fix the prototype if necessary).

Convert Prototype into Template

Once you are confident your prototype Target Document is correct, you can then generalize this into a template:

- Determine the context of the document - does it correspond to the entire system, a domain, object, attribute, event ... ? Specify the context as a set of parameters at the top of the template along with its name.
- Go through the body of the prototype, and for each element of analysis information, substitute the correct analysis element expansions, relative to the parameters you established at the top of the template, or to other variables you've used elsewhere (above this point) in this template.
- In sections where you referenced complete sets of analysis elements - such as all attributes of an object - use a list iteration construct.
- In areas where your prototype instantiates only one of a possible set of alternatives, use an if-else block to capture the choices.
- You may break off portions of the prototype into separate templates, or you may wish to chain your document together hierarchically, using template invocations to make the parametric connections.
- Arrange the templates as you would with any structured product – one per file – perhaps in a directory structure.

Verify Template Correctness

Once the templates are in place, you should test each one. Start from the bottom - where no other templates are invoked - and make sure each has valid notational syntax by invoking the Engine with the –`parse_only` switch. Then run through again extracting model information, and verify that component produces correct results. Use a limited amount of data initially before going out and attacking your entire system as a test model.

Developing and refining templates is a form of software development, and for large or complicated Target Document Sets you should allocate sufficient time to accomplish the task in an orderly and thorough manner - including testing and review time. Automatically translated templates are a component of a very powerful technology, leveraging a small amount of creative work across the bulk of an entire system. This can mean widely and consistently applying sound engineering, or equally vast propagation of error.

Produce Target Document Set

When the templates are completed, you can produce the desired Target Document Set. This may be an ongoing process itself - such as generating code against changing versions of the analysis. It could be to fill a spot need – such as generating a custom report. It could also be a one-time task – such as generating the review materials for a high level review.

For more regular use of a set of templates on a project-wide basis, place them under version control and make them available to the project from a standard location. Provide scripts that apply the templates in a uniform manner.

Modeling Conventions and Restrictions

The Transformation Engine requires that models to be transformed conform to the modeling rules and conventions described in "MDA Modeling Conventions." If you use Rational Rose, also refer to "PathMATE Modeler's Guide: Model Driven Architecture with Rational Rose." Both publications are available from www.pathfindermda.com.

3. Introduction To Template Syntax

An template is a template that is used to produce a textual document with special fields denoting analysis information or control structures. Let's build a view of the notation by starting with the simplest syntactic elements and work up.

All syntactic elements except free text are surrounded by **[** and **]** . These delimiters are used in your templates to surround special directives and substitution constructs.

This document uses a meta-syntax to present the syntactic elements of the template notation (Table 1).

Table 1. Meta-Syntax for Template Notation

Meta-Syntax	Description
<substitute>	delimit specific fields filled in based on your particular template, like <variable name>
[optional]	indicates optional items - 0 or one
{ set }	indicates a set of 0 or more items.
/ A | B | C /	items within slashes separated by | indicate a mutually exclusive set of choices
keyword	special keywords in syntax elements are bolded - they are to appear in your templates exactly as you see them here.

Basic Syntactic Elements

A syntactic element is a conceptual piece of an template. Everything in an template is a syntactic element of one type or another.

Free Text

Free text is everything in an template that is not explicitly enclosed in **[** and **]** delimiters. This text is copied directly into the target document. A \ (backslash) escapes special meaning for the following character. It is used to allow the "[" and "]" characters to appear in free text and pass through unparsed: "\[" and "\]". To make a "\" appear in the target document, use two in free text: "\\".

Whitespace in Free Text

In order to balance the formatting and readability needs of both the templates and resulting target documents, whitespace appearing outside delimited syntactic elements is either used to help format and clarify the template itself, or is intended for the contents of the target document(s). To support both roles for whitespace, we must evaluate whitespace-only (composed of spaces or tabs, or newlines) free text elements in the template context in which they are encountered.

In all models, an entire template line is considered as a whole. A single free text element cannot span line breaks. The algorithm is:

- If a line has nothing but whitespace free text, they are emitted into target documents
- If a line has nothing but whitespace free text and non-emitting syntactic elements, these free text elements are not emitted into target documents
- If any free text element of a line contains more than just whitespace, then all free text elements on the line are considered intended for the target document.
- If any syntactic element of a line is classified as "emitting", then all free text elements on the line are considered intended for the target document.

"Emitting" syntactic elements are: DATE, EXPAND, EXPORT, AND FILTER (see Template-Level Elements on page 15 for more information on these syntactic elements).

For these purposes, the beginning and end of the template file are considered non-emitting syntactic elements

Expressions

There are three primitives used to present analysis information or other values (including constant information) in a template: a variable, a field, and a constant. Any single instance of these primitives or sets of them combined with operators are considered expressions. All valid expressions resolve to a single value of at any one point in time. All valid expressions result in a value of a specific type.

A limited amount of mixed-type expressions are allowed, as specified below in the specific base data type sections.

Base Data Types

Boolean

The type indication **Boolean** indicates a variable that can equal TRUE of FALSE. A Boolean constant is specified as **TRUE** or **FALSE**.

unary operator:	"**!**"	logical NOT
binary operators:	"**&&**" and "**||**"	logical AND and OR
	"**==**", "**!=**", "**<**", "**<=**", "**>**" and "**>=**"	comparison

Operands for binary comparison operators may be of any type - see corresponding sections below for how these are applied.

Float

The type indication **Float** indicates a C-language double on the native platform. A floating point constant is specified as two sets of concatenated digits, separated by a **5.6778** or **0.0**.

unary operator:	"**-**"	negate
binary operators:	"**+**", "**-**", "*****", and "**/**"	arithmetic operations

- Integer expressions can be used as one operand in a Float expression
- Integer expressions can be compared to Float expressions

Integer

- The type indication **Integer** indicates C-language int on the native platform. An integer constant is specified as a concatenation of digits as in **1024**, or **2** .

unary operator:	"**-**"	negate
binary operators:	"**+**", "**-**", "*****", and "**/**"	arithmetic operations

- Arithmetic operations combining Integer and Float expressions are Float expressions
- Integer expressions can be compared to Float expressions

String

The type indication **String** indicates a variable length concatenation of ASCII characters. A string constant is specified as any set of characters contained within double quotes - **" "** , such as :

"Hi there"

"conundrum"

"-- HEY! --\n"

To insert a double-quote (") in a String constant, use \" as in: "**this is an embedded double-quote: \" - there - we did it**."

unary operator:	- NONE -	
binary operators:	"**+**"	concatenate

- Order-based comparisons use ASCII order
- Strings can be compared to the built-in constant **EMPTY_STRING** with "**==**" and "**!=**"

StringList

To support the manipulation of lists of strings, a variable of type StringList can be constructed. This is done specifically to support the SPLIT function (see String Parsing on page 20). A variable of this type can be in the same manner as a list of analysis elements, with a **FOREACH** element.

Variables

A variable contains and preserves (until overwritten) a piece of analysis information. It can be a handle to other analysis elements or an atomic value (integer, string, etc.). A variable is created with a signature, via an assignment, or as the index variable of a list iteration construct.

Variables are assigned their type explicitly in a signature directive. For the assignment and list iteration, the type of the analysis element assigned to the variable implicitly determines its type. The variable cannot be used before it is assigned in some way.

The scope of a variable is within a single template, however any variables used as actual parameters in a template invocation (expand) are passed by reference to the invoked templates to facilitate outputs. The scope of a variable declared anywhere in an template - even in an iteration variable can be used after its ENDFOREACH.

The name of a variable must start with a letter, and can be made up of any combinations of letters, digits, and underscore characters. A variable name cannot be the same as any of the keywords of this template notation:

ARCHETYPE
TEMPLATE
ASSIGN
BREAK
DATE
DIAGRAM
ELSE
ENDFOREACH
ENDIF
ENDWHILE
EXPAND
FILTER
FOREACH
IF
LOOKUP
OUTFILE
PROPERTY
SPLIT
SPRINGBOARDVERSION
STRTOID
STRTOINT
WHILE

Also, a variable name cannot be one of the type names available, including the base types:

String
Integer
Boolean
Float
AnalysisElementType

or one of the analysis element types defined in the "analysis element type" column of the analysis element field table in Analysis Elements on page 9 (these range from "**System**" to "**Parameter**").

Fields

A field is a reference to a specific sub-element of analysis information contained in a base analysis element. The fields available for each type of analysis element are defined in Appendix A. The general syntax for a field reference is:

<field base>**.**<field name>

where <field base> is a variable or field that is a handle to an analysis element, and <field name> comes from the "field name" column of Analysis Elements on page 9.

Analysis Elements

In addition to the base data types, syntactic elements can be of a type derived from the analysis elements.

Enumeration of Analysis Element Types

Corresponding to each leaf type of analysis element is an enumerate value. These values range from "**System**" to "**Parameter** ". When comparing or assigning to syntactic elements of this type, use the actual analysis element type name as indicated in Enumeration of Analysis Element Types on page 9. Fields of this type are referred to as an "Analysis Element Type."

Handle to Analysis Element

One instance of an analysis element can have a "handle" to another instance - this is the equivalent of a C-language pointer. When comparing against or assigning to syntactic elements of this type, use the integer constant 0 to indicate an empty handle, or use a variable containing an analysis element.

Variables of this type can be used as the base of a field specification, where the variable name is followed by a dot separated list of field names such as "**obj.name**", "**this_state.initiatingEvent.name**", and "**obj.relationships**".

List of Handles to Analysis Elements

Some analysis element fields contain a list of handles to other analysis elements. Use a list iteration control structure – described in List Iteration on page 19 – to traverse these lists, or get their counts. The name of a list type is "<analysis element type name>**List**".

The number of members in any list can be accessed as an integer expression, simply by using the variable or field that corresponds to the list. For example, to see if a domain is analyzed, use the following expression:

"**[IF (domain.objects > 0)]**"

Analysis Elements

An *analysis element* is a component of your analysis models, and is of an *analysis element type*. Each *analysis element type* has a set of information associated with it, arranged in *fields.* Each field is of a data type: see Base Data Types on page 6 above for a detailed description of each data type.

An *analysis element* appears in your template as a *variable*, or a *variable* appended with a set of field names (dot separated). *Variables* are discussed in more detail below.

The textual representation of an *analysis element* is expanded in your target document by placing it within the syntactic element delimiters in an export construct:

[<analysis element> **]**

Please note that fields and variables that are handles to analysis elements or list of handles to analysis elements cannot be exported.

Some *analysis element types* are supertypes of other *analysis element types*. This is indicated by an "is either" reference in the *analysis element type*'s description.

Analysis Element Linkage Table

This is a "top-down" hierarchical view of how the analysis elements relate to each other. Please refer to Appendix A for a complete description of all analysis elements and their fields.

System-Level Elements

```
DataTypeScope (isA System, Domain)
          userDefinedTypes (UserDefinedTypeList)
System
          diagrams(DiagramList)
          domains (DomainList)
Domain
          clients (BridgeList)
          diagrams(DiagramList)
          objects (ObjectList)
          relationships (RelationshipList)
          servers (BridgeList)
          services (DomainServiceList)
          supportDiags (StringList)
          subsystems(SubsystemList)
          userDefinedTypes (UserDefinedTypeList)
Bridge
          clientDomain (Domain)
          serverDomain (Domain)
Subsystem
          diagrams(DiagramList)
          domain(Domain)
          objects(ObjectList)
          parent(Subsystem)
          services(DomainServiceList)
          subsystems(SubsystemList)
Diagram
          scope(AnalysisElement)
AnalysisElement
          stereotype(Stereotype)
          properties(PropertyList)
Stereotype
          extendedElements(AnalysisElementList)
Property
```

Class-Level Elements

```
Object
            accessors (ObjectAccessorList)
            allAttributes (AttributeList)
            associatedRelationship (BinaryRel)
            attributes (AttributeList)
            defaultInitialState(State)
            diagrams(DiagramList)
            domain (Domain)
            events (EventList)
            participants (ParticipantList)
            receivedEvents (EventList)
            services (ObjectServiceList)
            sortStatements (AttributeSortList)
            states (StateList)
            subsystem(Subsystem)
            subTypes (ObjectList)
            superTypes (ObjectList)
            superTypeInRels(SubSuperRelList)
Attribute
            accessors (AttributeSelectionList)
AttributeOrdering
            statement (AttributeSort)
Participant
            relationship (BinaryRel)
            relative (Participant)
Relationship (isA  BinaryRel, SubSuperRel)
BinaryRel
            accessors (RelationshipAccessorList)
            associativeObject (Object)
            participant1 (Participant)
            participant2 (Participant)
SubSuperRel
            subTypes (ObjectList)
            superType (Object)
            subTypeNavigations         (SubSuperNavigationList)
SubSuperNavigation
            destination (Object)
            sourceInstance (Expression)
            subSuperRel(SubSuperRel)
Event
            accessors (EventAccessorList)
            destination (Object)
            parameters (ParameterList)
State
            defaultInitialState(State)
            entryAction(Action)
            exitAction(Action)
            incomingTransitions(NewStateTransitionList)
            initiatingEvents (EventList)
            nestedStates(StateList)
            nextStates (StringList)
            object (Object)
            outgoingTransitions(TransitionList)
            parent(State)
Transition (isA  NewStateTransition, NonStateTransition)
            initiatingEvent(Event)
            source(State)
NewStateTransition
            action(Action)
            destination(State)
            guard(Action)
```

Services

```
Service (isA  DomainService, ObjectService)
            action (Action)
            invokers (ServiceInvocationList)
            parameters (ParameterList)
DomainService
            domain (Domain)
            invokers (ServiceInvocationList)
            parameters (ParameterList)
ObjectService
            invokers (ServiceInvocationList)
            object (Object)
            parameters (ParameterList)
            polymorphism(PolymorphismType)
Action
            actionBlocks (StatementBlockList)
            expressions (ExpressionList)
            statements (StatementList)
            variables (VariableDefinitionsList)
StatementBlock
            statements (StatementList)
```

Action Statements

```
Statement (isA  Assignment, AttributeSort, Break, Continue, CreateServiceHandle,
ForEach, GroupSort, If, Invocation, InvokeServiceHandle, Return, WhileLoop)
Assignment
            lvalue (Expression)
            rvalue (Expression)
AttributeSort
            attributeOrderings (AttributeOrderingList)
            navigation (Expression)
            object (Object)
CreateServiceHandle
            lvalue (Expression)
            parameters (NameValuePairList)
            service (Service)
ForEach
            index (Expression)
            loopBlock (StatementBlock)
            navigation (Expression)
            object (Object)
            whereClause (Expression)
GroupSort
            subject (Expression)
If
            condition (Expression)
            elseBlock (ActionBlock)
            ifBlock (ActionBlock)
Invocation
            invokee (Expression)
InvokeServiceHandle
            handle (Expression)
            parameters (NameValuePairList)
Return
            returnValue (Expression)
WhileLoop
            condition (Expression)
            loopBlock (ActionBlock)
```

Action Expressions

```
Expression (isA  AttributeSelection, BinaryExpression, Constant, EventAccessor,
LocalVariable, MethodInvocation, ObjectAccessor, ParameterVariable,
RelationshipAccessor, ServiceInvocation, UnaryExpression)
AttributeSelection
        instance (Expression)
BinaryExpression
        operand1 (Expression)
        operand2 (Expression)
EventAccessor (isA  Cancel, Generate, ReadTime)
        destination (Expression)
        event (Event)
Generate
        arguments (ActualParameterList)
        delay (Expression)
LocalVariable
        declaration (VariableDefinition)
VariableDefinition
        initialValue (Expression)
MethodInvocation
        arguments (ActualParameterList)
        subject (Expression)
ObjectAccessor (isA  Create, Delete, Find)
        object (Object)
Create
        initialState (State)
        parameters (NameValuePairList)
Delete
        instance (Expression)
Find
        navigation (Expression)
        whereClause (Expression)
RelationshipAccessor (isA  Navigation, Link, Unlink)
        relationship (BinaryRel)
Link
        assocInstance (Expression)
        instance1 (Expression)
        instance2 (Expression)
Navigation
        destParticipant (Participant)
        sourceInstance1 (Expression)
        sourceInstance2 (Expression)
Unlink
        instance1 (Expression)
        instance2 (Expression)
ServiceInvocation
        arguments (ActualParameterList)
        service (Service)
        subject (Expression)
UnaryExpression
        operand1 (Expression)
ActualParameter
        value (Expression)
```

List Sort Ordering

Fields that are lists of certain types of analysis element are sorted by name:

Attribute
Domain
Event
Object
Relationship
Service
State

Lists of analysis element types not listed in this section are kept in the order they appear in the analysis, or are not sorted.

Template-Level Elements

Signature

The signature directive gives the template its name and argument list - its profile. Exactly one must appear in every template, as the first non-comment syntactic element.

[ARCHETYPE <template name> **(** {<inputs>}**;** {<outputs>} **)]**

where

<inputs> are a comma-separated list of: <data type> <variable name>.

<outputs> are a comma-separated list of: <data type> <variable name>.

<data type> is a type defined in Base Data Types on page 6.

<variable name> is the name of the template variable used as the formal parameter.

Input parameters are passed by value, and outputs are passed by reference. When invoking an template, variables must be used as actual parameters that correspond to output parameters, or an error will result.

Top-level templates (invoked directly from the command line) can have no outputs.

Variable

The variable declaration creates a variable of a specific type within the scope of the template (even if it is nested within a loop or if/else). It sets the initial value of the variable.

[VARIABLE <variable type> <variable name> **=** <expression> **]**

You can declare a list variable like any other. See List Construction on page 21 for information about how to populate a list. Do not specify an initial value:

[VARIABLE <variable type>**List** <list variable name> **]**

Assignment

The assignment creates a variable within the scope of the template (even if it is nested within a loop or if/else). It also casts a type on the variable - based on the analysis element used as the right hand value.

If a variable is reused, the old value is overwritten. If an assignment of the wrong type right hand value is made to an existing variable, a production-time error results.

[ASSIGN <variable name> **=** <expression> **]**

Invocation

The any template can be invoked from within the current template. Expressions used as actual parameters must match the formal parameters specified in the invokee's signature. Input parameters are passed by value, and outputs are passed by reference. When invoking a template, variables must be used as actual parameters that correspond to output parameters, or an error will result.

Invocations do not affect target document context - until a new file indicator is encountered in the new template, all output from the invokee goes into the currently indicated file.

[EXPAND <template name> **(** {<inputs>}**;** {<outputs>} **)]**

where

<inputs> are a comma-separated list of: <expression>

<outputs> are a comma-separated list of: <variable name>

<variable name> is the name of the template variable used as an output actual parameter

During production, the calling template's variable values are updated as the called template modifies them.

File Indicator

Unless otherwise directed, all output from an template goes to standard output. The file indicator opens the indicated file for write/overwrite access, creating a new one if it is not found. If another file (with a different name) was already open, it is pushed onto a file context stack, and the newly opened file is used.

[OUTFILE <output file name> **]**

where

<output file name> is a string value (see String on page 7)

To close an output file and pop back to the previous file context, the close directive is used. If a close is encountered on the highest context in the stack any further output will go to the standard output stream. If no file is currently open, then a warning will be issued.

[CLOSEOUTFILE]

Filters

A filter is a special type of string expression that executes a script or program with a specified command line. The output of this execution is captured and returned as the value of the string expression. The argument of the filter - the command line string expression - must be constructed explicitly, and include the program name, its full path (if it is not available through the PATH) and all switches and parameters.

[FILTER <command line string expression> **)]**

Date/Time Stamp

This element emits the date and time the current transformation session was started into the target document. This is done to ensure all documents generated from a single session are marked with the same date/time. This directive expands to <day> <month> <date> <hh>:<mm>:<ss> <year>, as in:"Wed Mar 27 15:30:38 1996".

[DATE]

Engine Program Version

This element emits the name and version of the Springboard program used. This can be useful in a configuration management setting to help identify the versions of the tools used. This directive expands to "sprngbrd version <version number> <version qualifier> (<build date>)", such as "sprngbrd version 1.104 Beta (7/9/96)".

[SPRINGBOARD**VERSION]**

Comment Block

Comments are used to explain your template, and are inserted within the delimiters ("[]") of any syntactic element. They do not appear in the resulting target document. They are patterned after C-language style comments.

[… **/*** <any free text for comment including newlines, etc...> ***/** … **]**

A typical use is to help associate related syntax elements, such as **[IF …(foo)…]** and **[ENDIF /* check foo */]**

Delete Analysis Element

To delete an analysis element from the model:

[DELETE <expression>**]**

where expression is the analysis element to be deleted.

Delete performs a shallow delete. For example, if an object is deleted, its attributes and operations are not deleted. Delete may cause dangling references. For example if an object is deleted, any action language CREATE statements that reference that object will have NULL object fields.

Delete may be performed within a FOREACH in the same template. For example, the following template is allowed:

```
[ARCHETYPE prune(System system)]
[FOREACH domain IN system.domains]
      [FOREACH object IN domain.objects]
            [IF (PROPERTY(object, "Generate", "T")
== "F")]
                  [DELETE object]
            [ENDIF]
      [ENDFOREACH]
[ENDFOREACH]

But the following is NOT allowed:

[ARCHETYPE prune(System system)]
 [FOREACH domain IN system.domains]
      [FOREACH object IN domain.objects]
            [IF (PROPERTY(object, "Generate", "T")
== "F")]
                  [EXPAND delete(object)]
            [ENDIF]
      [ENDFOREACH]
[ENDFOREACH]

[ARCHETYPE delete(Object object)]
      [DELETE object]
```

Control Structures

If-Else

Decisions about which portion of a template to expand (or which to invoke) can be made with If-Else blocks. The Boolean expression provided in the IF is evaluated when the statement is encountered (at target document production time). The else-block is optional.

For more information on Boolean expressions, please see Boolean on page 6.

[IF (<boolean expression>**)**]

{ <if-block syntactic elements> }

[ELSE]

{ <else-block syntactic elements> }]

[ENDIF]

Refer to Comment Block on page 18 for details on how to associate control structure components with comments.

While Loop

While loops iterate over template statements in the loop body while the Boolean test condition evaluates to TRUE.

For more information on Boolean expressions, please see Boolean on page 6.

[WHILE (<Boolean test expression>)**]**

{<loop body>}

[ENDWHILE]

List Operations

Some syntactic elements contain lists of analysis information. There are constructs to traverse a list, determine the number of elements in a list, or return a specific item from a list.

List Iteration

Use this to loop through the entire list. An optional separator string can be specified for placement between loop body expansions.

[FOREACH <index-variable> **IN** <list-element> [**SEPARATOR** "<separator string>"] **]**

{ <iteration block syntactic elements> }

[ENDFOREACH]

To exit the iteration before the end of the list, put a **[BREAK]** directive within the iteration block, at the point where you've decided to stop looping.

Specific List Elements

A list expression can be indexed, resulting in the value of a single element of a list. The index of a specific list element can be specified parenthetically to access that single element of a list. This is used in circumstances where an expression of the type of a single element of the list is required. (Please refer to Expressions on page 6 for more information on the use of an expression.) Any Integer expression can be used to specify the index - 1 references the first element. A production-time error results if there is no list element in the specified position.

<syntactic element of a list type>**(**<position>**)**

example: to reference the first attribute of the object **obj**, use:

obj.attributes(1)

String Parsing

To support the extraction of substrings from a string, the SPLIT function is provided:

<StringList variable> = **SPLIT (**<string expression to be parsed>**,** <string of separator characters>**)**

This operator works very much like a series of calls to the standard C-language utility function `strtok()`: the input string is broken up on the occurrence of any of the separator characters.

SPLIT is a StringList expression - it can be used in an ASSIGN or FOREACH syntactic element. If the <string expression to be parsed> is empty, or no tokens are found, EMPTY_STRING is returned.

NOTE - the <string expression to be parsed> is not modified, unlike `strtok()`.

String Conversion

Conversion from strings to analysis element pointers is provided by the STRTOID expression:

<analysis element variable> = **STRTOID(**<string expression to convert>**)**

Conversion from strings in base 10 to integers is provided by the STRTOINT expression:

<integer variable> = **STRTOINT(**<string expression to convert>**)**

Note: *Integers and pointers are converted automatically to strings. If a template assigns an integer or pointer variable to a string, the numeric value is converted to a string and stored in the string variable. Integers are represented in strings using base 10. Pointers are converted to ID(0x<hex pointer value>).*

The STRTOID and STRTOINT functions are helpful for storing integer and pointer values as properties of analysis elements. For example,

```
[ARCHETYPE top(System system)]
[ASSIGN PROPERTY(system, "NumberOfInstances") =
"5"]
[ASSIGN number_of_instances =
STRTOINT(PROPERTY(system, "NumberOfInstances",
"0"))]
[ASSIGN counter = 1]
[WHILE (counter <= number_of_instances)]
      instance [counter]
[ENDWHILE]
```

List Population Count

A list variable or field can be used as an Integer expression to determine how many items are in the list. The following example shows the use of the field `subtypes` of the Object `obj` in an IF construct:

[IF (**obj.subtypes** > 0)]

[/* do something with the subtypes */]

…

[ELSE]

    [/* there are no subtypes */]

…

[ENDIF]

List Construction

The Insert directive allows elements to be inserted into a list variable:

[INSERT <where> <allow_duplicates> <element> **INTO** <list_var_name>**]**

<*where*>: indicates where to add the new element. Default is BACK. The following locations are supported:

FRONT - inserts <*element*> in the first position in the list

BACK - inserts <*element*> in the last position in the list

SORTED_UP - inserts <*element*> so it is in ascending sorted order – assumes <*list_var_name*> is already sorted

SORTED_DOWN - inserts <*element*> so it is in descending sorted order - assumes <list_var_name> is already sorted

<*allow_duplicates*>: specify the keyword UNIQUE if the element should only be added if the element value is not already in the list. If UNIQUE is not specified, the element will be added to the list in the location specified even if it is already in the list.

<*list_var_name*>: name of the list for insertion

Remove deletes the front or back element from the specified list variable:

[**REMOVE** {**FRONT** | **BACK**} **FROM** <list_var_name>]

Remove deletes the first matching element, the last matching element, or all matching elements from the specified list variable:

[**REMOVE** {**FIRST** | **LAST** | **ALL**} <element> **FROM** <list_var_name>]

examples:

```
[/* insert non-unique to the back of the list*/]
[VARIABLE StringList   include_files]
[INSERT "stdio.h" INTO include_files]
[INSERT "ctype.h" INTO include_files]
[/* list is "stdio.h", "ctype.h" */]

[/* insert in ascending sorted order */]
[VARIABLE StringList   sorted_strings]
[INSERT SORTED_UP "pear" INTO sorted_strings]
[INSERT SORTED_UP "apple" INTO sorted_strings]
[/* apple is not added again since we are adding
uniquely */]
[INSERT SORTED_UP UNIQUE "apple" INTO
sorted_strings]
[INSERT SORTED_UP "pear" INTO sorted_strings]
[/* list is "apple", "pear", "pear" */]

[/* remove all instances of "pear" */]
[REMOVE ALL "pear" FROM sorted_strings]
[/* list is "apple */]

[REMOVE FRONT FROM include_files]
[/* list is "ctype.h" */]
```

Note: Lists may be passed as input or output parameters to archetype invocations. A local copy of the list is made when removing or inserting into an input archetype parameter. Any changes made to an input list parameter are not reflected in the calling context. Insertions and removals from an output parameter are passed back to the calling context.

Lists derived from fields cannot be modified by INSERT and REMOVE. The archetype interpreter makes a copy of all lists when they are modified:

[VARIABLE   object_list = system.domains(1).objects]

[/* INSERT makes a copy of the list returned from the field Domain.object */]

[INSERT object_list(1) INTO object_list]

Analysis element iteration

To iterate over all the instances of an analysis element type in the model, use the ALL expression.

ALL(<element type name>**)**

For example, to iterate over all the objects in all domains, use the ALL expression as follows:

[FOREACH object IN ALL(Object)]

[ENDFOREACH]

Note: The <element type name> must be the name of a type and not a variable of type analysis element type.

Analysis Element lookup

To lookup an analysis element by its qualified name, use the LOOKUP expression.

<analysis element variable> = LOOKUP(<element name>, <element type name>)

For example, to find the domain with prefix SW in the system Robochef, do the following:

[ASSIGN robochef_domain = LOOKUP("Robochef.FP", Domain)

NOTE: The <element type name> must be the name of a type and not a variable of type analysis element type.

In-Line Diagrams

If one of the formatted document command line options is specified (see Command Line Syntax on page 25), then UML diagram references can be placed into the target document with:

[DIAGRAM <diagram name>**]**

This syntactic element takes the name of a diagram - use a field marked "use with DIAGRAM" from Analysis Elements on page 9.

Coloring

Analysis augmentation or design information can be attached to analysis elements as coloring information, and be accessed for any analysis element via the **PROPERTY** construct. Support for this feature is MODEL dependent - some analysis element types do not have support for coloring information capture in some tools. Typically analysis elements with description fields also support coloring.

Properties are captured in the MODEL environment as name-value pairs, where the names are project-specific string constants. The property construct is a special type of string expression that returns the value of a named coloring property of a specified analysis element. If no value is specified for the named property, or if no property has the target name, then a production-time warning results and the default value specified is returned.

PROPERTY (<analysis element handle field>, <property name>, <default property value>)

PROPERTY (<analysis element handle field>, <property name>)

Please note: <property name> and <default property value> must be String expressions.

4. Command Line Syntax

Invoke the Transformation Engine through the springboard console application, which you can run from a shortcut, makefile, or DOS shell window using a command line interface. The command line syntax is:

```
springboard <options> <top template name> <top
template parameter values ...>
```

where:

<options>	**is any combination of:**
-Buffer_size <size>	Sets the ODBC buffer size for extract. Must be at least 3500. *Default*: 10000 (System Architect only).
-Config <file>	Specify the name of the action language expression config file. *Default*: <path install>/config/ pfdexp.cfg.
-Dir <dir>	search <dir> for templates. The current working directory is always searched first, and 0 or more directories may be specified, each with its own -dir option.
-Encyclopedia <dir>	explicitly specify the UML database source so an ODBC data source is not needed (System Architect only).
-EXtract_only	only extract MODEL database information and issue extraction errors. Do not parse templates and generate target documents. *Default:* extract, parse, and create target documents.
-Ignore_errors	proceed with extraction even if there were parsing errors; proceed with production even if there were parsing or extraction errors. *Default*: stop after first production error.
-Output <dir>	use <dir> as root for any relative pathnames specified in OUTFILE directives. *Default*: current working directory.
-PArse_only	only parse this template and any it invokes recursively, report any template syntax warnings or errors, and do not generate any code. *Default* : generate code once the templates are parsed.
-Suppress <cat> <msg #>:	suppress reporting of a specific message. *Default*: report all messages.
-Xml <file>	Specify the name of the XML file containing the analysis in XMI format (Rose only).
<top template name>	name of the top-level template
<top template parameter values ...>	values used as arguments to the top-level template

Notes:

- The first template parameter value must specify the target system, or an analysis element within the system.
- Option keywords are model-insensitive

Template File Search

The -dir option operates like a C compiler's -I option. The current working directory is searched first, and then each directory specified with -dir is searched in order. To find the to-level template, the Engine searches each directory: it first tries to find <template name>`.arc`, and if this doesn't exist, then <template name>`.rtf` is tried.

For subsequent EXPANDed templates (not top-level), only `.arc` files are selected, ignoring `.rtf` files.

Formatting Modes: Unformatted/Formatted

The formatting mode is determined by the extension of the file containing the top-level template. For `.arc` files, Unformatted mode is used. For `.rtf` files, Formatted mode is used.

Unformatted mode:

- All templates files have `.arc` extensions
- Plain, ASCII text is expected in all templates
- Only textual output is supported - no DIAGRAM directives are allowed
- All templates files are expected to have `.arc` extensions

Fomatted mode:

- The top level template file has a `.rtf` extension, all other subsequently expended templates files have `.arc` extensions
- The top level template file is expected to be a Rich Text Format file
- The DIAGRAM directive is supported

Note: Be sure to turn off "Smart Quotes" in Microsoft Word with Tools > Options > AutoFormat. In formatted mode, it may be helpful to use a separate Word style for Engine directives (syntactic elements – in "[ ]"). Then you may wish to choose to make the style invisible to help in whitespace management, or to aid readability of the template.

Generated documents containing diagrams contain only a reference to the diagram image file (.wmf). This means that the original model support files area must be available for the documents to appear. To make a document "transportable" off the network they were generated on, use MS Word to convert these references to insertions: in Word go to Edit->Links, select all Source Files, click "Save Picture in document", and then hit "Break Link". This will pull the contents of the inserted diagrams' .wmf files into the Word document. It is also recommend that at this point you do a Save As and make it a Word document (.doc) - it speeds subsequent open times and saves disk space.

Template Parameter Values from the Command Line

The number and type of template parameter values must match the arguments specified in the template's signature. See Base Data Types on page 6 for information on how to construct constants of each base data type. For formal parameter types that correspond to analysis elements, a fully qualified name for the analysis element instance must be provided. The following types of analysis elements may be specified through a fully qualified name on the command line:

System	<system name>
Domain	<system name>.<dom prefix>
DomainService	<system name>.<dom prefix>.<service name>
Subsystem	<system name>.<dom prefix>.<subsystem name>
Stereotype	<stereotype name>
BaseType	<type name>
GroupType	<type name>
GroupIterType	<type name>
UserDefinedType	<scope qualified name>.<enumerate name>
InstanceReferenceType	<system name>.<dom prefix>.<obj prefix>.Ref<<class name>>
Object	<system name>.<dom prefix>.<obj prefix>
ObjectService	<system name>.<dom prefix>.<obj prefix>.<service name>
Parameter	<service qualified name>.<parameter name>
Attribute	<system name>.<dom prefix>.<obj prefix>.<attr name>
Association	<system name>.<dom prefix>.A<rel num>
Participant	<system name>.<dom prefix>.A<rel num>.<object name>[.<role name>]
SubSuperRel	<system name>.<dom prefix>.S<rel num>
State	<system name>.<dom prefix>.<obj prefix>.<state name>
Event	<system name>.<dom prefix>.<obj prefix>.<event label>

Note: Text in [] is optional. For example, if a Participant does not have a role name, the role name and its preceeding period are omitted from the qualified name.

5. Templates and Sample Systems

Complete code and report template examples exist online with your PathMATE product installation. If you installed PathMATE in the default installation directory, you can find the templates and sample models in these directories:

Report templates reside in

C:\pathmate\reports

and have report examples that include simple ASCII text, fully formatted Word documents, HTML, and consistency checking.

Code generation templates for C++ reside in

C:\pathmate\design\cpp\templates

Start with the template *sys_top.arc*. Code generation templates for C and Java are available if you purchased these modules.

The Carsuffle, Robochef, and SimpleOven sample systems reside in

C:\pathmate\samples

The models for these systems are provided in Rational Rose, and serve as examples of a valid MDA systems.

A. Analysis Element Field Reference

The analysis elements correspond convey the information captured in your UML models. Please refer to Base Data Types on page 6 for a description of the data type for each field.

For analysis elements that are subtypes, all available fields are listed including the fields already defined for the supertype.

Action: supertype: *AnalysisElement;* subtypes*:none*; *a procedure or operation*
actionBlocks (StatementBlockList) *: the blocks of PAL statements that make up this action; the first block is the "root block" containing the entire action; the rest are in sequential order*
actionText (String) *: the actual PAL statements for this action in raw textual form.*
expressions (ExpressionList): *the list of all expressions in this action (in all blocks)*
scope(AnalysisElement) : *The* Domain *this action is in (or its* System, *in the case of a system init).*
statements (StatementList) *: the complete list of all statements for this action, ignoring block boundaries*
type (AnalysisElementType) *: always is* Action
variables (VariableDefinitionsList): *the definitions of all local variables defined in this action*

ActualParameter: supertype: *AnalysisElement;* subtypes*: none; an actual parameter that uses position-association*
value (Expression) *: the actual parameter value*

AnalysisElement: supertype: *none*; subtypes:; an element of the model
type(AnalysisElementType): *type of the analysis element*
id(String): *unique identifier valid for this session only*
properties(PropertyList): *list of name-value pairs associated with this model element*
diagrams(DiagramList): *list of diagrams associated with this model element*
qualifiedName(String): *fully qualified name of the model element*
stereotype(Stereotype): *extension applied to this model element. May be 0 if not extended.*
uuid(String): unique identifier assigned to model element by the modeling tool. Does not change from session to session.

Assignment: supertype: Statement*;* subtypes*: none; the assignment of a new value to an attribute or variable in an* Action
block (ActionBlock) *: the* ActionBlock *this is contained in*
lineNumber (Integer) *: relative to the top of the entire action (top line == 1)*
lvalue (Expression) *: the assignee*
rvalue (Expression) *: the new value for the assignee*
type (AnalysisElementType) *: always is* Assignment

Attribute: supertype: AnalysisElement*;* subtypes*: none*
accessors (AttributeSelectionList) *: all PAL expressions that access this attribute*
dataType (DataType)
description (String)
langId (String) *:* name *sanitized for use as a C-language identifier*
object (Object)
name (String)

AttributeOrdering: supertype: AnalysisElement*;* subtypes*: none; the use of an attribute in an* AttributeSort
ascending (Boolean) *: indicates the sort direction*
attribute (Attribute)
statement (AttributeSort)

AttributeSelection: supertype: Expression*;* subtypes*: none; an expression that reads or writes an attribute value*
action (Action): *the action this is in*
attribute (Attribute)
dataType (DataType) *: matches the attribute's* dataType
instance (Expression) *: the object instance this is an attribute of (== 0 in a Where clause)*
isWrite (Boolean) *: indicates if this is reads or writes the attribute value*
type (AnalysisElementType*) : always is:* AttributeSelection

AttributeSort: supertype: Statement*;* subtypes*: none; reorders a group of instance references based on attribute values*
attributeOrderings (AttributeOrderingList) *: the list of* AttributeOrderings *that control this sort*
block (ActionBlock) *: the* ActionBlock *this is contained in*
lineNumber (Integer) *: relative to the top of the entire action (top line == 1)*
navigation (Expression) *: == 0 unless sorting nav list*
object (Object) *: the Object these are attributes of*
type (AnalysisElementType*) : always is:* AttributeSort

BaseType: supertype: DataType*;* subtypes*: none; A built-in, predefined type*
basicType (Integer)*: 0 – pointer, 1 – Integer, 2 – Real, 3 – String, 4 – Character, 5 - DataContainer*
langId (String) *:* name *sanitized for use as a C-language identifier*
name (String)
type (AnalysisElementType) *: always is* BaseType

BinaryExpression: supertype: Expression*;* subtypes*: none; an expression that has two operands and one operator*
action (Action): *the action this is in*
dataType (DataType) *: the type of this expression*
operator (Integer) *: please refer to the Operator Table at the end of this section*
operand1 (Expression) *: the left side operator*
operand2 (Expression) *: the right side operator*
type (AnalysisElementType*) : always is:* BinaryExpression

BinaryRel: supertype: Relationship*;* subtypes*:none*
accessors (RelationshipAccessorList) *: all PAL expressions that invoke accessors of this relationship*
associativeObject (Object) *: == 0 for non-associative relationships*
description (String)
name (String) *: includes "R<number>:<meaning>"*
number (Integer)
participant1 (Participant) *: one end of the relationship*
participant2 (Participant) *: the other end of the relationship*
type (AnalysisElementType) *: always is* BinaryRel

Break: supertype: Statement*;* subtypes*: none; interrupts execution of this loop iteration, and finishes the loop*
block (ActionBlock) *: the* ActionBlock *this is contained in*
lineNumber (Integer) *: relative to the top of the entire action (top line == 1)*
type (AnalysisElementType) *: always is* Foo

Bridge: supertype: AnalysisElement*;* subtypes*: none; A requirement flow line connecting two* Domains *on the domain chart.*
clientDomain (Domain)*: The domain imposing requirements through the bridge*
description (String)
serverDomain (Domain)*: The domain satisfying requirements through the bridge*

Cancel: supertype: EventAccessor*;* subtypes*:* none; *an invocation of an event cancel accessor*
action (Action): *the action this is in*
dataType (DataType) *: == 0*
destination (Expression) *: a reference to the destination object instance (== 0 for create events)*
event (Event) *: the event this accesses*
type (AnalysisElementType*) : always is* Cancel

Constant: supertype: Expression*;* subtypes*: none; an expression that has a fixed value*
action (Action): *the action this is in*
dataType (DataType) *:*
type (AnalysisElementType*) : always is:* Constant
value (String)

Continue: supertype: Statement*;* subtypes*: none; interrupts execution of this loop iteration, and starts on the next iteration*
block (ActionBlock) *: the* ActionBlock *this is contained in*
lineNumber (Integer) *: relative to the top of the entire action (top line == 1)*
type (AnalysisElementType) *: always is* Continue

Create: supertype: ObjectAccessor*;* subtypes*: none*; *an invocation of an object create accessor*
action (Action): *the action this is in*
dataType (DataType) *: a reference to this object*
initialState (State) *: required when creating an active object*
object (Object) *: the object is accesses*
parameters (NameValuePairList)
type (AnalysisElementType*) : always is* Create

CreateServiceHandle: supertype: Statement*;* subtypes*: none; an invocation of this built-in service*
block (ActionBlock) *: the* ActionBlock *this is contained in*
lineNumber (Integer) *: relative to the top of the entire action (top line == 1)*
lvalue (Expression) *: the assignee*
parameters (NameValuePairList)
service (Service) *: service to be invoked later*
type (AnalysisElementType*) : always is:* CreateServiceHandle

DataType: supertype: AnalysisElement*;* subtypes*:* BaseType, GroupType, GroupIterType, InstanceReferenceType, ServiceHandle, UserDefinedType; The data type for an atomic data item
langId (String) *:* name *sanitized for use as a C-language identifier*
name (String)
type (AnalysisElementType) *: can only be one of:* BaseType, GroupType, GroupIterType,
InstanceReferenceType, ServiceHandle, UserDefinedType

DataTypeScope*: supertype:* AnalysisElement; *subtypes*: System, Domain*,*
initializationHook (Action) : *The init action for this scope.*
type (AnalysisElementType) *: can only be one of:* System, Domain
userDefinedTypes (UserDefinedTypeList): *user defined types defined for this scope*

Delete: supertype: ObjectAccessor*;* subtypes*: none*; *an invocation of an object delete accessor*
action (Action): *the action this is in*
dataType (DataType) *: == 0*
instance (Expression) *: the instance being deleted*
object (Object) *: the object it accesses*
type (AnalysisElementType*) : always is* Delete

Diagram: supertype: *none*; subtypes: *none*
diagramType(String): *The type of the diagram. One of "Class Diagram", "Sequence Diagram", "Use Case", or "Collaboration Diagram"*

filename(String): *the fully qualified filename for use with the DIAGRAM directive*
name(String): *The name of the diagram in the host editor.*
scope(AnalysisElement): *The object, domain, or subsystem to which this domain pertains.*

Domain: supertype: DataTypeScope*;* subtypes*: none*
analyzed (Boolean): *indicates if the domain is realized, or if it has analysis*
clients (BridgeList) *: list of bridges from this domain's clients*
description (String)
diagrams(DiagramList) : *List of all the diagrams for this domain. Includes IM and support diagrams.*
im (String): *the fully qualified filename for diagram graphics for the domain's information model - for use with the DIAGRAM directive*
langId (String) *:* name *sanitized for use as a C-language identifier*
name (String)
objects (ObjectList)
prefix (String)
relationships (RelationshipList)
servers (BridgeList) *: list of bridges from this domain's servers*
services (DomainServiceList)
subsystems(SubsystemList): *list of subsystems partitioning this domain. List contains only immediate subsystems*.
supportDiags (StringList): *a list of fully qualified filenames for diagram graphics for all diagrams with names starting with "<system.name>>." - for use with the DIAGRAM directive*
system (System) : *The enclosing* System.
type (AnalysisElementType*) : always is* Domain
userDefinedTypes (UserDefinedTypeList): *user defined types defined for this scope*

DomainService: supertype: Service*;* subtypes*: none*
action (Action) : *The procedure to be performed upon invocation*
dataType (DataType)*: the return value data type; == 0 for services with no return value*
description (String): *The analyst-entered service description.*
domain (Domain)
invokers (ServiceInvocationList) *: all PAL expressions that invoke this service*
langId (String) *:* name *sanitized for use as a C-language identifier*
name (String)
parameters (ParameterList)
subsystem(Subsystem)
type (AnalysisElementType) *: always is:* DomainService

Event: supertype: AnalysisElement*;* subtypes*: none*
accessors (EventAccessorList) *: all PAL expressions that invoke accessors of this event*
description (String)
destination (Object) *: the* Object *with the matching prefix*
isCreate (Boolean) *: indicates if this causes a transition into a create state*
langId (String) *:* name *sanitized for use as a C-language identifier*
name (String) *: excludes destination prefix and colon*
parameters (ParameterList)

EventAccessor: supertype: Expression*;* subtypes*:* Cancel, Generate, ReadTime; *an invocation of an event accessor*
action (Action): *the action this is in*
dataType (DataType) *: the data type of this accessor's value (== 0 for accessor invocation with no return value)*
destination (Expression) *: a reference to the destination object instance (== 0 for create events)*
event (Event) *: the event this accesses*
type (AnalysisElementType*) : can only be one of:* Cancel, Generate, ReadTime

Expression: supertype: AnalysisElement*;* subtypes*:* AttributeSelection, BinaryExpression, Constant, EventAccessor, LocalVariable, ObjectAccessor, ParameterVariable, RelationshipAccessor, ServiceInvocation, UnaryExpression; *an invocation of a function and/or something that returns/has a value (as an rvalue), or something that can have it's value set (as an lvalue)*
action (Action): *the action this is in*
dataType (DataType) *: the data type of this expression's value (== 0 for function invocation with no return value)*
type (AnalysisElementType*) : indicates which subtype it is*

Find: supertype: ObjectAccessor*;* subtypes*: none*; *an invocation of an object find accessor*
action (Action): *the action this is in*
dataType (DataType) *: == 0*
findType (Integer) *: FIRST, LAST*
navigation (Expression): *optional relationship navigation clause to start Find from*
object (Object) *: the object this Find accesses*
type (AnalysisElementType*) : always is* Find
whereClause (Expression) *: selection criteria; may == 0*

ForEach: supertype: Statement*;* subtypes*: none*; *an iterative traversal of a set of object instances*
block (ActionBlock) *: the* ActionBlock *this is contained in*
index (Expression) *: the local variable or parameter used as the cursor*
lineNumber (Integer) *: relative to the top of the entire action (top line == 1)*
loopBlock (StatementBlock) *: the loop body*
navigation (Expression): *optional relationship navigation clause to start Find from*
object (Object) *: the object this Find accesses*
type (AnalysisElementType*) : always is* ForEach
whereClause (Expression) *: selection criteria; may == 0*

Generate: supertype: EventAccessor*;* subtypes*:* none; *an invocation of an event generate accessor*
action (Action): *the action this is in*
arguments (ActualParameterList)
dataType (DataType) *: == 0*
delay (Expression) *: the amount of time to wait before queueing the event (== 0 for immediate events (non-delayed); units are design-specific)*
destination (Expression) *: a reference to the destination object instance (== 0 for create events)*
event (Event) *: the event this accesses*
type (AnalysisElementType*) : always is* Generate

GroupIterType: supertype: DataType*;* subtypes*: none; Defines an iterator for a set container*
base (DataType)*: The type this maps to*
langId (String) *:* name *sanitized for use as a C-language identifier*
name (String)
type (AnalysisElementType) *: always is* GroupIterType

GroupSort: supertype: Statement*;* subtypes*: none; reorders a group of atomic data values*
ascending (Boolean) *: indicates the sort direction*
block (ActionBlock) *: the* ActionBlock *this is contained in*
lineNumber (Integer) *: relative to the top of the entire action (top line == 1)*
subject (Expression) *: the group this will reorder*
type (AnalysisElementType*) : always is:* GroupSort

GroupType: supertype: DataType*;* subtypes*: none; Defines a set container*
base (DataType)*: The type this maps to*
langId (String) *:* name *sanitized for use as a C-language identifier*
name (String)
type (AnalysisElementType) *: always is* GroupType

If: supertype: Statement*; subtypes: none; a sequential conditional logic construct*
block (ActionBlock) *: the* ActionBlock *this is contained in*
condition (Expression) *: the controlling condition*
elseBlock (ActionBlock) *: the block of statements to execute if the condition is FALSE*
ifBlock (ActionBlock) *: the block of statements to execute if the condition is TRUE*
lineNumber (Integer) *: relative to the top of the entire action (top line == 1)*
type (AnalysisElementType) *: always is* If

InstanceReferenceType: supertype: DataType*;* subtypes*: none; a handle to an object instance*
object (Object)*: the object this refers to*
langId (String) *:* name *sanitized for use as a C-language identifier*
name (String)
type (AnalysisElementType) *: always is* InstanceReferenceType

Invocation: supertype: Statement*;* subtypes*: none; a statement that invokes a accessor, service or method*
block (ActionBlock) *: the* ActionBlock *this is contained in*
invokee (Expression) *: the accessor, service or method this calls*
lineNumber (Integer) *: relative to the top of the entire action (top line == 1)*
type (AnalysisElementType) *: always is* Invocation

Link: supertype: RelationshipAccessor*;* subtypes*: none*; *an invocation of a relationship link accessor*
action (Action): *the action this is in*
assocInstance (Expression) *: a reference to the instance of the associated object (== 0 if no associative object)*
dataType (DataType) *: a single or Group of references to the destination instance*
instance1 (Expression) *: reference to the instance corresponding to the* participant1 *end of the relationship*
instance2 (Expression) *: reference to the instance corresponding to the* participant2 *end of the relationship*
relationship (BinaryRel)
type (AnalysisElementType*) : always is* Link

LocalVariable: supertype: Expression*;* subtypes*: none; a variable constrained in scope to the containing action*
action (Action): *the action this is in*
declaration (VariableDefinition): *the* Statement *that declares this variable*
type (AnalysisElementType*) : always is:* LocalVariable

MethodInvocation: supertype: Expression*;* subtypes*: none; the invocation of a built-in method (not a domain or bridge service)*
action (Action): *the action this is in*
dataType (DataType) *: method return value data type (== 0 if no return value)*
langId (String) *:* name *sanitized for use as a C-language identifier*
name (String) : *name of the method to be invoked*
arguments (ActualParameterList)
subject (Expression) *: identifies the class instance this is a method of (== 0 for static methods)*
type (AnalysisElementType*) : always is:* MethodInvocatrion

NameValuePair: supertype: AnalysisElement*;* subtypes*: none; an actual parameter that uses name-association*
paramName (String) *: the formal parameter name for this argument*
value (Expression) *: the actual parameter value*

Navigation: supertype: RelationshipAccessor*;* subtypes*: none*; *an invocation of a relationship navigation accessor*
action (Action): *the action this is in*
dataType (DataType) *: a single or Group of references to the destination instance*
relationship (BinaryRel)
destParticipant (Participant) *: the destination end of the relationship (indicates the direction of travel)*
sourceInstance1 (Expression) *: the starting point of the navigation*
sourceInstance2 (Expression) *: the OTHER starting point of the navigation (only used for navigation to the associtive class instance in a m:m)*
toAssociatve (Boolean) *: indicates the final destination is to the instance of the relationship's associative object*
type (AnalysisElementType*) : always is* Navigation

NewStateTransition: supertype:Transition*;* subtypes: *none; a transition from one state to another in response to the reception of an event.*
action(Action): the transition action to be performed if this action is taken
destination(State): *the state this state machine transitions to when this event is received, if guard evaluates to TRUE*
guard(Action): the guard action to be performed to see if this transition is to be taken
initiatingEvent(Event): *the event causing this transition*
source(State): *the state this state machine is in when the event is received*
type (AnalysisElementType*) : always is* NewStateTransition

NonStateTransition: supertype:AnalysisElement*;* subtypes:*none; a transition logic response for a state in response to the reception of an event that does not result in a transition to a new state (isDeffered and isIgnored cannot be TRUE at the same time).*
initiatingEvent(Event): *the event causing this transition*
isDeferred(Boolean): TRUE indicates this event is to be requeued for later re-dispatch
isIgnored(Boolean): TRUE indicates this event is to be discarded
source(State): *the state this state machine is in when the event is received*
type (AnalysisElementType*) : always is* NonStateTransition

Object: supertype: AnalysisElement*;* subtypes*: none*
accessors (ObjectAccessorList) *: all PAL expressions that invoke accessors of this object*
allAttributes (AttributeList) *: all attributes defined for this object and all of it's supertypes and their parents*
associatedRelationship (BinaryRel) *: == 0 for non-associative objects*
attributes (AttributeList) *: the attributes defined for this object (excluding supertypes)*
defaultInitialState(State)
description (String)
diagrams(DiagramList): *the list of diagrams for this object including the STD*
domain (Domain)
events (EventList) *: the events with this object's prefix*
langId (String) *:* name *sanitized for use as a C-language identifier*
name (String)
participants (ParticipantList)*: the ends of relationships that this object is in*
prefix (String)
receivedEvents (EventList) *: the events that cause transitions on this object's STD*
services (ObjectServiceList)
sortStatements (AttributeSortList)
states (StateList)
std (String): *the fully qualified filename for diagram graphics for the object's state transition diagram - for use with the DIAGRAM directive*
subsystem(Subsystem)
subTypes (ObjectList) *: my immediate children (not their children)*
superTypes (ObjectList) *: my immediate parents (not their parents)*
superTypeInRels(SubSuperRelList) : *list of relationships where this object is supertype*

ObjectAccessor: supertype: Expression*;* subtypes*:* Create, Delete, Find; *an invocation of an object accessor*
action (Action): *the action this is in*
dataType (DataType) *: the data type of this accessor's value (== 0 for accessor invocation with no return value)*
object (Object) *: the object is accesses*
type (AnalysisElementType*) : can only be one of:* Create, Delete, Find

ObjectService: supertype: Service*;* subtypes*: none*
action (Action) : *The procedure to be performed upon invocation*
dataType (DataType)*: the return value data type; == 0 for services with no return value*
description (String): *The analyst-entered service description.*
instanceBased (Boolean)
invokers (ServiceInvocationList) *: all PAL expressions that invoke this service*
langId (String) *:* name *sanitized for use as a C-language identifier*
name (String)
object (Object)
parameters (ParameterList)
polymorphism(PolymorphismType) : NOT_POLYMORPHIC indicates the service is not polymorphic, POLYMORPHIC_INTERFACE *indicates that this service defines the interface to this service for subtype implementations,* POLYMORPHIC_IMPLEMENTATION *indicates that this service defines the implementations to this service for this subtype*
type (AnalysisElementType) *: always is* ObjectService

Parameter: supertype: *none;* subtypes*: none; a* Service *argument or an* Event *data item*
dataType (DataType)
defaultValue (String)
description (String)
langId (String) *:* name *sanitized for use as a C-language identifier*
mode (Integer) *: 0 - input only; 1 - output only; 2 - input/output*
name (String)

ParameterVariable: supertype: Expression*;* subtypes*: none; the use of a parameter of the containing action*
action (Action): *the action this is in*
dataType (DataType)
parameter (Parameter) *: matches the* Parameter's dataType
type (AnalysisElementType*) : always is:* ParameterVariable

Participant: supertype: AnalysisElement*;* subtypes*: none; The participant is one end of a* BinaryRel*.*
conditional (Boolean)
multiple (Boolean)
name (String) *: the role phrase for this end of the relationship*
object (Object)
relationship (BinaryRel)
relative (Participant) *: the other end of the* BinaryRel*.*

Property: supertype: *none*; subtypes: *none*; *Name value pair assigned to an analysis element.*
keyword(String): *name of the property*
value(String): *value assigned to the property*

ReadTime: supertype: EventAccessor*;* subtypes*:* none; *an invocation of an event read time accessor*
action (Action): *the action this is in*
dataType (DataType) *: assignment-compatible with Real*
destination (Expression) *: a reference to the destination object instance (== 0 for create events)*
event (Event) *: the event this accesses*
type (AnalysisElementType*) : always is* ReadTime

Relationship: supertype: AnalysisElement*;* subtypes*:* BinaryRel, SubSuperRel
description (String)
number (Integer)
type (AnalysisElementType) *: can only be one of:* BinaryRel, SubSuperRel

RelationshipAccessor: supertype: Expression*;* subtypes*:* Navigation, Link, Unlink; *an invocation of a relationship accessor*
action (Action): *the action this is in*
dataType (DataType) *: the data type of this accessor's value (== 0 for accessor invocation with no return value)*
relationship (BinaryRel)
type (AnalysisElementType*) : can only be one of:* Navigation, Link, Unlink

Return: supertype: Statement*;* subtypes*: none*
block (ActionBlock) *: the* ActionBlock *this is contained in*
lineNumber (Integer) *: relative to the top of the entire action (top line == 1)*
returnValue (Expression): *the value being returned by this statement*
type (AnalysisElementType) *: is always* Return

Service: supertype: AnalysisElement*;* subtypes*:* DomainService, ObjectService
action (Action) : *The procedure to be performed upon invocation*
dataType (DataType)*: the return value data type; == 0 for services with no return value*
description (String): *The analyst-entered service description.*
invokers (ServiceInvocationList) *: all PAL expressions that invoke this service*
langId (String) *:* name *sanitized for use as a C-language identifier*
name (String)
parameters (ParameterList)
type (AnalysisElementType) *: can only be one of:* DomainService, ObjectService

ServiceHandle: supertype: DataType*;* subtypes*: none; defines a handle to a service and its parameters that can be specified at runtime.*
langId (String) *:* name *sanitized for use as a C-language identifier*
name (String)
type (AnalysisElementType) *: always is* ServiceHandle

ServiceHandleInvocation: supertype: Statement*;* subtypes*: none; a statement that invokes a ServiceHandle*
block (ActionBlock) *: the* ActionBlock *this is contained in*
lineNumber (Integer) *: relative to the top of the entire action (top line == 1)*
parameters (NameValuePairList)
serviceHandle (Expression) *: the ServiceHandle being called*
type (AnalysisElementType) *: always is* InvokeServiceHandle

ServiceInvocation: supertype: Expression*;* subtypes*: none; the invocation of an object or domain service*
action (Action): *the action this is in*
arguments (ActualParameterList)
dataType (DataType)
service (Service)
subject (Expression) *: identifies the object instance this is a method of (== 0 for non-instance-based methods)*
type (AnalysisElementType*) : always is:* ServiceInvocation

State: supertype:AnalysisElement*;* subtypes*: none*
actionSummary (String) *: high-level English summary of action from STD*
defaultInitialState(State) : *The nested state that is the default initial substate (only valid for a superstate)*
description(String) : *the state's description*
entryAction(Action) : *the action to be executed upon entry to this state*
exitAction(Action) : *the action to be executed upon exit from this state*
incomingTransitions(NewStateTransitionList): *the set of transitions into this state*
initiatingEvents (EventList) *: the list of different events that can cause a transition into this state*
kind(StateKind): *indicates the type of state or psuedostate*
langId (String) *:* name *sanitized for use as a C-language identifier*
name (String): *the name of this state*
nestedStates(StateList): *the list of state enclosed by this superstate*
nextStates (StringList) *: the STT row for this state; there is an enter for each of the* receivedEvents *for this state's* Object (retained for backward compatibility).
object (Object): *the object whose lifecycle contains this state*
outgoingTransitions(TransitionList): *the set of transition logic responses for all events into this state*
parent(State): *this state's superstate (may be NULL)*
type (AnalysisElementType) *: always is* State

Statement: supertype: AnalysisElement; subtypes: Assignment, AttributeSort, Break, Continue, CreateServiceHandle, ForEach, GroupSort, If, Invocation, InvokeServiceHandle, Return, WhileLoop; *a single line of PAL*
block (ActionBlock) *: the* ActionBlock *this is contained in*
lineNumber (Integer) *: relative to the top of the entire action (top line == 1)*
type (AnalysisElementType*) : indicates which subtype it is*

StatementBlock: supertype: AnalysisElement*;* subtypes*: none*; *a block of PAL statements within a process model*
action (Action) *: the* Action *that contains this block*
statements (StatementList) *: the contained in this* ActionBlock

Stereotype: supertype: AnalysisElement; subtypes: *none*; *an extension of the semantics of an analysis element*
name(String) : *the name of the stereotype*
extendedElements(AnalysisElementList): *the analysis elements that use this extension*

SubSuperNavigation: supertype: Expression*;* subtypes*: none; a "cast" from a supertype to one of its subtypes*
action (Action): *the action this is in*
dataType (DataType) *: the type of this expression*
destination (Object) *: the subtype*
sourceInstance (Expression) *: the instance of the supertype*
subSuperRel(SubSuperRel*): the subtype supertype relationship navigated*
type (AnalysisElementType*) : always is:* SubSuperNavigation

SubSuperRel: supertype: Relationship*;* subtypes*:none*
description (String)
number (Integer)
isRealization(Boolean) : *TRUE for realization relationships, FALSE for all other inheritance relationships*
subTypes (ObjectList)
superType (Object)
subTypeNavigations (SubSuperNavigationList) *: navigations down this subtype supertype relationship*
type (AnalysisElementType) *: always is* SubSuperRel

Subsystem: supertype: AnalysisElement; subtypes: *none*
description(String)
diagrams(DiagramList): *list of diagrams for this subsystem. Includes IM and support diagrams.*
domain(Domain): *The parent domain if there is no parent subsystem.*
im (String): *the fully qualified filename for diagram graphics for the domain's information model - for use with the DIAGRAM directive*
langId (String) *: name sanitized for use as a C-language identifier*
name (String)
objects (ObjectList)
parent(Subsystem): *subsystem containing this subsystem*
services(DomainServiceList)
subsystems(SubsystemList): *any nested subsystems*
supportDiags(StringList): *a list of fully qualified filenames for diagram graphics for all diagrams with names starting with "<subsystem.name>>." - for use with the DIAGRAM directive*

System: supertype: DataTypeScope*;* subtypes*: none*
description (String)
diagrams(DiagramList): *a list of diagrams for the system. Includes domain chart and support diagrams.*
domainChart (String): *the fully qualified filename for diagram graphics for the domain chart - for use with the DIAGRAM directive*
domains (DomainList)
initializationHook (Action) : *The init action for this scope.*
langId (String) : name *sanitized for use as a C-language identifier*
name (String)
supportDiags (StringList): *a list of fully qualified filenames for diagram graphics for all diagrams with names starting with "<system.name>>." - for use with the DIAGRAM directive*
type (AnalysisElementType*) : always is* System
userDefinedTypes (UserDefinedTypeList): *user defined types defined for this scope*

Transition: supertype:AnalysisElement*;* subtypes: NewStateTransition, NonStateTransition*; a transition logic response for a state in response to the reception of an event.*
initiatingEvent(Event): *the event causing this transition*
source(State): *the state this state machine is in when the event is received*
type (AnalysisElementType*) : indicates which subtype it is*

UnaryExpression: supertype: Expression*;* subtypes*: none; an expression that has one operand and one operator*
action (Action): *the action this is in*
dataType (DataType) *: the type of this expression*
operator (Integer)
operand1 (Expression)
type (AnalysisElementType*) : always is:* UnaryExpression

Unlink: supertype: RelationshipAccessor*;* subtypes*: none*; *an invocation of a relationship unlink accessor*
action (Action): *the action this is in*
dataType (DataType) *: a single or Group of references to the destination instance*
instance1 (Expression) *: reference to the instance corresponding to the* participant1 *end of the relationship*
instance2 (Expression) *: reference to the instance corresponding to the* participant2 *end of the relationship*
relationship (BinaryRel)
type (AnalysisElementType*) : always is* Unlink

UserDefinedType: supertype: DataType*;* subtypes*:* UserEnumerate, UserNonEnumerate*; A data type defined by the user*
base (DataType)*: The type this maps to*
langId (String) *:* name *sanitized for use as a C-language identifier*
name (String)
scope (DataTypeScope): *the analysis scope where this is defined*
type (AnalysisElementType) *: is either* UserEnumerate *or* UserNonEnumerate

UserEnumerate: supertype: UserDefinedType*;* subtypes*: none; An enumerate defined by the user*
base (DataType)*: The type this maps to (*Integer*)*
langId (String) *:* name *sanitized for use as a C-language identifier*
name (String)
scope (DataTypeScope): *the analysis scope where this is defined*
type (AnalysisElementType) *: always* UserNonEnumerate
valueList (StringList): *the list of enumerate values for this type*

UserNonEnumerate: supertype: UserDefinedType*;* subtypes*: none; A non-enumerate data type defined by the user*
base (DataType)*: The type this maps to*
isExtern (Boolean) : *TRUE indicates this is an EXTERN type (defined outside of analysis).*
langId (String) *:* name *sanitized for use as a C-language identifier*
name (String)
scope (DataTypeScope): *the analysis scope where this is defined*
type (AnalysisElementType) *: always* UserNonEnumerate
VariableDefinition: supertype: Statement*;* subtypes*: none; the declaration of a local variable*
block (ActionBlock) *: the* ActionBlock *this is contained in*
dataType (DataType) *: variable type*
initialValue (Expression) *: an optional initial value for the variable*
isConst (Boolean) : *TRUE indicates this a constant.*
isExtern (Boolean) : *TRUE indicates this is externally declared/initialized*
lineNumber (Integer) *: relative to the top of the entire action (top line == 1)*
name (String) *: variable name*
type (AnalysisElementType) *: always is* VariableDefinition

WhileLoop: supertype: Statement*;* subtypes*: none; an iterative conditional logic construct*
block (ActionBlock) *: the* ActionBlock *this is contained in*
condition (Expression) *: the controlling condition*
lineNumber (Integer) *: relative to the top of the entire action (top line == 1)*
loopBlock (ActionBlock) *: the block of statements to repeatedly execute while the condition is TRUE*
type (AnalysisElementType) *: always is* WhileLoop

B. Operators and Constants

Operators:

number	*operator*	*description*
OP_ADD	+	binary add
OP_BIT_AND	&	bitwise AND
OP_BIT_OR	|	bitwise OR
OP_BIT_XOR	^	bitwise XOR
OP_COMPLEMENT	~	complement
OP_DIVIDE	/	divide
OP_EQ	==	equal
OP_GREATER	>	greater than
OP_GREATER_EQ	>=	greater than or equal
OP_INDEX	[]	subscript
OP_LESS	<	less than
OP_LESS_EQ	<=	less than or equal
OP_LOG_AND	&&	logical AND
OP_LOG_OR	||	logical OR
OP_LSH	<<	shift left
OP_MODULO	%	modulo
OP_MULTIPLY	*	multiply
OP_NEQ	!=	not equal
OP_RSH	>>	shift right
OP_SUBTRACT	-	binary subtract
OP_UMINUS	-	negate
OP_UNOT	!	not
OP_UPLUS	+	unary positiive

Parameter Modes:

MODE_INPUT
MODE_OUTPUT
MODE_INPUT_OUTPUT

Built-in Types:

BASIC_TYPE_BOOLEAN
BASIC_TYPE_CHARACTER
BASIC_TYPE_INTEGER
BASIC_TYPE_REAL
BASIC_TYPE_STRING
BASIC_TYPE_HANDLE
BASIC_TYPE_GENERIC_VALUE
BASIC_TYPE_VOID

Find Types:

FIND_FIRST
FIND_LAST

State Kinds:

INITIAL_STATE
FINAL_STATE
SHALLOW_HISTORY_STATE
DEEP_HISTORY_STATE
REGULAR_STATE

C. Capacities and Limitations

DATA ITEM LIMITATIONS

Engine imposes the following limits on string lengths:

• coloring property value:	4096
• event name:	100
• literal value:	100

All other limits in the number or length of items are only those imposed by your UML editing environment.