

**Response to OMG RFP ad/98-11-01
Action Semantics for the UML
Submitted August 15, 2000**

Submitters:

Concept Five Technologies, Inc.

I-Logix

INRIA

Kennedy-Carter

Motorola

Kabira Technologies, Inc.

Project Technology, Inc.

Rational Software Corporation

Telelogic

T-Vec

Supported by:

Alcatel

Booz, Allen and Hamilton

Rox Software, Inc.

Simware

Software Productivity Consortium.

Contents

Table of Figures	vii
Preface	ix
0.1 Proposal summary	ix
0.2 Resolution of RFP mandatory and optional requirements and issues	ix
0.3 Technical Proposal	xii
0.4 Compliance Issues	xiii
Part I. Summary of Proposal	1
1. Overview	3
1.1 Action Semantics	3
1.2 Principles	3
1.3 Defining Semantics	6
1.4 The Execution Model	8
1.5 The Actions	10
1.6 Frequently Asked Questions	12
2. The Static Model	15
2.1 Chapter Structure	15
2.2 Well-Formedness Rules	16
2.3 Description of a Class	16
2.4 UML Object Constraint Language (OCL)	18
3. The Dynamic Model	21
3.1 Time Ordering and Causality	21
3.2 Execution Rules	22
3.3 Additional Chapter Structure For Actions	22
3.4 Examples	25
CreateObjectAction	25
VariableAction (abstract)	26
Part II. Execution Semantics	29
4. Execution Fundamentals	31
4.1 Instances and Links	31
4.2 Instance and Link Execution Model Classes	35
AttributeValue	35
AssociationExtentIdentity	36
AssociationExtentSnapshot	36
Change	36
ClassifierExtentIdentity	37
ClassifierExtentSnapshot	37
DataValueIdentity	38
ExecutionSnapshot	38
History	39
Identity	39
InstanceIdentity	39
LinkEndValue	39
LinkIdentity	40
LinkObjectIdentity	40
LinkObjectSnapshot	40
LinkSnapshot	41
ObjectIdentity	41

Contents

ObjectSnapshot	41
Snapshot	42
Step	42
5. State Machine Execution	43
5.1 State Machine Execution Model	43
5.2 Run-to-Completion Step	45
5.3 Occurrences	46
5.4 State Machine Execution Classes	47
ChangeOccurrence	47
ObjectSnapshot	48
Occurrence	48
SignalOccurrence	48
StateMachineExecutionIdentity	48
StateMachineExecutionSnapshot	49
TimeOccurrence	49
Part III. Actions	51
6. Action Foundation	53
6.1 Action Specification	53
6.2 Action Execution	55
6.3 Action Foundation Classes	58
Action	58
ControlFlow	59
DataFlow	60
InputPin	60
OutputPin	61
Pin	61
PrimitiveAction	61
Procedure	62
6.4 Action Foundation Execution Model Classes	62
ActionExecutionIdentity	62
ActionExecutionSnapshot	62
PinValue	64
PrimitiveActionExecutionIdentity	65
ProcedureExecutionIdentity	65
ProcedureExecutionSnapshot	65
7. Composite Actions	67
7.1 Composite Action Specification	67
7.2 Composite Action Execution	70
7.3 Composite Action Classes	75
Clause	75
ConditionalAction	76
GroupAction	77
LoopAction	78
Variable	79
7.4 Composite Action Execution Model Classes	80
ClauseExecutionIdentity	80
ClauseExecutionSnapshot	80
LoopExecutionSnapshot	84
VariableExecutionIdentity	86
VariableExecutionSnapshot	86

8. Read and Write Actions	89
8.1 Object Actions	89
8.2 Attribute Actions	90
8.3 Association Actions	91
8.4 Variable Actions	95
8.5 Other Actions	95
8.6 Additional OCL Operations for Read and Write Actions	96
8.7 Read and Write Action Classes	97
AssociationAction (abstract)	97
AttributeAction (abstract)	98
CreateLinkAction	99
CreateLinkObjectAction	100
CreateObjectAction	101
DestroyLinkAction	102
DestroyObjectAction	103
LinkEndCreationData	104
LinkEndData	105
QualifierValue	105
ReadAssociationAction	105
ReadAttributeAction	107
ReadExtentAction	108
ReadIsClassifiedObjectAction	108
ReadLinkObjectEndAction	110
ReadLinkObjectQualifierAction	111
ReadSelfAction	112
ReadVariableAction	112
ReclassifyObjectAction	113
VariableAction (abstract)	114
WriteAttributeAction	115
WriteLinkAction (abstract)	117
WriteVariableAction	118
9. Computation Actions	121
9.1 Computation actions	121
9.2 Computation Classes	122
ApplyFunctionAction	122
CodeAction	123
LiteralValueAction	124
NullAction	124
9.3 Execution Classes	125
ApplyPrimitiveFunctionExecution	125
CodeExecution	125
LiteralValueExecution	125
NullActionExecution	125
10. Collection Actions	127
10.1 General Rules for Collection Actions	127
10.2 Collection Action Classes	128
FilterAction	128
IterateAction	130
MapAction	132
ReduceAction	134
10.3 Execution Model Actions	137
FilterExecution	137
IterateExecution	138

Contents

MapExecution	138
ReduceExecution	139
11. Messaging Actions	141
11.1 Procedures and Calls	141
11.2 Sending Signals	141
11.3 Invitations	142
11.4 Messaging Classes	142
CallAction	142
InviteAction	144
InvitePacket	145
SendAction	145
SendPacket	146
Signal	146
11.5 Execution Model Classes	148
CallActionExecution	148
InviteActionExecution	149
SendActionExecution	150
Part IV. Appendices	151
A Updates to UML	153
A.1 Foundation	153
A.2 Behavioral Elements	156
B Action Language Examples	161
B.1 The Action Languages	161
B.2 Presentation of the Examples	161
B.3 Control Structures	163
B.4 Object Manipulation	165
B.5 Messaging Actions	176
B.6 A Complete Example	179
General Index	185
Index of Classes	187
Index of Features	189
Index of OCL Operations	191

Table of Figures

Figure 1. Identity and snapshot execution model	7
Figure 2. Example history	8
Figure 3. Example model (an action metamodel)	15
Figure 4. An Instance Diagram showing Actions in user model.	16
Figure 5. Multiplicity Model	18
Figure 6. Execution dynamics fundamentals	21
Figure 7. Generic life cycle for action execution	24
Figure 8. Data-value identity	31
Figure 9. Object execution model	32
Figure 10. Link execution model	33
Figure 11. Link object execution model	33
Figure 12. Classifier extent model	34
Figure 13. Association extent model	34
Figure 14. State machine execution model	44
Figure 15. Occurrence Model	47
Figure 16. Action foundation model	53
Figure 17. Action foundation execution model	56
Figure 18. Life cycle for action execution	57
Figure 19. Composite actions metamodel	67
Figure 20. Composite action execution model	70
Figure 21. Life cycle for group-action execution	71
Figure 22. Life cycle for clause execution	72
Figure 23. Life cycle for conditional-action execution	73
Figure 24. The life cycle for loop-action execution	74
Figure 25. Object Action metamodel	89
Figure 26. Attribute action metamodel	90
Figure 27. Link identification metamodel	92
Figure 28. Read-association action metamodel	92
Figure 29. Read link object actions	93
Figure 30. Write link action metamodel	94
Figure 31. Variable Actions	95
Figure 32. Other action metamodel	96
Figure 33. Computation classes metamodel	121
Figure 34. Computation execution classes	122
Figure 35. Collection actions	128
Figure 36. Filter action	129
Figure 37. Iterate action	131
Figure 38. Map action	133
Figure 39. Reduce action	135
Figure 40. Collection action execution	137
Figure 41. Messaging classes metamodel	143
Figure 42. Messaging classes execution metamodel	147
Figure 43. Procedure and Expression	154
Figure 44. Procedure	154
Figure 45. Procedures and Actions	156
Figure 46. Messaging, Computation, Collection, and Composite Actions	157
Figure 47. Read and Write Actions	157
Figure 48. Invitations	158
Figure 49. Ill-formed state machine	159
Figure 50. Signals on entry and exit procedures	159
Figure 51. If-then-else Logic	163

Table of Figures

Figure 52. Multi-way Decision	164
Figure 53. Simple Object Creation	165
Figure 54. Object Creation with Attribute Assignment	166
Figure 55. Object Destruction	167
Figure 56. Writing of Attributes (Single Attribute, Single Object)	167
Figure 57. Writing of Attributes (Multiple Attributes, Single Object)	168
Figure 58. Writing of Attributes (Single Attribute, Multiple Objects)	169
Figure 59. Single Object Selection	170
Figure 60. Obtaining a Collection of Objects	171
Figure 61. Creating a Link	172
Figure 62. Destroying a Link	173
Figure 63. Navigating an Association	174
Figure 64. Obtaining a Collection by Navigation	175
Figure 65. Loop over a Navigation	176
Figure 66. Simple Operation Invocation	177
Figure 67. Operation with Parameters	177
Figure 68. Sending of Signal (no parameter)	178
Figure 69. Sending of Signal with Parameters	178
Figure 70. The Context of the Examples	179
Figure 71. Example SDL Specification	180
Figure 72. Mapping of Assignment Statement <code>var := 3;</code>	180
Figure 73. Mapping of Assignment Statement <code>var := var + attr;</code>	180
Figure 74. Mapping of Call Statement <code>call Op2(7);</code>	181
Figure 75. Mapping of Create Statement <code>p := create P1;</code>	182
Figure 76. Mapping of Output Statement <code>output s to p;</code>	182
Figure 77. Mapping of Conditional Statement <code>if (xin > var) var := 0;</code>	183
Figure 78. Mapping of Procedure <code>Op2;</code>	184

Preface

This proposal is a response to OMG RFP ad/98-11-01, Action Semantics for the UML. This chapter summarizes the relationship of this proposal to the RFP. The main body of the document describes the technical proposal itself.

0.1. Proposal summary

Copyright waiver

This document may be freely copied by the OMG or by any member of the OMG.

Submission contact point

Steve Mellor, steve@projtech.com

Guide to material in the submission

See Chapter 1, “Overview”.

Overall design rationale

See Chapter 1, “Overview”.

Statement of proof of concept

See Chapter 1, “Overview” and Appendix B, “Action Language Examples”. The latter presents examples of constructs from various current action languages as an illustration of how the concepts in this proposal would be applied and also as a proof of concept of the approach.

0.2. Resolution of RFP mandatory and optional requirements and issues

This document is submitted in response to the Request for Proposal (RFP) from the OMG for an Action Semantics for the UML (ad/98-11-01). The submission meets the requirements of the RFP, as shown in the table below.

Table 1: Resolution of RFP requirements

Requirement (RFP sections)	Solution
Relationship to existing OMG Specifications	The submission is consistent with and uses the terminology of the forthcoming UML 1.4 specification.
6.5.1: Be an update to UML 1.4	This initial submission is an update to UML 1.4, as it is currently in revision. The final submission will be an update to UML 1.4 assuming that is adopted before the final deadline. See also Chapter A, “Updates to UML” for changes to UML.
6.5.2: MOF Compliance	The submission proposes additions to the UML metamodel that contain a few usages of multiple inheritance and association classes. If these are not supported in MOF 1.4, then the final submission will be modified to be MOF-compliant, or the issue will be addressed as a change to the UML physical metamodel. The final submission will be compliant with MOF 1.4 if that is adopted before the final deadline.

Table 1: Resolution of RFP requirements

Requirement (RFP sections)	Solution
6.5.3: Software-Platform Independence	The submission adheres to the the platform-independent nature of UML. It enables mappings to multiple implementations by hand or automatically. See Chapter 1, “Overview” and subsequent references.
6.5.4:Data Access	Chapter 8, “Read and Write Actions” covers this requirement.
6.5.5: Functions	Chapter 9, “Computation Actions”covers this requirement. Modelers can combine computation actions with those in Chapter 10, “Collection Actions” to operate on collections.
6.5.6: Self-reference	The Class ReadSelfAction in Chapter 8, “Read and Write Actions” covers this requirement.
6.5.7: Collections	Chapter 10, “Collection Actions” provides several actions for manipulating collections and their contents.
6.5.8: Associations	Chapter 8, “Read and Write Actions” provides the association operations required by the RFP. That chapter defines additional semantics for associations in cases where UML is unclear in its specification.
6.5.9: Order of Execution	Chapter 6, “Action Foundation” satisfies this requirement by making control flow optional and providing for parallel control flows. Data flows place the minimal constraints on execution order required to supply inputs to actions.
6.5.10: Constraints	This submission supports the specification of procedures that have no side effects and return a Boolean value, that is, a constraint. It does not specify the enforcement of constraints or detection of violations. It also does not define exactly how procedures are used as constraints. For example, it is not specified what inputs a procedure should have when used as a precondition, or when the constraint should be tested.
6.5.11: Behavioral Components	Chapter 9, “Computation Actions” supports uninterpreted behavior.
6.5.12: Properties	Uninterpreted properties are modeled with tagged values.
6.5.13: Extensibility	This submission proposes additional metamodels for UML that can be extended in the same ways as existing UML metamodels.
6.5.14: Execution Profiles	The submission covers actions models for methods and for state machines attached to classes. It inherits the semantics for other applications of actions from current UML, for example, in collaborations, and methods modeled as activity graphs. Areas in which execution may be profiled and a mechanism for doing that will be given in the final submission.

In addition to these requirements, the RFP requests that the submission discuss several specific topics as follows:

Composition

Proposals should define how the action semantics treats composition (i.e. if a whole is deleted by an action statement, what does this mean with respect to the parts?)

This submission assigns the most minimal semantics to the read and write actions. Such an approach places the least constraint on modelers and does not attempt to normalize the semantics of contentious areas like composition. Consequently, a cascading delete is defined semantically in terms of individual, separate deletes. A surface language can define a higher-level action that carries out a cascading delete in one step.

Proposals should describe how the action semantics treats multiplicity and conditionality of associations (i.e. if an instance is deleted by an action statement contrary to the multiplicity and conditionality specified on an association, what is the responsibility of the action semantics?)

In most cases, no semantics is given in this submission when an action violates aspects of static UML modeling that constrain runtime behavior, including the various constraints on creating links like changeability and multiplicity. The only exception is minimum multiplicity, which is given a semantics equivalent to the lower multiplicity being zero, that is, the lower multiplicity is ignored. However, reading links is not affected by any constraints that may have been violated in their creation and destruction. This accounts for cases where modelers provide semantics for areas which this submission does not cover, for example, creating links that violate maximum multiplicity. Reading links is only constrained by navigability and visibility.

Proposals should define the semantics of an object instance's lifetime. Is the user responsible for deleting objects to maintain consistency of the association?

In keeping with the philosophy outlined above, an object can be deleted only by explicitly using the action for that purpose. The user is responsible for deleting objects to maintain the consistency of an association, if that is how they choose to repair the inconsistency.

Proposals should indicate if and how an action can invoke an operation. Although the action semantics may be used to define the semantics of an operation, it is far from clear what the semantics of invoking a non-data-access operation of a target class are in the context of the run-to-completion semantics of the state chart. In particular, proposals should discuss how call semantics and signal-passing semantics coexist. Proposals that tackle this issue should define the semantics carefully.

The submission defines a call in the standard manner, synchronously; hence run-to-completion of a procedure in a state machine will include the called operation. It is the user's responsibility to ensure that a called operation, whatever the caller, does not change object state in a manner inconsistent with a state machine attached to the object. Chapter 11, "Messaging Actions" supported by Chapter 5, "State Machine Execution" describe this area in detail.

Proposals should indicate which elements of the action specification can be extended using stereotypes or tagged values.

Any element in the metamodels of this submission can be extended using the extension mechanisms of UML.

Proposals should provide for the specification of the generation of UML signals (e.g., under what conditions is the signal generated, which object or objects are the intended recipients, what actual parameter values accompany that signal, etc.). Proposals should also discuss queuing and handling of signals, defers etc. with and without ordering.

The submission defines the semantics of an action that can send a signal, with data carried along in the manner given by the UML. Sending a signal to a set of objects is handled using collection actions. Event occurrences are not queued as such. They are captured in terms of snapshots of objects that maintain links to event occurrences that have arrived at a state machine, but have not been dispatched. This submission does not define the semantics of deferred events, but also does not erase or modify the semantics defined currently in UML. This principle applies to all areas of state machine semantics not covered by the submission. See Chapter 5, "State Machine Execution" for more information.

While OCL compliance is not a requirement, proposals should discuss compliance with OCL, and justify deviations. This requires an implicit reverse-engineering of OCL into the Action Semantic model.

OCL is a surface syntax that could be mapped into the action semantics. The submitters invite interested parties to build a mapping from OCL to the action semantics metamodel. The OCL used for specifying formal semantics in this submission is extended in ways that do not conflict with current OCL. See Section "UML Object Constraint Language (OCL)" on page 18 and Section "Execution Rules" on page 22.

A suggested format for a proposal includes the following: (1) A UML model of the proposed Action Semantics. Primitives of this model include, and possibly extend existing UML Actions; (2) A model of an execution engine; and (3) A textual description of the expected behavior of each semantic construct within the execution engine. This shall be sufficient to provide a set of requirements for a simulator engine running UML models.

(1) There is a UML model of the actions in each chapter of Part III, Actions.

(2) Rather than defining semantics in a purely operational or denotational way, the underlying model is a synthesis of denotational and operational approaches. Constraints expressed in OCL are defined on histories of snapshots of entities as they change. Any execution engine that meets these semantics is an acceptable translation of the user's specification.

(3) There is a textual description of the basis for abstraction of each action. The precise semantics are defined using OCL against a model of execution.

Proposals will be evaluated based on how well they:

- *can be used to build complete and precise models that specify problems at a higher level of abstraction than a programming language or a graphical programming system.*

The actions defined here support specification at a high level of abstraction, and the submission makes little effort to support programming language constructs.

- *can support formal proofs of correctness of a problem specification.*

This submission supports profiles that enable formal proofs of correctness. For example, a profile may focus on primitive actions that maintain no system state of their own. They may access data, or carry out a computation, in which case boundary conditions can be established for them.

- *make possible high-fidelity model-based simulation and verification.*

Models with actions are executable, and (almost) complete, which enables high-fidelity model-based simulation and verification. However, to be complete, there must be definitions of primitive functions such as addition, string manipulation, Bessel functions and the like. These are defined in a profile. Once included into the simulation and verification environment, the action semantics are complete. A very few areas remain to be profiled for final submission, for example, the order in which event occurrences are dispatched from the unhandled occurrences of a state machine.

- *enable reuse of domain models, because each domain is completely specified, though not in a form embedded in code.*

Because the submission does not provide for constructs outside the abstraction level required for domain specification (i.e., it does not provide programming constructs), and the model is complete, each domain may be specified completely, but not in a form embedded in code. This enables reuse of domain models.

- *provide a stronger basis for model design and eventual coding.*

Actions require the user to think carefully about the sequencing and data access in a system, as well as the more detailed computations. The more a user applies the action semantics to a model, the stronger basis there is for both model design and coding.

- *support code generation to multiple software platforms.*

Because the constructed models are complete, and each action is primitive, with minimal interaction with other actions, the form of the model can be rearranged to support multiple software platforms and their execution engines.

0.3. Technical Proposal

The technical proposal is presented in the main body of this document.

0.4. Compliance Issues

Optional versus mandatory interfaces

This proposal describes the format for expressing UML models dealing with computational behavior and does not contain operational interfaces as such.

Proposed compliance points

This proposal is intended to replace the existing UML constructs dealing with actions and the specification of computational behavior. It is intended that the entire action semantics be an optional compliance point for UML tools not requiring such specification. Within the proposal, no subordinate compliance points have been defined. It is intended, however, that tools be allowed to specify various surface action languages in their supported models, and access to raw action semantic models by users is an optional compliance point.

Changes or extensions to adopted OMG specifications

This proposal is intended to replace certain portions of the previous UML 1.3 and forthcoming UML 1.4 specification and to become an integral part of the UML specification. Finalization of these changes will be subject to OMG procedures for integrating adopted specifications. The changes to UML are summarized in Appendix A, “Updates to UML”.

Part I. Summary of Proposal

Like Gaul, the submission is divided into three parts:

- **Introduction:** Provides an overview of the submission, and describes the structure of the chapters;
- **Execution Semantics:** Provides a formal, denotational basis for describing the semantics of UML as a whole in terms of UML; and
- **Action Semantics:** Provides the model for actions, the rationale for the choices made, and class definitions.

The Introduction, like the submission, is also divided into three parts:

- An Overview of the proposal (recommended reading)
- A description of the manner of description of the static model, including chapter formats and OCL
- A description of the manner of description of the dynamic model, i.e. how we chose to describe the dynamic semantics of actions.

The Execution Semantics describes an execution model, used to define the semantics of actions. The execution model could be applied to the entire formalizable part of UML, but that work is outside the scope of this submission. Therefore, this part cover only the application of the execution model to those portions of UML affected directly by actions: object memory and execution of procedures.

The Action Semantics part describes the user-visible actions as well as the classes that model the execution of the actions.

If you wish to come to grips with the kinds of actions defined in the submission, but not their details, you can achieve this quickly by reading the Overview Chapter, the immediately following chapter describing the static model conventions, and the class descriptions in each Chapter in Part III.

If you wish to understand the model of execution defined by the submission for use in describing the semantics of UML in general as well as the action semantics, read the Introduction (Part I) and Execution Semantics (Part II).

There are two appendices. The first appendix covers the changes to UML required or suggested by the submission.

The second appendix contains examples that show mappings from various existing action languages into the model proposed by the submission.

In addition, the document <http://cgi.omg.org/cgi-bin/doc?ad/00-08-01> on the OMG web server describes a mapping from the SDL language to the action semantics. The purpose is to demonstrate that the action semantics is capable of representing a complete, complex, and practical action language.

I Summary of Proposal

Chapter 1. Overview

This chapter describes the principles applied in the development of the submission, provides a high-level overview of the technical content, guides the reader through the document, and answers some frequently asked questions.

1.1. Action Semantics

This submission is a response to a request for proposal for an action semantics for the Unified Modeling Language.

An *action*, broadly, is some computation, such as executing a function, sending a signal, reading or writing data, and iterating over a set. At present, the actions defined for the UML are little more than placeholders. The modeler can indicate a few critical properties, such as sending a signal and invoking a function, but then must resort to writing code, because only a few special-purpose actions are defined. To be useful as more than a tool for sketching software structure, the UML repertoire of actions should be extended to be able to specify computation, making UML an executable specification language.

The UML, as extended by action semantics, however, should not be taken to be a new programming language, or as a graphical programming language. There are already many programming languages that can be used with UML, or could be formally integrated with it, and the world doesn't need yet another programming language. Rather, the action semantics provides for the *specification of systems in sufficient detail that they can be executed*, and the action semantics should provide *just enough* semantics to enable the specification of computation.

A standard action semantics enables interchange of complete specifications in UML. Because the semantics are independent from the syntax, users can develop models using their own favorite syntax or language. Moreover, the definition of a standard syntax, being a matter of taste, is likely to be a lengthy and contentious process. Defining the semantics first also allows multiple conforming syntaxes to be developed onto a solid base.

The action semantics defines primitive, simple constructs. A particular action language could implement each semantic construct one-to-one, or it could define higher-level, composite constructs to offer the modeler both power and convenience. These higher-level constructs do not need to be defined as a part of the semantics, because they already exist as composites of primitives.

The semantic primitives are defined to enable the construction of multiple execution engines, each with potentially different organizations. Hence, a model compiler builder can optimize the structure of the software to meet specific performance requirements, so long as the semantic behavior of the specification and the implementation remain the same. This then requires a mapping from the structure of the specification to the structure of the implementation.

These, and other features, are responses to requirements called for in the RFP. We strongly recommend that the reader first re-examine Chapter 6 of the *RFP* to set context for this submission.

1.2. Principles

Scope

The submission proposes a definition for the semantics of actions in UML, completely replacing the current placeholder in UML 1.4, *Figure 2.15: Common Behavior—Actions* and the associated class descriptions. The submission also proposes a limited number of other changes to the UML specification, gathered together in *Appendix A: Changes to UML*. The submission does not change or extend the notation of UML.

In accordance with the RFP, this submission does not propose a notation for actions, only their semantics. Once the semantics are agreed, a notation for actions—an action language—can then be defined. This semantics-first approach has the advantage of focusing attention first on the critical issue of semantics while delaying the rather more contentious issue of syntax. This also allows for multiple conforming syntaxes for quite different languages; some could be graphical.

This submission provides example syntaxes defined consistent with the semantics using several existing languages, as well as some invented notations, as an aid to understanding the rather abstract content of the proposal. This approach makes the semantics concrete without prejudicing the syntax.

The action semantics definition, like the existing definition of actions, is intended to be an integral part of UML. It may be packaged so that it is an optional compliance point, but actions are as much a part of UML as uses cases, deployment diagrams, or activity graphs.

Independently, activity graphs and diagrams provide a semantics and notation for the definition of control, data, and event-driven activities that may be used as methods of operations. There is overlap in the semantics of activity graphs and actions as proposed in this submission, particularly in the area of control and data flow.

Collaborations use actions to specify the interaction of instances or of roles. These interactions can send signals, invoke operations, create and destroy objects, that is, any action that one object can perform on another. This submission does not explicitly address the use of actions in collaborations, especially in the case of role interaction.

This submission does not make a systematic correlation between actions and activities, though every effort has been made not to be arbitrarily inconsistent. We expect that work on the relationship of activity graphs and collaborations to action semantics will take place in the future. This allows this submission to focus its effort on providing a base for further work in as consistent a way as possible.

Context

State machines may be used to describe use cases, classes, subsystems, packages, and behavioral features. Potentially, a state machine for one of these has a different context than for another. This becomes an issue when they communicate: Is the meaning of a `SendSignalEvent` action the same when sent from a state machine for a class to a state machine for another class, as it is if a state machine for a class were to send a signal event to a state machine for a use case? This issue of special interest to the action semantics submitters, as it falls to them to define the semantics of a `SendSignalEvent` action, and other actions that realize communications, such as an invocations or call events.

The number of potential definitions for communications is combinatorial. There are many places where a procedure can appear, and many ways to communicate. These communications then affect a procedure in a state machine or method for a use case, a class, a subsystem, and so on. This submission takes the position that we should define the semantics of these communications in some well-defined context, but it is not feasible at this stage to define the semantics in all possible contexts.

Consequently, this submission defines the semantics for synchronous and asynchronous interactions between both operations and state machines *defined for classes*. Each instance of a class, i.e. each object, has its own state configuration, as does the class itself. This does not prevent the use of the action semantics for other uses of the state machine, such as use cases or activity diagrams, but this submission does not define how these interact.

Undefined Semantics

There are two kinds of undefined semantics in this submission: those that *remove* semantics from UML and those that *leave* the semantics as defined by UML. The internal specification of each action may remove semantics from UML. For example, the semantics of actions on features with classifier `targetScope` is undefined in this submission, and would remain undefined after it is integrated with UML. However, the way actions are used leave the semantics to UML. For example, the semantics of using actions in collaborations, or in state machines used as methods, is undefined in this submission, and the semantics are inherited from UML.

There are several cases where semantics may be left undefined by this specification:

- There is no general agreed-upon meaning.
- The meaning is ambiguous.
- The action or execution foundation is not able to support the desired semantics yet.

An example of the first case is that no semantics is defined for using `CreateObjectAction` to instantiate an abstract class. An example of second is using `WriteAttributeAction` on an ordered attribute without specifying an insertion point. An example of the third is class `targetScope` for attributes and association ends. These cases of undefined semantics are described by well-formedness rules applying to the user model, and the execution rules applying to runtime execution.

The fact that no semantics is given for these situations does not mean that users cannot define their own semantics. Even though the action semantics uses the value-laden term “ill-formed” for these models, it is purely technical word and should not be taken to mean the specification excludes semantics from ever being defined for these cases. For example, some users might want to instantiate abstract classes for purposes of testing the root classes of their model. Or some may want the lack of insertion point in WriteAttributeAction to mean that the value is inserted at the end. Finally, some users may even want to extend the action semantic foundation to support class targetScope. These users could agree on semantics among themselves that covers their needs and does not conflict with this specification. That is perfectly consistent with the action semantics and the UML in general.

Specification and Software Structure

The UML provides a rich variety of representations for the structure of object-oriented software, including methods of parameter passing, deployment, components and the like. However, many of these constructs are not necessary for the specification of the semantics of actions, and only a limited number of constructs are required to model computation completely.

At the very simplest, actions need access to data, they need to transform and test data, and actions may require sequencing. A simple, sequential model could be adequate relative to a sequential computing environment, but it is not adequate today because it is unreasonably costly to map to today’s distributed computing environments. Consequently, the specification language has to include concepts of distributed, concurrent execution. This submission therefore allows for several (logical) threads of control executing at once, and synchronization mechanisms to ensure that procedures execute in a specified order. Semantics based on concurrent execution can then be mapped easily into a distributed implementation.

On the other hand, the fact that a specification language allows for concurrently executing objects does not necessarily imply the software structure. Some implementations may group together objects in to a single task and execute sequentially—so long as the behavior of the implementation is the same as the specification. The implication is that the specification language does not need to specify software components, such as tasking structure. Indeed the data and behavior of a “class” in the specification need not be rendered as a class in the implementation.

This submission proposes a concurrent semantics in which each instance of an object has its own state configuration. The action semantics requires the specification of only the required sequencing between procedures as defined in UML, and between actions inside procedures, as defined here. The implication is that the action semantics do not require the specification of tasking structures or of different forms of transfer of control in the implementation, only synchronous and asynchronous communication to sequence procedures in state machines and objects.

In short, the modeler can define concurrent, distributed abstract behavior, but not the software structure.

Interface to UML

Actions on a UML model make certain assumptions about their context within UML. There are assumptions about the static structure of a model, such as which data items an action can manipulate and which operations can be invoked, and there are assumptions about the dynamic behavior of the model at execution time, such as which transitions fire when an action sends a signal event. Taken together, these assumptions define an interface between the action semantics and the remainder of UML.

This submission defines the interface in terms of a *procedure*. A procedure is a group of actions caused to execute as a unit by the semantics of UML. Examples include a method of a class, a group of transition actions, a group of exit actions, and a group of guard actions. A procedure is much like an action, but it also takes a set of parameters that supply data from the remainder of the UML model.

This submission relies on the UML to define the semantics for state machines and invocations and therefore to define when a procedure executes. The action semantics defines the behavior of the actions in the procedure and exactly what data is accessible to the procedure once triggered.

Primitives

This submission defines the semantics in terms of primitives that can be composed by the user into larger groupings. For example, in a composition association where the deletion of an instance implies the deletion of all its components, the submission defines the delete action to remove only the single instance, and the submission requires further

deletions for each of the component instances. A surface language, however, could choose to define a delete-composition operation as a single unit as a shorthand for several deletions that cascade across other associations.

This submission, then, expresses the fundamental semantics in terms of primitive behavioral concepts that are conceptually simple to implement. The semantics do not define any “magic” that takes places behind the scenes. Modelers can work in terms of higher-level constructs as provided by their chosen surface language or notation.

Mappings

There are potentially many ways of implementing the same specification, and any implementation that preserves the information content and behavior of the specification is acceptable. Because the implementation can have a different structure from that of the specification, there is a *mapping* between the specification and its implementation. A one-to-one mapping would implement each class as a class and each state machine directly. But the mapping need not be one-to-one: an implementation may not use classes, or it might choose a different set of classes from the original specification.

The submission defines the action primitives to enable the maximum range of mappings. Specifically, we define the primitive actions so that they either carry out a computation or access object memory, and never both. This approach enables clean mappings to a physical model, even those with data organizations different from that suggested by the specification. In addition, any re-organization of the data structure will leave the specification of the computation unaffected.

The mapping may be carried out by hand by overlaying physical models of computers and tasks for implementation purposes, or the mapping could be carried out automatically. This submission neither provides the overlays, nor does it provide for code generation explicitly, but the submission makes both approaches possible.

Execution Engines

The specification may be implemented in one of several different *execution engines*, each of which may have different performance characteristics. For example, one engine might be fully sequential within a single task, while another may separate the classes into different processors based on potential overlapping of processing, and yet others may separate the classes in a client-server, or even a three-tier model.

The modeler can provide “hints” to the execution engine when the modeler has special knowledge of the domain solution that could be of value in optimizing the execution engine. For example, instances could—*by design*—be partitioned to match the distribution selected, so tests based on this partitioning can be optimized on each processor. The execution engines are not required to check or enforce such hints. An execution can either assume that the modeler is correct, or simply ignore it. An execution engine is *not* required to verify that the modeler’s assertion is true.

1.3. Defining Semantics

UML is a language for expressing system models. We create models of a system to describe the behavior of a system that already exists or to specify the behavior of a system that is to be built. But what, exactly, does it mean for the elements of such models to *specify* corresponding elements of the system? It is the semantics of UML that provide the answer to this question. To define precisely the semantics of UML actions, more formality is needed than was used in the UML 1.4 specification. This section describes how.

The semantics of a rich language such as UML is usually defined by mapping it to a simpler language whose deductive rules are already well known, generally a mathematical language. This mapping can be thought of as producing a kind of system-level “canonical implementation” for a UML model and is formally the *meaning* of the model. In this formal sense, objects in the semantic representation for UML *are* the system-level objects modeled by UML models.

For example, the structure of UML static modeling elements can be represented fairly straightforwardly using the typical mathematical language of sets, tuples and predicate calculus. Given a representation for the UML structural model elements, the meaning of behavioral model elements is a specification of how the represented instances change over time. Thus, if the values of attributes of an instance are represented as a sequence of name-value pairs, then the meaning of an operation of the instance that changes the attributes is a function that maps the sequence of pairs at one time to a sequence with new values at a later time. This is a denotational approach to semantic definition.

A drawback of such a mathematical representation of UML semantics is that it can be rather cumbersome, precisely because the mathematical language is less rich than the UML notation it is representing. However, with some care, we can make the mapping between (some subset of) the UML static modeling elements and the mathematical representation bidirectional. We can treat the UML static modeling notation as simply another notation for the mathematical representation. That is, we can use UML to model the memory kept by a *virtual machine* executing a UML model. The behavioral semantics of UML are then described by how this memory changes as the machine executes. This gives an *operational* viewpoint to the semantic definition.

However, this operational viewpoint poses a problem for formal definition of behavioral semantics. With a mathematical representation, the behavioral semantics are given by mathematical transformation functions from one time to another. With a UML-based model of the virtual machine, corresponding transformations would seemingly have to be defined using UML behavioral models. But it is the semantics of these very models that we are trying to define!

To avoid this circularity we model the concepts of the *identity* of some instance, which does not change over time, separately from a *snapshot* of the properties of that entity over some limited time interval. A “change in time” is then represented as the relationship between predecessor and successor snapshots and sequence of such changes forms a *history* of the entity over time. We can then define the behavioral semantics of the entity in terms of the “static” model of such sequences: a denotational approach that does not require the use of UML behavioral modeling.

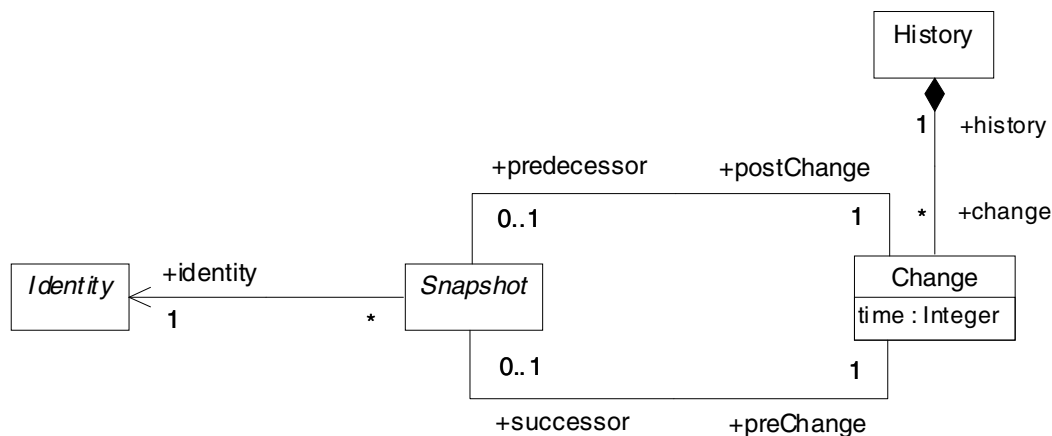


Figure 1. Identity and snapshot execution model

Figure 1 shows the fundamental model of identities, snapshots and histories. Figure 2 shows a schematic example of a small history. Note that the initial change in any history is a “creation” that has no predecessor snapshot, while the final change is a “destruction” that has no successor. Snapshots exist, then, only between changes, and so a snapshot always has a pre- and post-change.

As shown in Figure 1, each change is tagged with the specific time at which it occurs. A snapshot is considered to be valid between the times of its pre- and post-changes. Note that these times are purely “local” designations, independent of the times in other histories. There is no “global” time across histories. Thus, our fundamental execution model makes no assumptions about system-wide synchronization or distribution. Any required synchronizations between snapshots in different histories must be explicitly indicated.

The definition of the semantics, then, relies on modeling histories of snapshots, with each snapshot modeling an entity at some point in time as viewed by some observer. Even so, one can still *think* of the entity as “changing over time.” Intuitively, viewing a sequence of snapshots over time (as in Figure 2) is like a movie, each frame giving the mental picture of a single entity with fixed properties, but the sequence of frames showing the changing properties over the course of its history. This nicely reconciles the formal, denotation specification of the behavioral semantics with the intuitive, operational viewpoint.

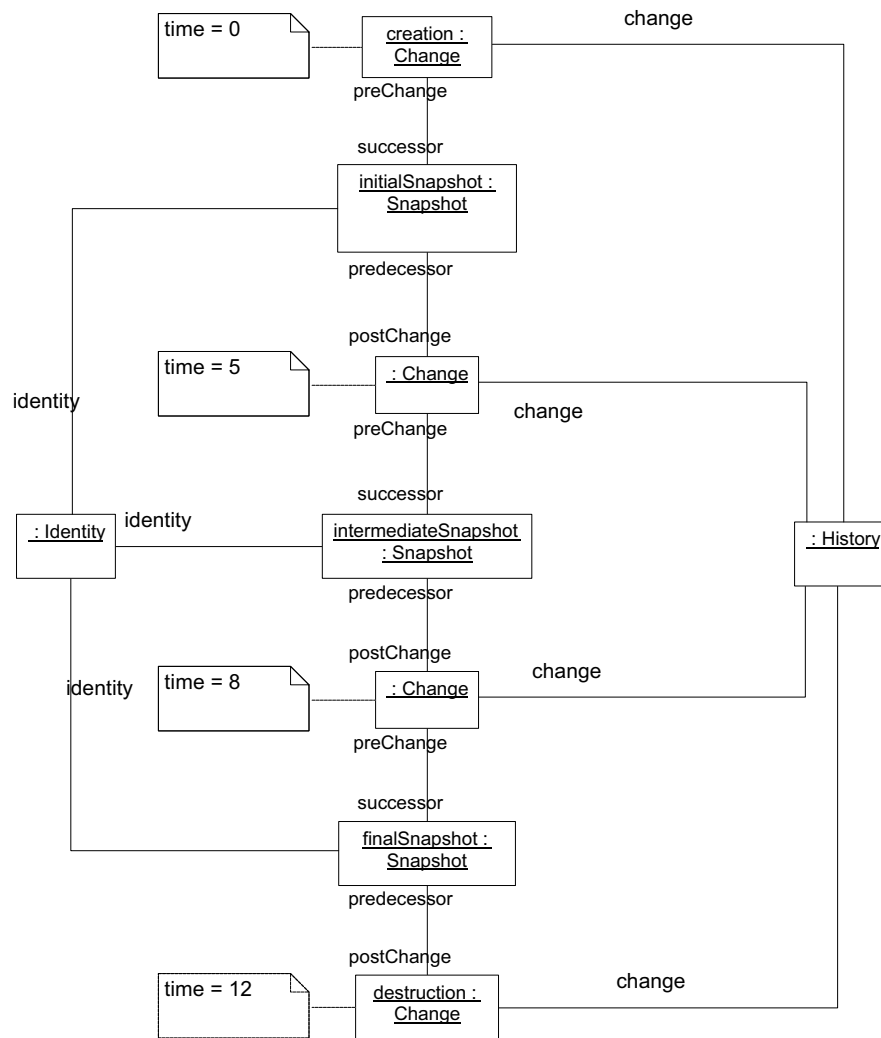


Figure 2. Example history

1.4. The Execution Model

This section describes the execution model used to define the semantics.

Snapshots and Local Time

A user model of a domain is an abstract representation of desired information and behavior for typical but unspecified instances. When the user model executes, it affects specific, identified instances, creating them and changing their links and attribute values. A complete definition of the action semantics must therefore model these user instances, and show how they change over time. We call this model of instances the *instance model*.

Mutable entities, such as instances, links and attributes, change their values over time. At some instant, a mutable entity has a value. Then an action changes it and it has a new value. At each instant, we may record the values associ-

ated with a single mutable entity in a *snapshot*. To capture the information content, a single snapshot is required whenever there is a change in associated values for the mutable entity.

The life history of a mutable entity can be captured as a sequence of snapshots, each of which captures the values for the mutable entity at some time. Two adjacent snapshots for a mutable entity represent the pre- and postconditions for the action that caused the entity to change. Looking at this from the perspective of the action, the semantics of the action can be defined in terms of pre- and post-conditions. The pre-condition is the snapshot immediately before the action executes, while the post-condition is the snapshot immediately following its execution.

This submission defines an *execution model* in terms of the changes to the instance model over time. Each change to any mutable entity of the instance model yields a new snapshot, and a sequence of snapshots constitutes a history for the entity. The submission then defines the semantics of actions in terms of the execution model. Note that the modeler of a model does not have access to, or need to know about, the concept of a snapshot, or indeed any other part of the execution model: these are purely internal concepts used to define the semantics.

Time Between Instances

Snapshots are purely local. A sequence of snapshots for a mutable entity forms a local view of time for that entity. There is no concept of a snapshot for the entire system; only for one mutable entity at a time.

However, multiple instances can execute “at the same time.” This submission takes the view that each instance has its own view of time, in which signal events arrive in sequence, but there is no guarantee of ordering between instances except insofar as they are explicitly sequenced. Time, then, behaves as it does in the real world: relativistically. There is no “global time,” and there is no interpretation for the phrase “at the same time.”

Consider, for example, three instances, unimaginatively named A, B and C, each executing on a different (logical) processor. If A first sends a signal event C1 to C, then a signal event B1 to B, and B sends a signal events C2 to C only after it has received the signal event B1, there is no mechanism that guarantees that order in which C will receive the signal events C1 and C2. If instance B is “fast” it could be that C2 arrives before C1, or vice versa. To guarantee an order, A and B must synchronize their activities with C; for example, if C sent a signal event A1, which had to be received by A before it sent C1, then we would know that C2 would always arrive before C1.

Because the relative timing of two instances cannot be assumed, the relative ordering of data access to instances cannot be predicted. Many instances may access data, but there is no guarantee of their relative ordering unless the instances are explicitly synchronized. This means that there is the possibility of “data access conflict” whereby one instance writes several attributes read by another instance, yet the reading instance get some values from before the write and others after the write. Again, it is the modeler’s responsibility to synchronize the instances if such synchronization is desired. In summary, because time is a local concept, access to memory is also local.

Execution of State Machines

The definition of the semantics of the state machine is outside the scope of the action semantics. However, the semantics must state exactly how procedures interact with state machines and classes, both in terms of the execution model and how procedures get their data. Accordingly, the behavior of state machines must be related to the execution model.

This submission refines the UML concept of an *event* into two related concepts: an event specification, and an event occurrence. An event specification is the definition of a kind of event that can be transmitted, while an occurrence is the actual presence of an event, destined for an instance, at some time. Note that, contrary to the text of the existing UML specification, an occurrence is not “an instance of a event (specification).” Rather, an event specification defines an archetype for the event, while an event occurrence is an abstraction of the actual transmission of an event. For example, we may have as instances of the event specification the events MakeDeposit and CloseAccount. Then, separately, we have instances of event occurrences: sending a signal event (occurrence) MakeDeposit to one’s account, say, which (one hopes) may occur repeatedly at different times. Each such occurrence of a deposit conforms to the archetype offered by the event specification.

An event specification includes a definition of the data carried by the event, such as the amount of money that must be deposited to the account. This defines a signature for all procedures that can be triggered as a result of an occurrence. In the case of a procedure on a transition, there can only be one procedure defined for the transition and the signature of the event specification will match that of the procedure. However, in the case of exit or entry to a state, several events each may lead out of or into the state, each of which could potentially have different signatures. Procedures used in state machines are required to have compatible signatures with all the events that might trigger

them, except for entry and exit procedures with no inputs. In this manner, hierarchical state machines can use procedures when the incoming signatures don't match and still get entry/exit procedures executed.

This submission also defines snapshots for occurrences and each state machine configuration. Hence, the progression of a state machine instance over time is modeled in the same way as instance data.

Each instance of a class that has a state machine associated with it has its own instance of the state machine in a particular state configuration. Each such copy is a *state machine execution*, which in turn has its own sequence of snapshots. The submission thus relates the execution model for instance data with the execution model for state machines.

1.5. The Actions

This section describes the motivation behind the choice of actions, particularly how they are structured and connected. The section also offers a quick summary of some of the actions themselves. The submission itself describes the actions in some detail.

Foundation

Traditional programming languages overspecify sequence, because actions, even within the same procedure, can potentially execute concurrently on different machines. For example, some execution engines might rely on a file server, and execute all data accesses on that separate machine, but over-specifying sequence inhibits such re-organization of the actions. The issue here is not concurrency of actions *per se*, but rather the requirement to be able to re-organize the actions for an efficient implementation.

This submission therefore treats all actions as executing concurrently unless explicitly sequenced by a flow of data or control. In addition, each action is defined so that is free of any context that describes how it used, so that data access and other computations are two separate primitive actions connected together, when required, so that each action is unaware of the source or destination of the data.

To meet these goals, this submission defines the concept of a *data flow*. A data flow carries data between two actions, and in so doing effectively sequences the actions. Hence we may execute a read action to access some data, flow that data to a computation action that computes the square, and then flow that result to a write action. The three actions *have to* execute in this order because each one cannot execute until the previous one produces its data.

The technical reasons for this choice are first that data flow makes this form of sequencing explicit. Second, because data flows are produced only once (as a result of a specific execution of an action), we may make assertions about the values on the data flow for verification and translation purposes. This is a property shared by *static single-assignment form languages* in which a variable can be assigned to only once. This sole difference between data flow and static single assignment in this context is that data flows use anonymous variables that come from a single source and go to potentially many destinations, while static single-assignment form allows a named variable to be written only once, and written many times.

A data flow connects to other components via *pins*. Each data flow has an input pin on one end and an output pin on the other. The input pin of the flow is necessarily the output pin from some other element, typically an action. And vice-versa.

This submission also provides the ability to read and write variables local to the procedure for upward compatibility with existing languages.

This submission also provides for control flows between actions for explicit sequencing that does not rely on data flow.

This submission makes use of the concept of data flow so that the sequencing by data between the actions is explicit. Any analysis of the semantics for testing, verification or translation requires knowledge of the data flow and the type of transformations shown in the fragments above. Using data flow also minimizes overspecification of sequence between the actions.

Group Actions

This submission provides for selection and iteration. In both cases, there is a need to group actions together so they may be executed as unit. Such groupings may be nested, and may accept and receive control flows. Data flows are

routed transparently to the actions involved. The presence of concurrency also affects how these traditional structures operate.

An IfAction comprises a group of clauses, which execute concurrently. Each clause has two parts: a test and a body. The result of a test may cause the execution of the associated body, or zero or more other tests within the same IfAction. The result is that potentially more than test may be true during execution, causing several bodies to execute.

The modeler may, of course, constrain the sequencing of the tests so that only one body can execute, and the structure behaves like a conventional if-then-else. The modeler may also assert that only one clause body shall execute *by design of the tests*, so once a body has been selected, no further tests or bodies are executed. No execution engine is required to verify that only one body can execute.

Some kinds of iteration require some degree of sequential execution, for example, a regression that takes the outputs of one cycle and feeds it as inputs to the next. This class of iteration that is managed by the LoopAction. (Iterations that can execute on the elements of a collection concurrently are handled differently, by Collection Actions. See below.) A LoopAction comprises a single clause, which in turn comprises a test and a body. The body is executed repetitively while the test is true.

Read and Write Actions

Objects, attributes, links, and variables have values that are available to actions. Objects have classifiers and they be created and destroyed; attributes and variables have values; links may be created and destroyed, and they have object ends, and qualifier values; all of which are available to actions. Read actions get values, while write actions modify values and create and destroy objects and links. Read and write actions share the structures for identifying the attributes, links, and variables that they are accessing.

Read actions do not modify the values they access, while write actions, as a general policy, have the most minimal effect possible. For example, creating an object does not execute constructors. Languages requiring additional semantics can define higher-level actions on the primitive ones given here. Unlike read actions, write actions may leave semantics unspecified in some cases. No semantics is usually given when an action violates those aspects of static UML modeling that constrain runtime behavior. For example, no semantics is given for creating an instance of an abstract class. The only exception is minimum multiplicity, which is given a semantics equivalent to the lower multiplicity being zero. Modelers of language bindings can assign their own semantics for undefined cases.

Computation Actions

Computation actions transform a set of input values to produce a set of output values. These actions work directly on values and produce values; they embody mathematical functions. They do not interact with object memory: they do not read or write attribute or link values. Also, they do not interact with other objects or other executions, so their control is entirely self-contained. These actions supply the primitive functions out of which computations are constructed.

This submission allows for the incorporation of primitive functions, such as mathematical functions or string manipulations, which may be gathered together to form a profile for specific uses, but it does not define a set of primitive functions as a part of the submission. This allows the actions to be extensible.

Collection Actions

Collection actions permit the application of an action to a set of data elements, possibly in parallel. They avoid the need for explicit indexing and extracting of elements from collections, avoiding the overspecification of control that would otherwise be necessary.

Each collection action contains a subaction, an embedded action that is executed once for each element in the input collection. There are four kinds of collection action. The map action applies a subaction in parallel to each of the elements of a collection of data, resulting in an output that is a collection of the same size and shape. The filter action selects a subset of the elements in a collection into a new collection of the same shape, based on a boolean result of the subaction applied to each element. The iterate action applies a subaction repeatedly to each of the elements in a collection, accumulating the effects in loop variables. The reduce action repeatedly applies a binary subaction to pairs of adjacent elements in a collection, implicitly replacing a pair of elements by the result, until the final result is a single element of the type in the collection. The binary subaction must be associative, therefore it may be

applied in parallel to many pairs of elements, because the exact order of application will not affect the final result. For example, summation or matrix multiplication are reduction operators.

Note that there is also a separate `LoopAction` that carries out actions repetitively where the outputs of the loop may be used as input on the next iteration.

Messaging Actions

Messaging actions exchange signals between objects. The messages may be synchronous (calls or invitations) or asynchronous (signals). In both cases, all information is transmitted by parameters passed by value. Those values may include object handles.

Exceptions

The submission does not presently include exceptions. However, the submitters view this topic as important; one that can be filled out later.

1.6. Frequently Asked Questions

This section answers briefly some frequently asked questions.

Why Don't We Just Use A Programming Language?

A programming language such as Java or C++ provides both too much and too little. It provides too much because it allows the modeler to access to implementation concepts (pointers or specifically allowable implementations of associations), yet it provides too little because a programming language does not understand UML concepts such as state, or attribute typing, nor how the concepts fit together. Programming languages also overspecify sequence.

Why Not Just Use OCL?

OCL is a perfectly fine surface language for the semantics defined here. See the Examples Appendix. Note however, that OCL explicitly does not permit changes to the instance model, and that's what the actions are all about! As OCL grows, though, it could become an action language, and if so, the semantics defined here would help define the semantics of OCL.

What Good Is An Action Semantics Without A Language or Notation?

The definition of the semantics provides for interchange of the semantics. For a modeler to build a model with these semantics there needs to be some conforming notation. There are already action languages in existence that conform to, or come extremely close to conforming to, the semantics defined here. In addition, OCL and SDL could be made compliant with a modicum of work.

How Does This Relate to Activity Diagrams?

Actions can be used in activity graphs in the same way they are used in state machines, because activity graphs are a kind of state machine. Each action state in an activity graph has an entry action specifying what is to be done at that step.

There is overlap between the semantics of action as proposed in this submission and activity graphs, because both support control and data flow modeling. The action semantics is more complete and precise in its modeling of control and data flow than activity graphs, but does not handle event-driven activities as activity graphs do.

How Does This Relate to Collaborations?

We have delayed showing how the actions proposed in this submission are used in collaborations. In particular, we have yet to normalize their application in role-based collaborations, because actions are taken on roles in this case rather than instances. We also have yet to normalize the interface between actions and the collaboration model generally.

The Submission Discusses State Machines. Is It Only Good for Real Time?

Objects have state, and state machines can model the behavior of objects irrespective of the subject matter, real time or otherwise. Consequently, the question is “ill-formed,” if frequently asked. In addition, the semantics defined here work fine if a model uses only methods. If you use only invocations on methods, think of kicking off the first method from a little state machine off to the side. After that, everything works synchronously. Finally, many people do use state machines, so this submission ensures that the two fit together properly.

Is This Submission Intended to be a Part of UML?

Yes. Today, actions are a part of UML. This submission replaces and elaborates the action semantics already in place. However, it is not yet clear whether it should be incorporated into UML 1.4 or constitute a new version UML 1.5. Also, we expect to package up action semantics so that they make an optional compliance point for UML.

Isn't It Incomplete?

Certainly. There are gaps between the definition of the actions and the precise behavior of state machines in multiple contexts. And there are additional elements, such as exceptions, we would like to add to the semantics. However, this is a good basis, and we need to know what else is missing to help produce the final submission.

Why Define the Semantics Using UML?

Because the UML is accessible to this audience, and because the work here can act as a basis for a more complete definition of the remainder of UML.

Why Did This Take So Long?

It didn't take that long—the time taken is less than that taken for the UML. True, there is less here than in UML, but actions require a more careful definition. Also, we chose to work with as many companies as possible as early as possible. This did take time because we had three entirely different points of view that had to be integrated over many meetings, but we expect that effort to pay off in now as we move into the final submission.

Chapter 2. The Static Model

This chapter describes how to interpret the static models and accompanying descriptions that comprise this proposal.

Chapters describing concrete action classes, such as reading, writing, messaging, and so on have additional dynamic semantics for execution, as described in Chapter 3, “The Dynamic Model”.

2.1. Chapter Structure

Each chapter begins with a brief introduction that outlines the content of the chapter, the theme behind the chapter, and any high-level design decisions or criteria that guide the chapter.

The following several sections introduce the key abstractions for readers. The material is organized for understanding, not for detailed reference. The purpose is to give the reader an intuitive feel for the concepts, and, particularly, to show how they work together and the reason that they are needed. These sections are not absolutely precise if this is at the expense of clarity, and they are not normative. Small examples, diagrams, and partial models help present concepts. Model diagrams may included in these conceptual sections (see Figure 3).

The next several sections, starting “<Chapter Name> Classes” define the formal precise semantics of all the classes covered in the chapter in alphabetical order. The intent is to describe the classes, including their attributes and associations, the well-formedness rules, and the semantics, both static and dynamic. These sections are normative.

Note. Notes, set off as shown here, draw attention to fine points or details to be addressed.

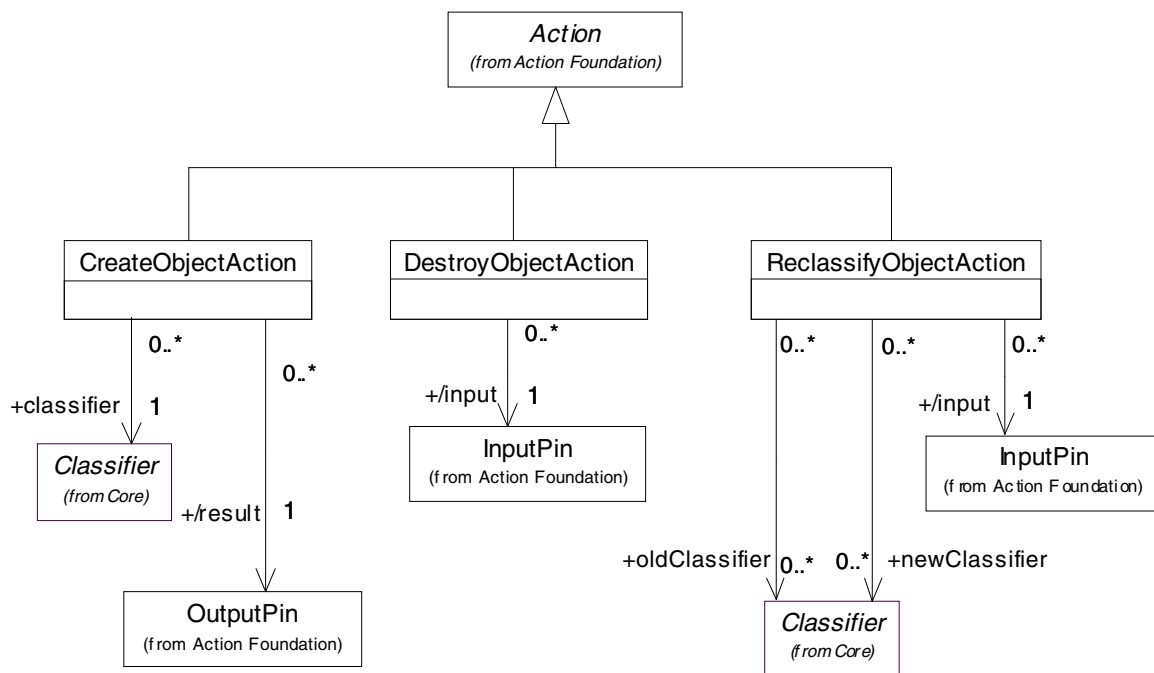


Figure 3. Example model (an action metamodel)

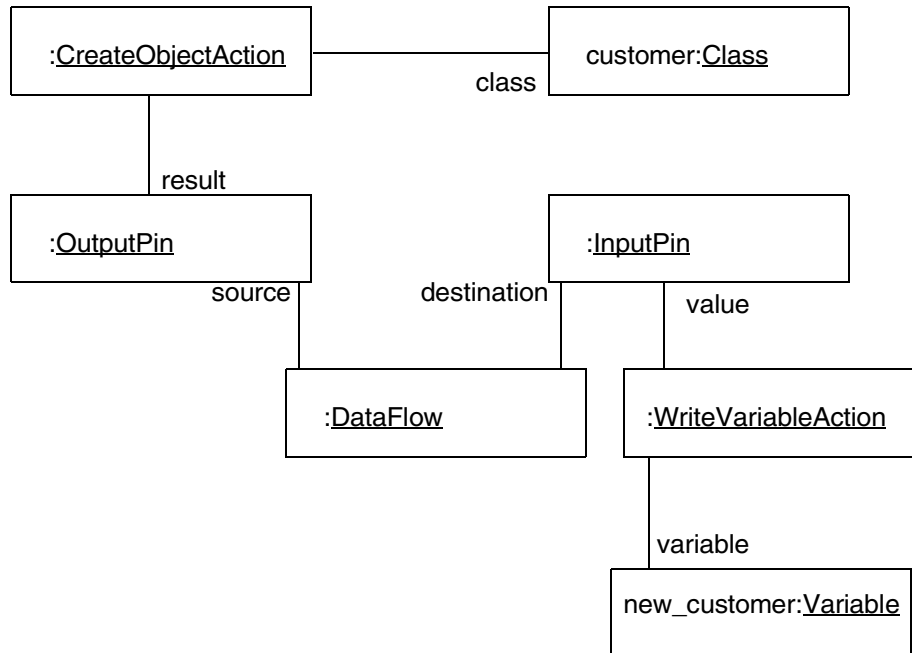


Figure 4. An Instance Diagram showing Actions in user model.

2.2. Well-Formedness Rules

To specify an action that creates an object, you make an instance in your model of the `CreateObjectAction` metaclass, using a surface language or some other mechanism, and then link it with the particular class you want to instantiate. The class to be instantiated, such as a `Customer`, is linked statically to the action in the user model, because a create action must necessarily specify—at “compile time”—what kind of class to create.

To pass the newly created object to other actions, the surface language (logically) makes an instance of the `OutputPin` class for the user model and links it to the instance of `CreateObjectAction`. Then the pin can be linked via data-flow constructs to input pins on other actions in the user model. Figure 3 shows this example with an instantiation of class `Customer`, and the new object being bound to variable called `new_customer`.

The action metamodel defines various *well-formedness rules* that apply to the user’s instantiation of the classes and the links created between them. For example, an instance of `CreateObjectAction` must have a class to instantiate specified with the action, as described above. Well-formedness rules ensure that the static aspects of an action will not get in the way of proper application of the execution rules. The well-formedness rules do not guarantee that execution rules will apply successfully, because all ill-formed execution scenarios cannot be described statically. A separate set of rules applies to execution, described in the next chapter.

To describe the model of actions formally, we shall need to define the classes therein such as those shown in Figure 1, their attributes, associations and the well-formedness rules that guarantee the correct construction of the instances as shown in Figure 2. The next section describes how.

2.3. Description of a Class

The formal class descriptions begin with a short summary of the technical meaning of the class. If it is an action, it briefly describes what the action does. It calls out notable semantic features that are given in detail by the formal semantics sections. One or two brief caveats or clarifications may be included in a second paragraph, if it helps to

avoid confusion with other concepts. This is *not* a recitation of its model syntax (i.e., not “An ABCClassName is an instance of metaclass Action”).

The following subsections are lists with specific formats for their contents. These sections are:

- Attributes.
- Associations.
- Well-formedness rules

A subsection is omitted if it has no elements.

Attributes

This section lists all attributes of the class, including their types and multiplicity. The models follow the naming conventions of UML 1.4, hence *attributeName* is used in preference to *attribute_name*.

For enumerations, a list of enumeration literals follows, one to a line:

If a well-formedness rule uses an attribute from a superclass, the attribute is said to be inherited, and it is marked `<inherited>` in the description. An abstract example is:

```
name: type [multiplicity]A description of the meaning of the attribute. For enumerations:
    foo—description of the literal
    bar—description of the literal
```

```
<inherited> inheritedName: type [multiplicity] First the name of the parent class then the description.
```

Here’s an example from the “Attribute action metamodel” on page 90:

```
isRemove: Boolean [1..1]Tells whether the action is adding or removing a value from the attribute. A value of
    true removes a value, and false adds a value.
```

Associations

This section lists all associations that have end names opposite from the class being described. If an association has no opposite end name, its description is omitted. All associations are described in at least one direction.

Inherited ends are listed at the end, and marked with the modifier `<inherited>`.

Associations are sometimes derived from more general ones to limit the kinds of classes that can participate. These are indicated by the modifier `<derived>`. The beginning of the description states the parent and association end from which an end is derived. An abstract example is:

```
endname: TargetClassName [multiplicity]Description of the association in the target direction
<inherited>inheritedEndname : TargetClassName1 [multiplicity]Parent class given first. Then a description of
    inherited association.
<derived> derivedEndname :TargetClassName [multiplicity]Parent class and end name is given first using the
    notation Class:endName. Then a description of the association.
```

Here’s an example from CreateObjectAction (see Figure 3 above):

```
classifier : Classifier [1..1] Classifier to be instantiated.
```

Well-formedness Rules

This section lists constraints on the model. The constraints are expressed first in English, then in the Object Constraint Language, OCL. The constraints only cover those aspects not expressed in the model already. For example, association multiplicity is not restated in the well-formedness rules. Well-formedness rules defined for a class are inherited by all its subclasses.

In the OCL, The ‘self’ variable refers to class being described, from which all association navigation must proceed. Any constructs used in the OCL that are not specified in OCL (as defined in the UML 1.4 specification) are defined in OCL itself, as an extension to the standard. If you are not familiar with OCL, a brief introduction appears in the next section.

Here's an example from CreateObjectAction:

```
[1] The classifier being created cannot be abstract. -- In English
    not (self.classifier.isAbstract = true) -- In OCL
```

Restating the details of the OCL constraint in English, we navigate from an instance of CreateObjectAction to the isAbstract attribute of its Classifier, and that value must not be true.

2.4. UML Object Constraint Language (OCL)

This section briefly describes the OCL and defines conventions for its use in this specification. See UML 1.4 for a full description of it. Additional conventions for using OCL in the execution semantics are given in the next Chapter.

What is OCL?

OCL is a part of UML designed for specifying operations and expressions operating on UML models, but have no side-effects on those models. It is primarily used to define boolean expressions that determine if a model is well-formed. These are sometime called *constraints*, hence the name of the language. The UML, including the action semantics, uses OCL to define all well-formedness rules for its models. Action semantics can be used for this purpose, but OCL provides a normative, human-readable syntax and many primitives and conveniences that make it particularly suited for writing operations with no side effects on UML models. In addition, it makes the semantics of UML non-circular. OCL can also define side-effectless operations and expressions that return other types, such as integers, reals, collections, and so on. Well-formedness rules often depend on additional OCL operations defined in a specification to make the constraints more concise.

As a very brief introduction, here is an explanation of an OCL operation used in the Action Semantics

```
context multiplicity::lowerbound( ) : Integer
post: multiplicity.range→exists(r : MultiplicityRange | r.lower = result)
and multiplicity.range→forall(r : MultiplicityRange | r.lower <= result)
```

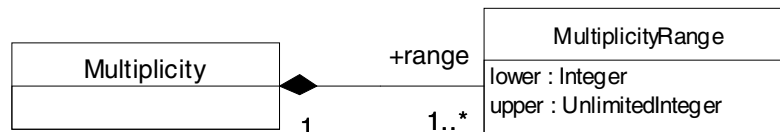


Figure 5. Multiplicity Model

The OCL operation *lowerbound* operates on instances of the UML class Multiplicity, which is a collection of ranges of integers. The operation returns the lowest lower bound of the ranges in a multiplicity. The operation is defined by declaring a postcondition. The OCL reserved word “result” refers to the result of the operation being defined.

The dot operator is used for operations on objects. The first clause of the conjunction searches over the ranges of the multiplicity, using the dot operator to navigate across an association with end name “range”. The result of applying the dot operator is a collection of multiplicity ranges. The “→” notation is used for operations on collections, so that the result from multiplicity.range above, which is a collection, is the operand for the operation exists(...). The expression r : MultiplicityRange introduces a variable r of type MultiplicityRange, which is then used in the remainder of the exists operation, up to the closing parenthesis.

The *lowerbound* operation checks that at least one of the ranges has the result as its lower bound. The second clause, using the forall, checks that all the ranges have lower bounds greater than or equal to the result. This ensures the result is the lowest lower bound of the ranges, which is the overall lower bound of the multiplicity.

Additional OCL operations

Additional OCL operations for use in this specification are defined for the following types.

Multiplicity

These operations provide a convenient way to work with multiplicities. The Multiplicity type is defined in the DataTypes package of the UML. Multiplicity data types can be created and enumerated the same way the predefined OCL types Set, Bag or Sequence are created and enumerated. For instance, you can write: Multiplicity(1..4, 6..*). The * maps to the “UnlimitedInteger” datatype.

- [1] <= operates on unlimitedInteger. It determines whether an unlimited integer is less than or equal to another.
 context UnlimitedInteger :: <= (ui2 : unlimitedInteger) : Boolean
 post: result = if self = #unlimited and ui2 = #unlimited then true
 else if self = #unlimited then false
 else if ui2 = #unlimited then true
 else unlimitedInteger <= ui2
 endif
- [2] contains operates on MultiplicityRange, taking an integer as input. It checks if a given integer is within the range specified by a multiplicity range.
 context MultiplicityRange::contains(i : Integer) : Boolean
 post: result = self.lower<=i and i<=self.upper
- [3] allows operates on Multiplicity, taking an integer as input. It checks if a given integer cardinality is allowed by a multiplicity.
 context Multiplicity::allows(i : Integer) : Boolean
 post: result = self.range→exists(r : MultiplicityRange | r.contains(i))
- [4] compatibleWith operates on Multiplicity, taking another multiplicity as input. It checks if one multiplicity is compatible with another.
 context Multiplicity::compatibleWith(other : Multiplicity) : Boolean
 post: result = Integer.allInstances()→forAll(i : Integer | self.allows(i) implies other.allows(i))
- [5] lowerbound operates on Multiplicity. It returns the lowest lower bound of the ranges in a multiplicity.
 context Multiplicity::lowerbound : Integer
 post self.range→exists(r : MultiplicityRange | r.lower = result)
 and self.range→forall(r : MultiplicityRange | r.lower <= result)
- [6] upperbound operates on Multiplicity. It returns the highest upper bound of the ranges in a multiplicity.
 context Multiplicity::upperbound : unlimitedInteger
 post self.range→exists(r : MultiplicityRange | r.upper = result)
 and self.range→forall(r : MultiplicityRange | r.upper <= result)
- [7] is operates on Multiplicity. It determines if the upper and lower bound of the ranges are the ones given.
 context Multiplicity::is(lowerbound : integer, upperbound : unlimitedInteger) : Boolean
 post: result = (lowerbound = self.lowerbound and upperbound = self.upperbound)

Downcast of Collections

It is sometimes useful to downcast an OCL collection, for example when a generic function returns an OCL Collection, but a particular usage requires a Sequence. Unfortunately, the OCL operators asSet(), asSequence() and asBag(), are defined only for the concrete types, and not for their abstract common supertype Collection. We assume they are defined on the abstract super-type Collection.

Chapter 3. The Dynamic Model

This chapter explains how to read the dynamic specifications of actions.

The approach to execution models taken here is based on *snapshots* of runtime elements as they change. Hence, the lifetime of an object is modeled as a series of snapshots, one snapshot for each point at which the object changes. An object snapshot lists the values of all the attributes of the object, all the association links in which it participates, and all the classes under which it is classified at the time. All the snapshots for an object are linked together in the order in which changes occur. The sequence of snapshots is the *history* for some *identity*, which represents the object as it exists independently of time.

3.1. Time Ordering and Causality

The snapshot concept is also applied to actions as they are executing as shown in Figure 6. A change in the history of such an execution is a *step* in the execution. Each step may require access to one or more other snapshots, as modeled by the accessor/referent association. A step may also cause one or more changes to happen, as modeled by the cause/effect association. A snapshot can be accessed by many steps from different execution histories, but that a change can have at most one cause. A change without an explicit cause is one that effectively happens because of the implicit execution machinery.

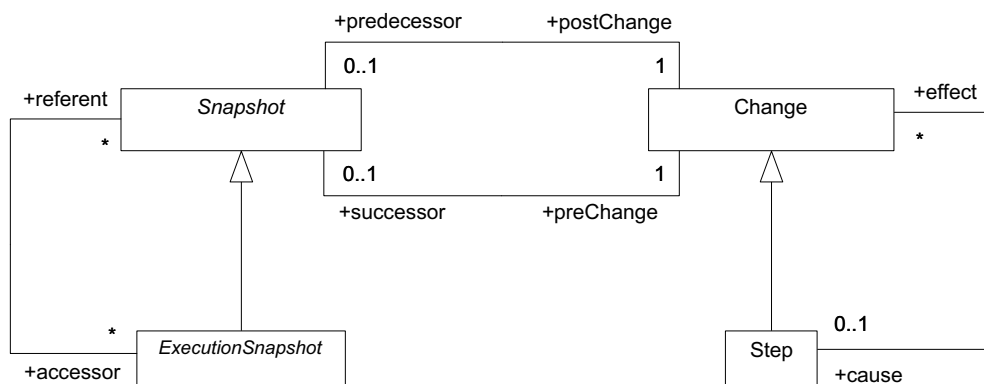


Figure 6. Execution dynamics fundamentals

Certain activities require the synchronization between entities. For example, reading an attribute of an instance requires that the value of that attribute be obtained relative to some specific snapshot of the instance. An entity references other entities by using their identities. An access to a snapshot obtains a specific snapshot from the history of the identified entity, but which snapshot, exactly, is not deterministic, because time is a local concept. All execution engines, though, must conform to the following constraints.

- *Time ordering.* If two execution snapshots in the same history access the same identity, then the later execution snapshot (relative to the “local time” of the execution history) will receive the later referent snapshot (relative to the “local time” of the accessed history).
- *Causality.* Closed causal loops cannot be allowed. It is not possible for step A1 in history A to be caused by step B2 in history B, step A1 to be followed by A2, step A2 to cause step B1 and step B1 to be followed by step B2.

Note. These are “global” constraints that apply across all histories involved in the execution of a model. They are not expressed formally in this submission.

3.2. Execution Rules

The snapshot concept is also applied to actions as they are executing, to capture the fact that they go through a lifecycle of waiting to be executed, being ready for execution with all inputs specified, executing, and completed.

Execution rules constrain and define behavior at runtime by relating an action execution snapshot to the one immediately after it in time. Action execution snapshots classes are not explicitly defined: there is no `CreateObjectActionExecutionSnapshot`. This is because there is not much interesting to say about it, except that it is the action execution snapshot of `CreateObjectAction`, which is obvious.

Similarly, action executions are elided when there is no interesting state to the execution between when it starts and when it finishes. For example, `CreateObjectAction` has no states between ready and completed, so `CreateObjectActionExecution` is not explicitly defined; but `ConditionalAction` has intermediate states as it moves through tests on its predicates, so it has various execution classes defined for it, such as `ClauseExecutionIdentity`.

The relationship between snapshots of action execution and snapshots of the objects they refer to requires some explanation. In a `WriteAttributeAction`, for example, the snapshot of the action execution must be able to refer to the snapshot of the object that has the attribute to test the multiplicity at the time a new value is added. However, the exact snapshot of the object being used depends on time at which the value is added relative to other actions that are operating in parallel, which could be changing the snapshot that `WriteAttributeAction` will modify.

The execution model given by the action semantics can handle the above situation, by navigating from the action execution snapshot—the context for the OCL description of the execution rules—to the identity on which the action is executing, then to the snapshot of that identity, and finally to the value on which the action executes. Lest the ‘L’ in OCL stand for logorrhea, all of this is also elided and only the value to be accessed is shown.

The dynamic semantics of a behavioral element is captured by a set of *productions* that determine the effects of the steps between execution snapshots for the element. A production has three parts: a *precondition*, a *postcondition* and *assertions*. We interpret a production as an execution rule in the following way.

- For every snapshot in a legal execution history for which the precondition is true, the postcondition must be true for the successor snapshot, thus relating two snapshots on either side of a step.
- For every snapshot in a legal execution history for which the precondition is true, the assertions must be true for *that* snapshot, so that if an assertion is violated, the meaning of the execution is simply undefined.

Assertion rules describe necessary conditions on the predecessor snapshot for it to be well-formed. For example, `WriteAttributeAction` requires its predecessor snapshot not to add a new value that will cause a violation of the multiplicity of the attribute. If the predecessor snapshot is not valid, the productions do not hold.

Any of the assertions or the postcondition may be omitted from the production. If the postcondition is omitted, then the production form is simply a convenient way to write a “state-dependent” execution rule. Details of the execution model are given in Chapter 4, “Execution Fundamentals”.

In writing preconditions and assertions, “self” always refers to the predecessor snapshot, while in the postcondition it refers to the successor snapshot. Access to other entities is always defined relative to this “self” execution snapshot. Preconditions may also be omitted, in which case they are assumed to be true. Postconditions are always omitted if preconditions are.

Assertions can also be stated as part of a production, which means that the assertion must be true in the predecessor snapshot only if the precondition is true there also. If the assertion fails when the precondition is true, the predecessor snapshot is ill-formed, and all the productions do not hold on the snapshot. This is equivalent to redundantly stating the precondition and assertion in another assertion outside any production, but it is more concise.

3.3. Additional Chapter Structure For Actions

This section describes additional chapter structure used in defining the semantics of user-visible actions. These are:

- Inputs.
- Outputs
- Lifecycle
- Execution semantics
- Additional OCL operations

Inputs

The type of a Pin is always Identity or one of its descendants. However, this is not the clearest way to see the inputs of an action, so for clarity this section lists the input pins for the action, referred to by the end name of the association connecting to an input pin, and specifies the multiplicity and the type for each input pin. There is also a description that explains the constraint to which the identity must conform. For example, the input pin for WriteAttributeAction must hold an identity that refers to the same as type of the attribute being written. In some cases, getting to a pin from an action requires navigating more than one association. In this case, the entire navigation path is given using the OCL dot operator.

Inherited pins are listed and marked with the modifier *<inherited>*. This section is included even if there are no input pins, with the text “None.” However, it is not included for abstract action classes.

- `inputName : type [multiplicity]` Description of pin goes here, including the type that constrains the value.
- *<inherited>* `inputName : type [multiplicity]` Parent class is given first. Then a description of inherited pin.
- `end1.end2.inputName : type [multiplicity]` Description of pin reached by navigating multiple associations.

Here’s an example from the “Attribute action metamodel” on page 90:

- `value : Identity [1..1]` Value of attribute to add or remove.

Outputs

This subsection lists the output pins using the same format as input pins.

Lifecycle

This subsection shows an informal state diagram showing the states an action execution goes through. This diagram is merely to help understanding and does not represent a formal state machine. The transitions are expressed in English. Usually each transition corresponds to a production. The precise description of the execution semantics is given in the execution semantics subsection. A lifecycle diagram is not shown if it is the same as the generic one shown in Figure 7, but the section will have a statement “The lifecycle is the same as the generic lifecycle”.

Execution Semantics

This subsection specifies the precise execution semantics for action execution snapshot corresponding to the action class. The semantics are written as a list of assertions and productions expressed in English and OCL. Semantics defined by a class applies all its descendants. The section begins with a declaration of the type of self, which is always a descendant of ActionExecutionSnapshot:

`context self : classname`

This class is not usually defined in the model because there is nothing more to say about it than it is the execution snapshot of the action class being described.

When the assertions appear before the productions, they apply to all the productions. When they are inside a production, they appear after the precondition of the production containing them.

There is no priority among productions. If more than one precondition is satisfied, any valid precondition may apply, the postconditions of which may or may not invalidate other preconditions.

Assertion: English description of the assertion goes here.

OCL description of the assertion goes here.

Now appears the list of productions with entries for an English description, the precondition, assertion if any, and the postcondition. Each production corresponds to a transition on the lifecycle model. The lifecycle diagram is informal, but the production must be precise.

Production 1: English description of the preconditions and postconditions goes here. The terms *predecessor snapshot* and *successor snapshot* may be used to refer to action execution snapshots being constrained by the production, and related object snapshots.

Precondition:

OCL description of the precondition. Usually includes an action execution status as the first “and” clause.

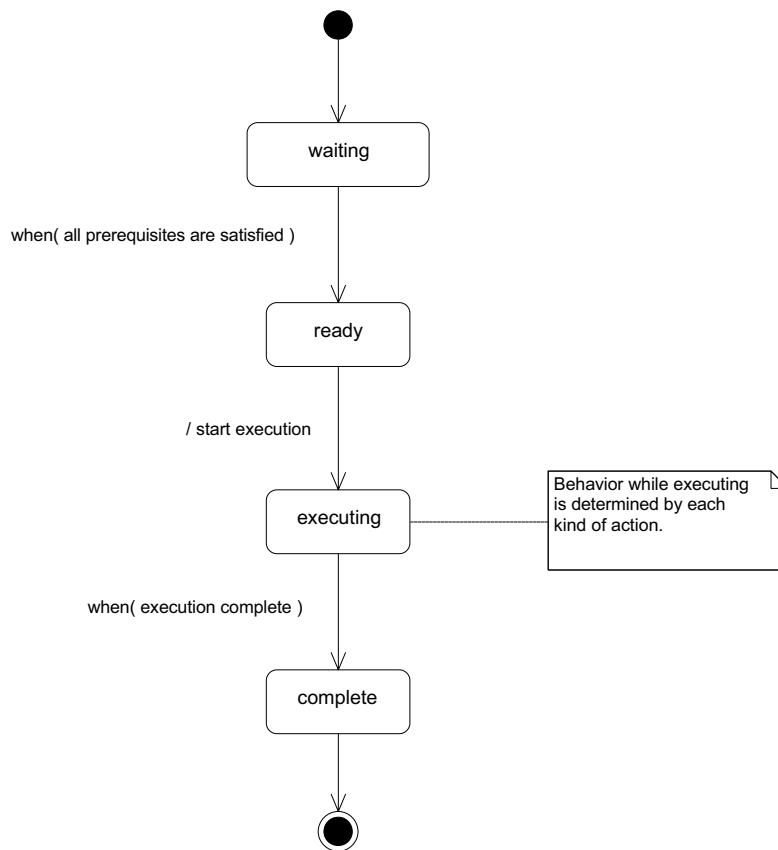


Figure 7. Generic life cycle for action execution

Assertion:

Include an assertion, if required.

Postcondition:

OCL description of the postcondition. Usually includes an action execution status as the first “and” clause.

Some conventions regarding the OCL specifications of execution semantics are:

- The variable *self* in preconditions refers to the predecessor snapshot of the action execution.
- The variable *self* in postconditions refers to the successor snapshot of the action execution.
- All navigations begin from *self*.
- All attributes or associations traversed must appear in the action metamodel or execution model.
- The OCL dot operator and OCL functions are evaluated in the predecessor- or successor-snapshot, depending on which condition evaluated them. These usually operate on identities, in which case they “dereference” the iden-

tity to one of its snapshots that is applicable in the pre/postcondition. See earlier discussion under “Time Ordering and Causality” on page 21.

- Postconditions may use the OCL operator *@pre*, postfixed to object features that should be accessed in the predecessor snapshot. Applying *@pre* on an OCL function means that it is evaluated completely in the predecessor snapshot—all the dot operators navigations in the OCL function are interpreted in the predecessor snapshot.
- If an identity is not mentioned in the postcondition, it is not changed by the production.
- The “let” statement for variable declaration or additional OCL operations may be used to avoid repetition.
- Additional OCL operations are defined in a separate section after the execution semantics.

Additional OCL Operations

Additional OCL operations that apply to the class or its execution snapshots are defined here. These apply to all the descendants of the class also. Naming conventions follow the conventions of the UML 1.4 specification. Operation names are not be prefixed with “get”. This section is omitted if there are no additional OCL operations.

3.4. Examples

Here are the class definitions for `CreateObjectAction` and `VariableAction` defined in Chapter 8, “Read and Write Actions”.

CreateObjectAction

This class represents the instantiation of a classifier.

Identity for a new object is created, storage for the object of the given classifier is allocated, and the classifiers of the object are set to the given classifiers. The new object is returned as the value of the action. The action has no other effect. In particular, no constructors are executed, no initial expressions are evaluated, and no state machines transitions are triggered.

No semantics is defined for creating objects from association classes or abstract classifiers.

Associations

- `class : Classifier [1..1]` Classifier to be instantiated.
- `<derived> result : OutputPin [1..1]` Derived from `Action:outputPin`. Gives the output pin on which the result is put.

Inputs

None.

Outputs

- `result : ObjectIdentity [1..1]` The created object. The type of identity must be the classifier specified for the action.

Well-formedness rules

- [1] The classifier cannot be abstract.
not `(self.classifier.isAbstract = true)`
- [2] The classifier cannot be an association class
not `self.classifier.oclIsKindOf(AssociationClass)`
- [3] The classifier of the result pin must be the same as the classifier of the action.

```
self.result.type = self.classifier
```

[4] The multiplicity of the output pin is 1..1.

```
self.result.multiplicity.is(1,1)
```

Lifecycle

The lifecycle is the same as the generic lifecycle.

Execution Semantics

Production 1: The result has exactly one classifier in the successor snapshot, which is the same as the classifier specified as a parameter of the action. The result has no attribute values in the successor snapshot and does not participate in any associations.

Precondition:

```
self.status = ready
```

Postcondition:

```
let newidentity : ObjectIdentity = self.pinValue→select(pin = self.executionID.action.result).value in
    self.status = completed
    and newidentity.new()
    and newidentity.classifier→size() = 1
    and newidentity.classifier→includes(self.executionID.action.classifier)
    and newidentity.linkEndValue→size() = 0
    and newidentity.attributeValue→size() = 0
```

VariableAction (abstract)

This class provides associations and well-formedness rules for all variable actions.

Associations

- variable : Variable [1..1] Variable being accessed.

Well-formedness rules

[1] The action must be in the scope of the variable.

```
self.variable.isAccessibleBy(self)
```

Lifecycle

The lifecycle is specified by subclasses of this action.

Execution Semantics

The semantics is specified by subclasses of this action.

Additional OCL operations

getVariableValues operates on *VariableAction* execution snapshots. It returns the values of the variables in the context of the action execution corresponding to the variable of the action, in the snapshot in which the operation is called.

```
context WriteVariable::getVariableValues() : Sequence
```

```
post: result = self.context.variableExecution→select(variable = self.executionID.action.variable).value
```

isVariableValueExist operates on *VariableAction* execution snapshots. It takes flag indicating whether to test order and determines whether the variable value specified by the action execution exists for the snapshot in which the operation is called.

```
context WriteVariableAction::isVariableValueExist (isOrderTest : boolean) : Boolean
```

```
post: result = self.getVariableValues→ includes(self.pinValue→select(pin =  
self.executionID.action.value).value)  
and (not isOrderTest  
or self.pinValue→select(pin = self.executionID.action.insertAt).value )
```


Part II. Execution Semantics

This submission relies on an *execution model*, a description of the behavior over time of each mutable entity in a user's model. The execution model captures the history of each mutable entity as a series of snapshots. Pairs of adjacent snapshots constitute a pre- and post-condition pair for some action. Consequently, we can define the semantics of actions by stating the pre- and post-conditions in terms of snapshots.

The execution model comprises two parts. The first describes the conceptual entities required to capture snapshots of mutable entities. The second describes the application of this model to state charts.

Chapter 4. Execution Fundamentals

This chapter focuses on the fundamental representation in the execution model of the identities and snapshots of instances and links. Each of the subsections in this section provides an overview discussion of a portion of this model together with a class diagram for the portion. All classes are fully described together in a single alphabetical list in Section 4.2.

Chapter 5, “State Machine Execution”, focuses on the model for state machine execution.

4.1. Instances and Links

Data Values

We distinguish between *instance identities* and *link identities*, because it is the instance identities that are the values of attributes and association ends. We further distinguish *data-value identities* from *object identities*, because a data value is immutably associated with a single data type, while an object can have multiple classifiers that can change over time (allowing for multiple and dynamic classification). Thus, the data type of a data value can be associated directly with its identity, as shown in Figure 8, while the classifiers of an object must be associated with its snapshot. As data values have no mutable properties, they have no snapshots at all and are represented solely by their immutable identity.

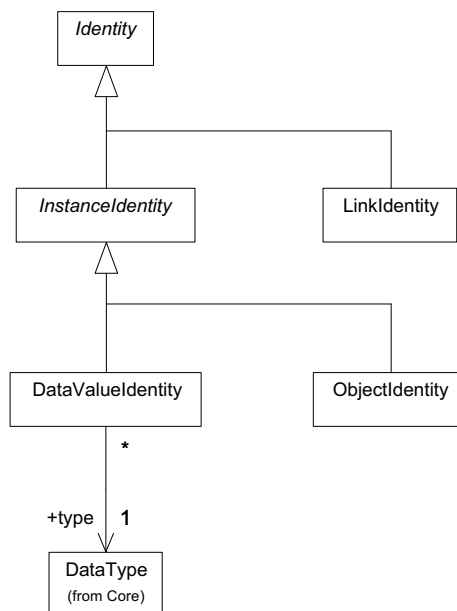


Figure 8. Data-value identity

Objects

An object has properties that change over time; it is a *mutable entity*. The values of the properties at any time are given by the snapshot for the object effective at that local time of the history of that object. The mutable properties of an object associated with its snapshot include the following.

- The set of classifiers of the object.
- Values for the attributes specified by all of these classifiers.
- Values for the (navigable) opposite link ends of associations in which the classifiers participate.

We do not include snapshot specializations for other types of mutable instances, such as instances of nodes, components and use cases. The execution model here is rich enough to model these other mutable instances, once restricted appropriately (e.g., a component instance must have a single component as its classifier throughout its history).

This concept of an object snapshot is formalized in Figure 9.

Links

A link is related to an association in an analogous manner to the relationship of an instance to its classifier(s). The link provides link-end values for the association ends of the association. However, the set of link-end values for a link is part of the very identity of the link, in the sense that a link *is* a tuple of values for the ends of an association, as shown in Figure 10.

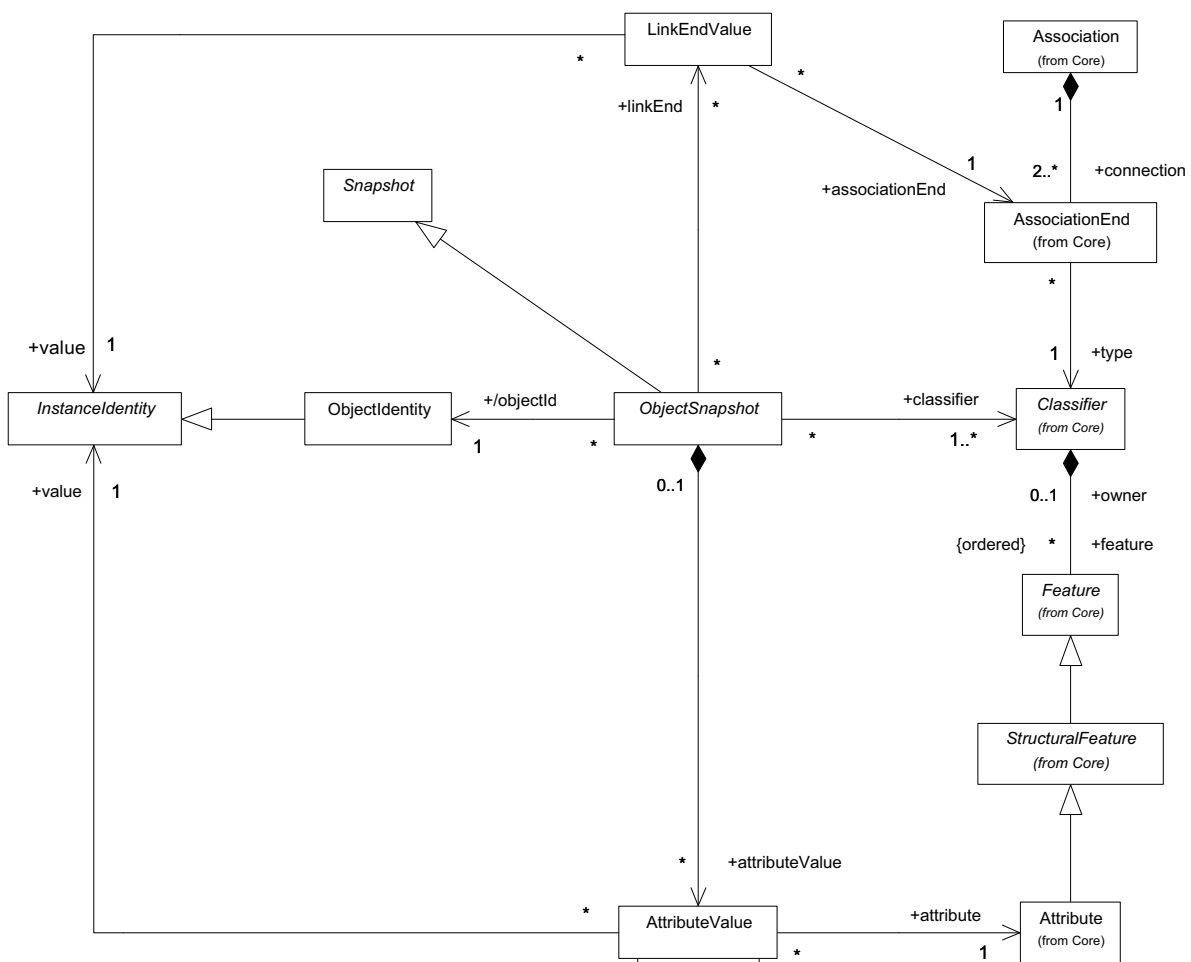


Figure 9. Object execution model

A link also has a snapshot because links are created and destroyed. However, a normal link does not otherwise have properties that change over time, and thus has only a single snapshot in its history between creation and destruction changes. A link object, though, may have object properties and may have any number of snapshots in its history. (see the next section for a discussion of link objects).

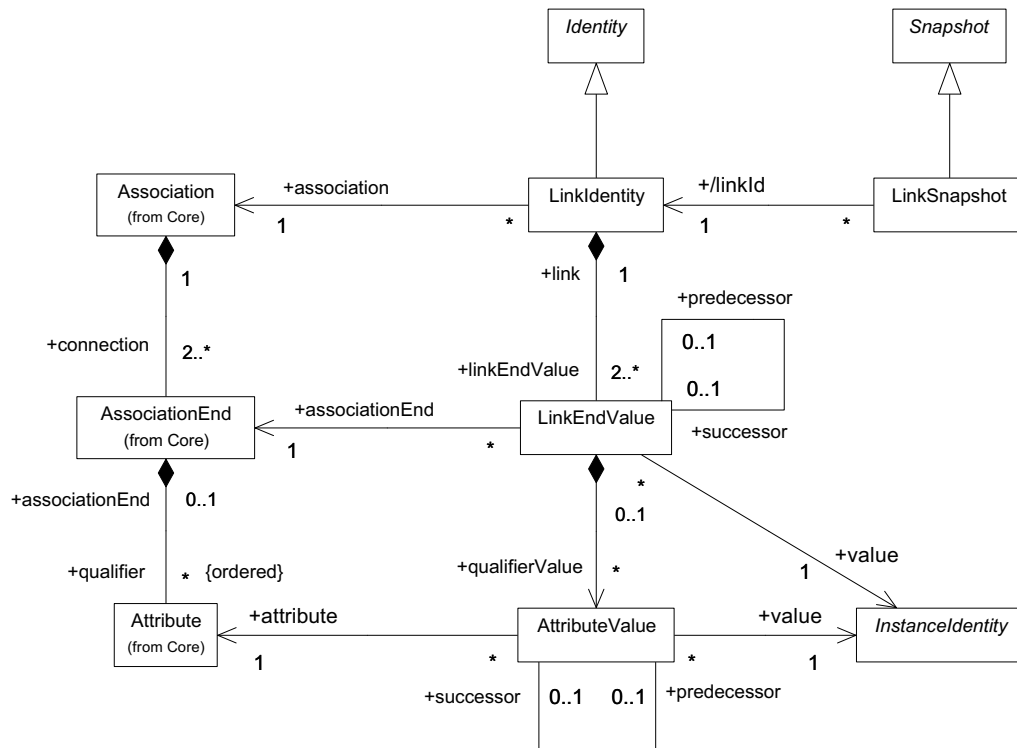


Figure 10. Link execution model

Link Objects

A link object is an instance of an association class and has both the properties of a link and the properties of an object. As shown in Figure 11, link-object identities and snapshots specialize the identities and snapshots from both link and object. Since link snapshots have no properties, the link properties are all associated with the link-object identity. The mutable properties of a link object are inherited by treating the link-object snapshot as an object snapshot.

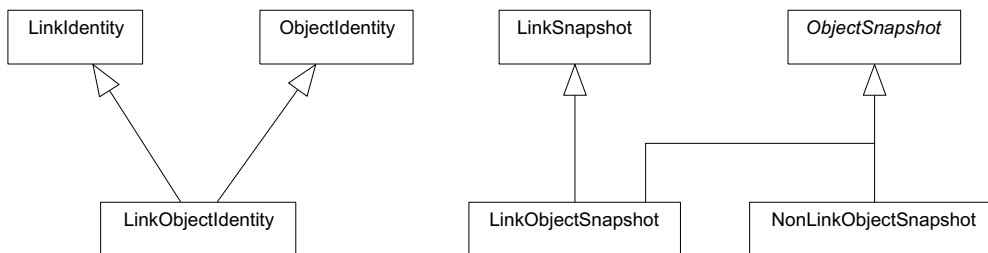


Figure 11. Link object execution model

Extents

The *extent* of a classifier is the set of instances of that classifier that exist at some time. However, since each instance has its own history and there is no global notion of time in the execution model, there is no *a priori* concept of a set of instances “at some point in time”. Nevertheless, it is necessary to model access to the extent of a classifier.

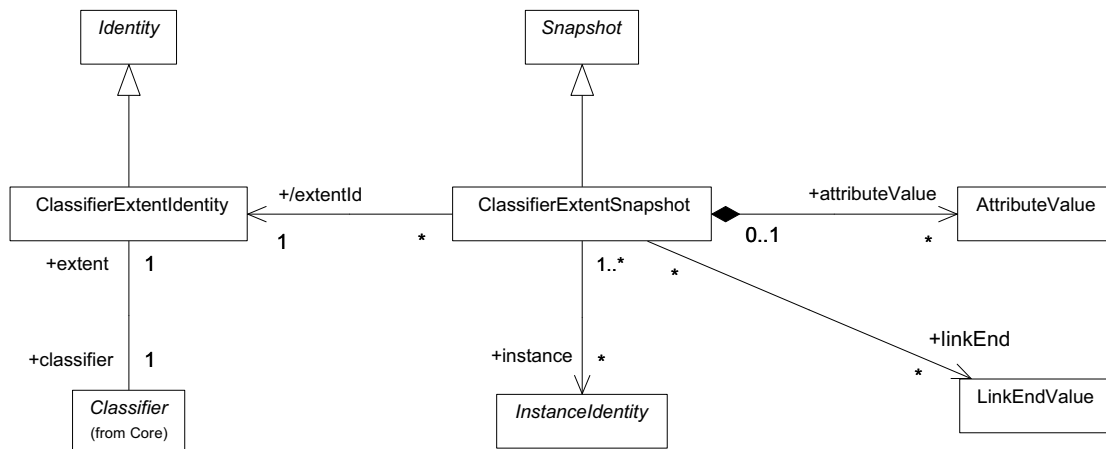


Figure 12. Classifier extent model

We model classifier extents as explicit entities, as shown in Figure 12. Each classifier has an associated classifier-extent identity. The classifier-extent snapshot has as a property the “current” set of instances of the classifier. Whenever an instance of the classifier is created or an instance is dynamically reclassified to the classifier, the identity of the instance is added to the extent. Multiply classified instances belong to more than one extent.

These changes in the classifier-extent history are explicitly caused by creation, destruction or reclassification actions. Further, a read-extent action must have an access synchronization with a specific extent snapshot to read it. Thus the set of instances is the result of a read-extent action is determined dynamically via the extent snapshot.

The classifier-extent snapshot has as properties the values for attributes and link ends for its classifier with classifier-level owner scope. This is modeled in the same way as the attribute and link-end values of an object snapshot.

An association also has an extent, as shown in Figure 13. Each association has an associated association-extent identity. A link identity is added to the extent when the link is created and removed when the link is destroyed. In addition, the cardinalities of the links of an association are always defined relative to the association extent.

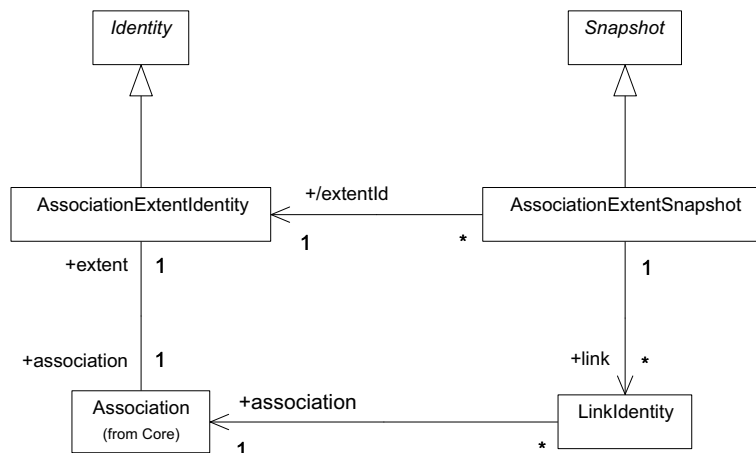


Figure 13. Association extent model

Note. A real execution-engine implementation need not provide one central, universal extent for every classifier and association. It could maintain extents with restricted scope, or even no explicit extents at all.

4.2. Instance and Link Execution Model Classes

For a given identity *id*, there are three related snapshots of interest relative to this snapshot:

- *current(id)* is the snapshot for *id* for which the execution snapshot has a referent link. This is used to get values for identities in preconditions and assertions and for constraints (as opposed to effects) in postconditions. Note that properties of the execution snapshot itself can be retrieved directly.
- *pre(id)* is the snapshot for *id* for which the *predecessor* to the execution snapshot has a referent link. This is used in a postcondition to get the referent relative to the precondition snapshot.
- *post(id)* is the snapshot for *id* that was the result of a change caused by the preceding step that led to the execution snapshot. This is used in a postcondition to get the snapshot that resulted from an effect of the action on another entity, and it also asserts that a change was caused by the last step (unlike “current”, which only asserts an access synchronization).

AttributeValue

An attribute value provides a single value for a specific attribute or association-end qualifier. The value is represented as an instance identity. In the case of an object identity, this effectively represents a *reference* to the object, since it is not dependent on the object snapshot. Note that if the attribute has a multiplicity, then there may be several attribute values for that attribute, but each attribute value only has a *single* associated instance identity. If the attribute is ordered, then the attribute value also has links to the previous and/or next attribute values in the ordering for the values of the attribute.

Note. UML 1.3 does not provide for the ordering of attribute values, but this is planned for UML 1.4.

Associations

<i>attribute</i>	The attribute whose value is given by this attribute value.
<i>predecessor</i>	The attribute value for the same attribute that represents the value, if any, that precedes this attribute value in an ordered attribute.
<i>successor</i>	The attribute value for the same attribute that represents the value, if any, that follows this attribute value in an ordered attribute.
<i>value</i>	The actual instance identity representing a value of the attribute.

Additional operations

- [1] This operation returns the position, as a positive integer, of an attribute value in the sequence of attribute values for an ordered attribute.

context `AttributeValue::at() : Integer`

post: `result = if self.predecessor→isEmpty() then 1 else self.predecessor.at() + 1 endif`

Wellformedness Rules

- [1] *Attribute ordering consistency.* An attribute value may only have a predecessor or successor if its attribute is ordered. If it has a predecessor or successor, then it must be part of a list of attribute values for the same attribute.

`(self.attribute.ordering = #unordered implies self.predecessor→isEmpty() and self.successor→isEmpty())`

and `(self.predecessor→isEmpty()`

or `(self.predecessor<>self and self.predecessor.attribute=self.attribute and self.predecessor.successor=self))`

and `(self.successor→isEmpty()`

or `(self.successor<>self and self.successor.attribute=self.attribute and self.successor.predecessor=self))`

Note. It would seem logical to include here a conformance rule requiring the value of an attribute to have a classifier that conforms to the type of the attribute. However, objects can be dynamically classified, so it is not possible in general to determine their classifier(s) from their identity. Instead, any check for type conformance must be done when reading and writing the value to the attribute.

AssociationExtentIdentity

An association-extent identity is the identity for the association extent for a specific association.

Associations

association The association for which this identity identifies the extent.

AssociationExtentSnapshot

An association-extent snapshot records the extent of an association that is updated as links are created and destroyed.

Associations

extentId (*Derived*) The identity of the association-extent snapshot, as an association-extent identity.
link The set of links that make up the current extent of the association.

Wellformedness Rules

- [1] *Association-extent-identity constraint.* The identity of an association-extent snapshot must be an association-extent identity.

```
self.identity.oclIsKindOf(AssociationExtentIdentity)
and self.extentId = self.identity.oclAsType(AssociationExtentIdentity)
```

- [2] *Association-extent conformance.* All links of an association-extent snapshot must have the same association as the extent.

```
self.link->forAll(association = self.extentId.association)
```

- [3] *Link uniqueness.* No two links in an association-extent snapshot can have identical values for all link-ends and qualifiers.

```
self.link->forAll(l1, l2 : LinkIdentity | l1.value = l2.value and l1.qualifierValue = l2.qualifierValue
implies l1 = l2)
```

- [4] *Association-multiplicity conformance.* The set of links of an association-extent snapshot must conform to the multiplicity of each of the association ends of the association of the extent.

Note. OCL to be done.

Change

A change is a relationship between two successive snapshots in the history of a mutable entity (except for the first and last changes in a history, which represent entity creation and destruction). Note that snapshots do not “change over time” as such. Rather, there is a *separate* snapshot for interval of time at which the entity has a different set of properties, connected by changes that occur at specific points in time.

Attributes

time The “local” time at which this change occurs.

Associations

cause The step caused this change.

history The history of which this change is a part.

predecessor The snapshot representing entity properties before this change.

successor The snapshot representing entity properties after this change.

Wellformedness Rules

- [1] *Change-snapshot consistency.* If a change has both predecessor and successor snapshots, then they must have the same identity.

`self.predecessor→notEmpty()` and `self.successor→notEmpty()` implies
`self.predecessor.identity = self.successor.identity`

- [2] *Change-history consistency.* A change must have a predecessor or a successor (or both) and must be part of a single list of changes in the same history.

`(self.predecessor→notEmpty() or self.successor→notEmpty())`
and `(self.predecessor→empty or self.predecessor.successor = self and self.predecessor.history = self.history)`
and `(self.successor→empty or self.successor.predecessor = self and self.successor.history = self.history)`

- [3] *Time-ordering consistency.* A change cannot have a time that is earlier than the time of its predecessor.

`self.predecessor→isEmpty()` or `self.predecessor.time <= self.time`

Note. The above rule allows “instantaneous” snapshots with pre- and post-changes at the same time. This allows multiple causes to effect an entity “at the same time”. Nevertheless, even snapshots with the same time value are still totally ordered within a history.

ClassifierExtentIdentity

A classifier-extent identity is the identity for the classifier extent for a specific classifier.

Associations

classifier The classifier for which this identity identifies the extent.

ClassifierExtentSnapshot

An classifier-extent snapshot provides a record of the extent of an classifier that is updated as instances are created, destroyed and reclassified. The classifier-extent snapshot also maintains the values of the attributes and link ends of the classifier that have classifier-level owner scope.

Associations

attributeValue The set of attribute values for classifier-scope attributes.

extentId (*Derived*) The identity of the classifier-extent snapshot, as an classifier-extent identity.

instance The set of instances that make up the current extent of the classifier.

linkEnd The set of values for the opposite link ends of navigable association ends of classifier-scope associations.

Wellformedness Rules

- [1] *Classifier-extent-identity constraint.* The identity of an classifier-extent snapshot must be an classifier-extent identity.

```
self.identity.oclIsKindOf(AssociationExtentIdentity)
and self.extentId=self.identity.oclAsType(ClassifierExtentIdentity)
```

Note. Other well-formedness rules to be done.

DataValueIdentity

In UML, data values represent “immutable” instances whose properties do not change over time. Indeed, the properties of a data value are implicit in the identity of the data value itself, so data values do not have snapshots or histories at all. A data value can have only one classifier, which must be a data type.

Associations

type The data type of the value represented by this data-value identity.

ExecutionSnapshot

The execution of some behavioral element generally progresses through a number of stages over time. An execution is considered an entity in its own right, with its own identity and history. An execution snapshot represents a stable stage between steps in an execution. During the interval represented by the execution snapshot, the execution may access properties of other snapshots. The association between an execution snapshot and accessed snapshots represents synchronization points between the history of the execution and the histories of the accessed entities.

Associations

referent The set of snapshots of other entities accessed by this execution, synchronized with this execution snapshot.

Additional operations

- [1] This operation returns the current referent snapshot for the given identity.
 context ExecutionSnapshot::current(id : Identity) : Snapshot
 post: referent→includes(result) and result.identity = id
- [2] This operation returns the referent snapshot for the given identity relative to the preceding execution snapshot.
 context ExecutionSnapshot::pre(id : Identity) : Snapshot
 pre: not self.isNew()
 post: result = self.predecessor().oclAsType(ExecutionSnapshot).current(id)
- [3] This operation returns the snapshot for the given identity that is the result of a change caused by the step preceding this execution snapshot.
 context ExecutionSnapshot::post(id : Identity) : Snapshot
 post: self.preChange.oclAsType(Step).effect.successor→includes(result) and result.identity = id
- [4] This operation asserts that the step preceding this execution snapshot caused the creation of the entity with given identity.
 context ExecutionSnapshot::new(id : Identity) : Boolean
 post: result = post(id).isNew()
- [5] This operation asserts that the step preceding this execution snapshot caused the destruction of the entity with given identity.
 context ExecutionSnapshot::destroyed(id : Identity) : Boolean
 post: result = self.preChange.oclAsType(Step).effect
 →exists(c : Change | c.successor→isEmpty() and c.predecessor.identity = id)

Wellformedness Rules

- [1] *Execution-step constraint.* The pre- and post-changes of an execution snapshot are both steps.
`self.preChange.ocIsKindOf(Step)` and `self.postChange.ocIsKindOf(Step)`

History

The history of an entity records the sequence of changes that occur over the life of that entity. Between these changes are snapshots of the properties of the entity that are valid during the time intervals between changes. Since a history is for a single entity, all these snapshots must have the same identity.

Associations

change The set of changes that make up this history.

Wellformedness Rules

- [1] *History consistency.* A history has one creation change and one destruction change.

`self.change→select(predecessor→isEmpty())→size() = 1`
`and self.change→select(successor→isEmpty())→size() = 1`

Identity

The identity of an entity is immutable, timeless and unique from all other identities.

InstanceIdentity

An instance identity is the identity of some instance of a classifier.

LinkEndValue

A link-end value provides a single participant value for a specific end of a link. The value is represented as an instance identity. In the case of an object identity, this effectively represents a *reference* to the object, because it is not dependent on the object snapshot.

If the specified association end has qualifiers, then the link-end value must also include attribute values for each of the qualifiers. If the association end is ordered, then the link-end value also has links to the previous and/or next link-end values in the ordering for that end of the link snapshot.

Note. The instance-level specification of qualifier values is incorrect in the UML 1.3 specification. The association “qualifierValue” is mentioned in the description of LinkEnd, but it is not shown on Figure 2-16 (Common Behavior - Instances and Links) or covered in the well-formedness rules.

Associations

<i>associationEnd</i>	The association end whose value in a link is represented by this link-end value.
<i>link</i>	The identity of the link containing this link-end value.
<i>predecessor</i>	The link-end value for the same association end in a different link for the same association that represents the value, if any, that precedes this link-end value in an ordered association end.
<i>qualifierValue</i>	The set of values for qualifiers of the association end.
<i>successor</i>	The link-end value for the same association end in a different link for the same association that represents the value, if any, that follows this link-end value in an ordered association end.
<i>value</i>	The actual instance-identity that is the participant value of this link end.

Additional operations

- [1] This operation returns the position, as a positive integer, of a link-end value in the sequence of link-end values for an ordered association end.

```
context LinkEndValue::at() : Integer
```

```
post: result = if self.predecessor→isEmpty() then 1 else self.predecessor.at() + 1 endif
```

Wellformedness Rules

- [1] *Association-end ordering consistency.* A link-end value may only have a predecessor or successor if its association end is ordered. If it does have a predecessor or successor, then it must be a part of a linked list of link-end values for the same association end but different links.

```
(self.associationEnd.ordering = #unordered implies self.predecessor→isEmpty() and
```

```
self.successor→isEmpty())
```

```
and (self.predecessor→isEmpty() or self.predecessor.successor=self)
```

```
and (self.successor→isEmpty() or self.successor.predecessor=self)
```

```
and self.predecessor→union(self.successor)→forAll(associationEnd=self.associationEnd and link<>self.link)
```

- [2] *Qualifier conformance.* For each qualifier of the association end of a link-end value, the link-end value must have a number of attribute values for that qualifier allowed by the multiplicity of the qualifier and the instance-identity values of those attribute values must all be distinct.

```
self.associationEnd.qualifier→forAll(q : Attribute |
```

```
let values = self.qualifierValue→select(attribute = q) in
```

```
q.multiplicity.allows(values→size()) and values→isUnique(value))
```

LinkIdentity

The identity of a link is determined by the association of the link and by the values the link gives to the association ends of the association. Therefore, a link identity is associated with both an association and a set of link-end values, reflecting the immutability of a link. A link has *exactly one* link-end value for each association end of its association.

Associations

association The association for the link represented by this link identity.

linkEndValue The set of values of the ends of the link corresponding to the association ends of the association.

Wellformedness Rules

- [1] *Association-end conformance.* Each link-end value of a link identity must be for a distinct association end of the association of the link identity and each non-optional association end (i.e., one with a multiplicity lower bound greater than zero) must have a corresponding link-end value.

```
self.linkEndValue.associationEnd→isUnique(ae: AssociationEnd | ae)
```

```
and self.linkEndValue.associationEnd = self.association.connection
```

LinkObjectIdentity

A link object is an instance of an association class, which is both an association and a class. As such, a link-object identity is both a link identity and an object identity.

LinkObjectSnapshot

A link object is an instance of an association class, which is both an association and a class. As such, a link-object snapshot is both a link snapshot and an object snapshot.

Wellformedness Rules

- [1] *Link-object-identity constraint.* The identity of a link-object snapshot must be a link-object identity.

```
self.identity.oclIsKindOf(LinkObjectIdentity)
```

- [2] *Association-class conformance.* Exactly one of the classifiers of a link-object snapshot must be an association class that is the same as the association related to its identity.

```
self.linkId.association.oclIsKindOf(AssociationClass)
and self.classifier→count(self.linkId.association.oclAsType(AssociationClass)) = 1
```

Note. Since the association of a link object is fixed here via its link-object identity, it is not possible for the link object to be reclassified from one kind of association to another or for a non-link object to be reclassified as a link object.

LinkSnapshot

Since a link (if it is not a link object) has no mutable properties, there are none defined on a snapshot. However, a link is created and destroyed, and so the snapshot represents the duration of its existence in time (as just a single snapshot, in the case of a link that is not a link object).

Associations

linkId (Derived) The identity of the link snapshot, as a link identity.

Wellformedness Rules

- [1] *Link-identity constraint.* The identity of a link snapshot must be a link identity.
 self.identity.oclIsKindOf(LinkIdentity) and self.linkId = self.identity.oclAsType(LinkIdentity)

ObjectIdentity

An object identity represents the identity of an object or any other mutable instance.

ObjectSnapshot

An object snapshot represents a snapshot of the properties of an object or any other mutable instance. These properties include the classifiers of the object, the values of the attributes of the object and the values of the opposite link ends of the object.

Associations

attributeValue The set of values for attributes of the classifiers of the object snapshot.

classifier The set of classifiers of the object represented by the object snapshot.

linkEnd The set of values for the opposite link ends of navigable association ends of associations in which classifiers of the object snapshot are involved.

objectId (Derived) The identity of the object snapshot, as an object identity.

Wellformedness Rules

- [1] *Object-identity constraint.* The identity of an object snapshot must be an object identity.
 self.identity.oclIsKindOf(ObjectIdentity) and self.objectId = self.identity.oclAsType(ObjectIdentity)
- [2] *Attribute conformance.* For each attribute owned by each classifier of an object snapshot, there must be a number of attribute values conforming to the multiplicity of the attribute and the instance-identity value of these attribute values must all be distinct. If an attribute is ordered, then all the attribute values for that attribute must be linked together in a single predecessor/successor list.
 self.classifier.allAttributes→forAll(a : Attribute |
 let values = self.attributeValue→select(attribute = a) in
 a.multiplicity.allows(values→size()) and values→isUnique(value)
 and (a.ordering = #ordered implies
 values→select(predecessor→isEmpty())→size()<=1
 and values→select(successor→isEmpty())→size()<=1))

- Given the above, the attribute ordering rule on Attribute implies the values
- must be in a single linked list

- [3] *Association conformance.* For each association in which a classifier of an object snapshot is involved, for each navigable opposite association end of the association, for each distinct set of values for qualifiers of that association end (if any) there must be a number of link-end values consistent with the multiplicity of that association end and the instance-identity values of those link-end values must be distinct.

```
self.classifier.allAssociations→
  forAll(a:Association|a.connection→
    forAll(ae:AssociationEnd|ae.allOppositeAssociationEnds→
      select(ae2:AssociationEnd|ae2.isNavigable)→
        forAll(ae2:AssociationEnd
          let S = self.linkEndValue→select(lev:LinkEndValue|lev.associationEnd = ae2) in
            ae2.multiplicity.allows(S→size()) and S→isUnique(value)
```

Note. Operation Classifier::allAttributes is defined in the UML 1.3 specification.

Snapshot

A mutable entity (such as an object, as opposed to a data value) will have a unique identity over its entire lifetime, but its properties may change over time. A snapshot is a record of a complete set of properties for the entity that remain unchanged over the interval of time between two changes.

Associations

<i>accessor</i>	The set of execution snapshots that access this snapshot.
<i>identity</i>	The identity of the entity whose values are represented by this snapshot.
<i>postChange</i>	The snapshot resulting from this change.
<i>preChange</i>	The snapshot effected by this change.

Additional operations

- [1] This operation returns the set of predecessors of this snapshot (which will always be a singleton or the empty set).
 context Snapshot::predecessor() : Set(Snapshot)
 post: result = self.preChange.predecessor
- [2] This operation returns the set of successors of this snapshot (which will always be a singleton or the empty set).
 context Snapshot::successor() : Set(Snapshot)
 post: result = self.postChange.successor
- [3] This operation asserts that this is the first snapshot in its history.
 context Snapshot::isNew() : Boolean
 post: result = self.predecessor()→isEmpty()
- [4] This operation asserts that this is last snapshot in its history.
 context Snapshot::terminates() : Boolean
 post: result = self.successor()→isEmpty()

Step

A step is a change in the history of an execution.

Associations

<i>effect</i>	The set of changes caused by this change.
---------------	---

Wellformedness Rules

- [1] *Step-snapshot consistency.* The predecessor and successor snapshots of a step must be execution snapshots.
 self.predecessor.ocIsKindOf(ExecutionSnapshot) and self.successor.ocIsKindOf(ExecutionSnapshot)

Chapter 5. State Machine Execution

The behavior of a method is specified by a single *procedure* that acts as the body of that method. A procedure is a unit of behavior that accepts as input the values of the in and in-out parameters of the method and produces as output values for the in-out, out and return parameters. In the course of its execution, a procedure may also cause changes to object memory, through the creation of new instances and links and changes to attribute values.

A single execution of a state machine may trigger the execution of several procedures. The occurrence of an event on a transition of a state machine may trigger behavior associated with that transition as well as behavior associated with state entry and exit caused by that transition. Each of these behaviors is specified by a procedure.

This chapter focuses on the semantics of state machine execution as a context for the triggering of procedure execution. The discussion here does not completely formalize the definition of state-machine semantics, which is beyond the scope of the action semantics defined in this submission, but relies on the semantic definition given in the UML specification. Once the syntactic matching of parameters is defined, the semantics of method execution is given fully by procedure execution, so method execution is not considered further here.

The formal definition of procedures and their execution semantics is given in Chapter 6, “Action Foundation”.

Note. The concept of a *procedure* is part of the new action metamodel. The UML 1.4 metamodel is modified to replace the Action class in State Machines with the Procedure class, as well as providing an association from Method to Procedure to define method bodies. See Appendix A, “Updates to UML”.

5.1. State Machine Execution Model

The UML specification allows a state machine to be used as the behavioral specification for either a classifier or a behavioral feature. We consider here the use of state machines for the specification of classifier behavior only. The behavior of behavioral features (methods, specifically) is given by a single procedure. Only mutable objects may have an executable state-machine behavior specification, since the “current state” of the state machine is effectively a mutable property of the mutable instance.

Note. It is also possible in UML to use state machines (most commonly activity graphs) to specify the behavior of an operation. The relationship of this approach to the action semantics is still to be worked out.

An *active* object is an object one of whose classifiers is a class with the “isActive” attribute set to true. Such an object runs concurrently with other active objects. This allows it to accept asynchronous signals and otherwise respond to the occurrence of incoming events. There are potentially a number of ways to specify the behavior of an active object. The approach described here is to use a state machine attached to the active-object class whose execution provides the concurrent behavior of the object.

A *passive* object is one that is not active. All behavior of a passive object must be triggered by synchronous calls to methods of the object. Ultimately, any chain of such calls must originate in some active object so all activity in a system is ultimately traceable to active objects.

Note. Passive-object classes may also have state machines. A “protocol state machine” can be used to specify the legal ordering of calls to methods of a passive object (Section 2.12.5 of the UML 1.3 specification). Invitations are not covered in this discussion yet.

The continued execution of a state machine is governed by its current *state configuration*, the set of all currently active states in the machine, including all nested and concurrent states (as defined in Section 2.12.4 of the UML 1.3 specification). The state machine responds to the *occurrence* of an event by triggering matching transitions leaving one or more of these active states, possibly executing associated procedures resulting in a new state configuration. The occurrence that triggers such a *step* in the state machine is the *current occurrence* during the execution of the

step. Signal events and call events have arguments (the attributes of the signal or the parameters of the operation), whose values for the current occurrence are available to the procedures executed in response to that occurrence.

Note. The UML specification uses the term *current event*, but the term *current occurrence* clearly separates the specification of an event from its occurrence.

During a state-machine step, the execution of the state machine is insensitive to new occurrences. This is known as the *run-to-completion* property of state machines. Nevertheless, events may occur at any time. Therefore, it is necessary during the execution of a state machine to keep track of occurrences of state-machine events that have not yet been handled.

Note. Interrupts, such as kills, supersede run to completion. These are not dealt with yet in this submission. “Synch states” are also not yet addressed.

Figure 14 shows the execution model for the basic state-machine concepts discussed above. As shown, each execution of a state machine has its own unique identity. Following the general approach described in Chapter 4, “Execution Fundamentals”, the properties of such an execution at any time are modeled by a *state-machine execution snapshot*. The dynamic semantics of a state machine is then formalized in terms of histories of state-machine execution snapshots. .

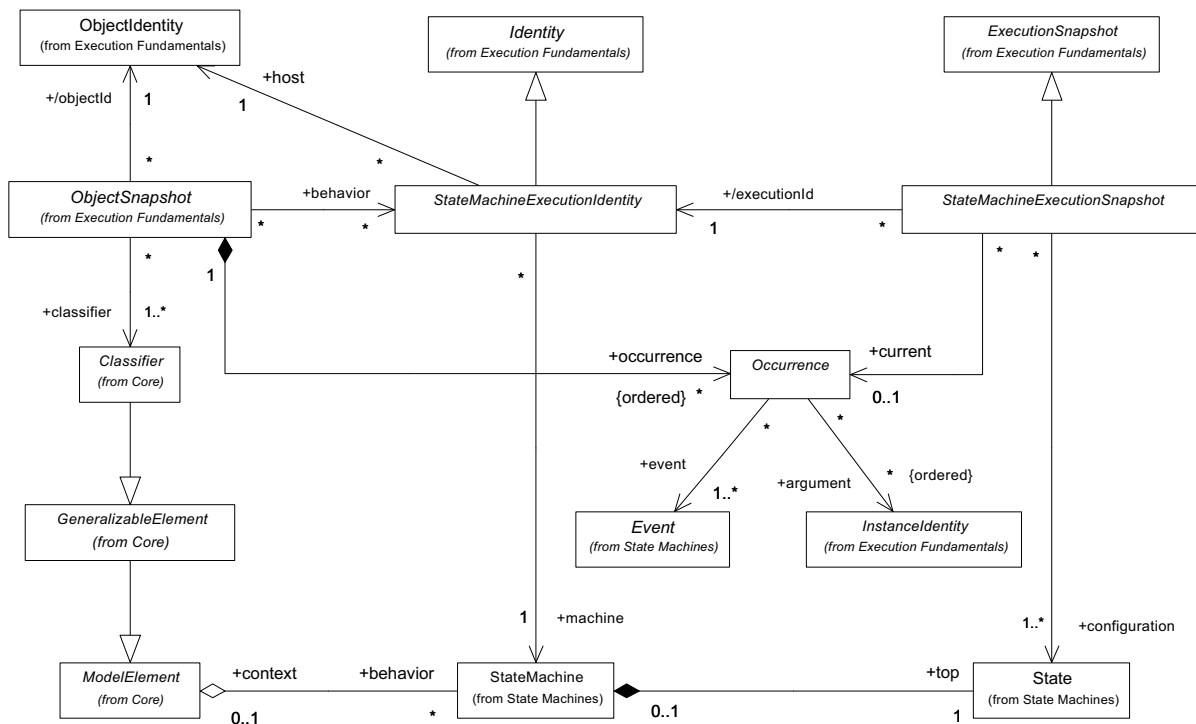


Figure 14. State machine execution model

The active object snapshot maintains the set of unhandled occurrences. When an occurrence of interest to an active object happens, it causes a change in the history of the object, resulting in an object snapshot with a set of unhandled occurrences. This does not affect the history of the state machine execution. Rather, a step in the execution of the state machine is always due to the *handling* of an occurrence dispatched to the active object.

A state-machine execution history *conforms* to the state machine if every step in the history is legal according to the state-machine specification. The current occurrence of a legal step (that is, the current occurrence of the starting

snapshot of the step) is the one dispatched from the active object. The state machine specification then determines the procedures triggered by the occurrence and the state configuration of the ending snapshot of the step.

Note. We do not yet dealt with deferred and completion events.

This is an essentially local view of the execution of a state machine. Each succeeding snapshot is at a later time than the preceding snapshot in the same state-machine execution history. This provides a *local frame of reference* for time within one state machine. However, there is no assumption of synchronous operation of the entire system, nor indeed any assumption that there even exists a single time across the entire system. There is no assumption of centrality or globality. There is no central repository, queue, control, timer, arbiter, etc.

Note. A centralized approach would be a permissible optimization for an execution engine for state machines, but is not assumed in the formal execution model.

State machines interact with each other via signals or calls between their corresponding active objects. Causal chains of such interactions may result in sequencing constraints in the execution of the state machines. For example, if one state machine sends a signal to the another, triggering the second state machine to send a signal back to the first, then the reception of the second signal of necessity must occur after the time the first signal is sent. But any actions that are not causally connected may proceed concurrently and even at different rates.

5.2. Run-to-Completion Step

A single run-to-completion step corresponds to moving from one state-machine-execution snapshot to the next in the history of the state machine. Given a current occurrence that triggers a step, the UML 1.3 specification defines the set of transitions in the state machine enabled by the occurrence. Each of these (possibly compound) transitions may fire, resulting in the execution of a set of procedures, with the procedures for one transition executing concurrently with the procedures from other transitions.

For each enabled transition that is not an internal transition, the following kinds of procedures may be executed:

- *Guards.* A guard is specified by a procedure that has arguments that conform, in order, to the arguments (if any) of the triggering event and a single Boolean result. Guards on a transition up to the first choice vertex (if any) are executed concurrently when the transition is enabled. An enabled transition fires if all these guards result in “true”. Guards on a transition after a choice vertex are evaluated only after all procedures on the transition up that choice vertex have been executed.
- *Exit procedures.* If the transition fires, then some number of states are exited. An exited state may have an exit procedure. An exit procedure may have an associated signal event (this is a change to UML 1.4), in which case the state machine is ill-formed if the state can be exited by anything but an occurrence of a signal event with a signal that matches the signal of the exit procedure or is a descendant of that signal. The procedure must have arguments that conform, in order, to the arguments of the triggering event. An exit procedure without an associated signal event is always legal, but it may not have any arguments. In all cases, an exit procedure has no results. Exit procedures along a single transition segment are executed sequentially from the most nested state outwards. If a compound transition has multiple initial segments (such as a set of segments into a join), then the set of exit procedures from any one segment executes concurrently with the set of exit procedures from other segments.
- *Transition procedures.* Various segments of a compound transition may have attached procedures that are executed when that segment is traversed. A transition procedure has arguments that conform, in order, to the arguments (if any) of the triggering event and no results. Transition procedures are executed in order along the path traversed through the transition. (If the transition includes choice vertices, then the actual path will be determined dynamically and the execution of transition procedures will be interleaved with the execution of guard procedures at the choice vertices.)
- *Entry procedures.* Once the transition has been traversed, some number of states are entered, including states that are entered due to initialization transitions of newly entered composite states. An entered state may have an procedure. Entry procedures have the same form as exit procedures, with the same event matching rules. Entry pro-

cedures along a single transition segment are executed sequentially from the least nested state inwards. If a compound transition has multiple final segments (such as a set of segments out of a fork), then the set of entry procedures from any one segment executes concurrently with the set of entry procedures from other segments.

An internal transition never causes any states to be exited or entered, but always has a transition procedure, and it may have a guard. If the internal transition is enabled, its guard is evaluated and, if the guard is true (or absent), the transition fires, executing its transition procedure.

The run-to-completion step ends when all the procedures associated with all firing transitions have completed execution and a new state configuration has been established for the state machine.

5.3. Occurrences

The UML specification defines four kinds of events: change events, time events, signal events and call events. The semantics of call events is not defined by this submission. There are three kinds of occurrences each corresponding to a kind of event.

Note. Invite occurrences have not yet been incorporated into the occurrence model.

- *Change occurrence.* A change event occurs when a given boolean condition becomes true. A single change occurrence may trigger multiple events, all of whose boolean conditions of necessity come true together. There must be an event in the set of events triggered by a change occurrence such that the boolean condition of that event logically implies the boolean condition of each other event in the set. There must be no other change events in the state machine that are implied by events in the set but are not in the set themselves. A change occurrence has no arguments.
- *Time occurrence.* A time event occurs when its given deadline is reached. A single time occurrence may trigger multiple events. A time occurrence has no arguments.

Note. The above description does not deal with how condition changes and time deadlines are actually detected, which is part of the dynamic semantics of state machines as defined in UML. Rather, the model here simply defines how such occurrences, once detected, are recorded in order to trigger the behavior of a given state machine

- *Signal occurrence.* A signal occurrence records the *reception* of a signal by a state machine. (The sending of signals is discussed in Chapter 11, “Messaging Actions”.) A signal is a classifier, so an instance of a signal is represented by an object snapshot. The values of the attributes of the signal instance provide the arguments of the signal occurrence. A signal occurrence triggers all events in a state machine that specify the received signal or any ancestor of the received signal.

Note. The signal occurrence and the signal instance may have different times. The time of the signal instance is the time that the signal was sent. The time of the signal occurrence is the time of the *reception* of the signal, which may be later due to transmission delays. And it may be later still before the signal occurrence actually triggers a response from the state machine, if ever. Note also that there may be several (or no) signal occurrences for a single signal instance. This allows the modeling of broadcast (multicast) signals and signals that are sent but never actually received.

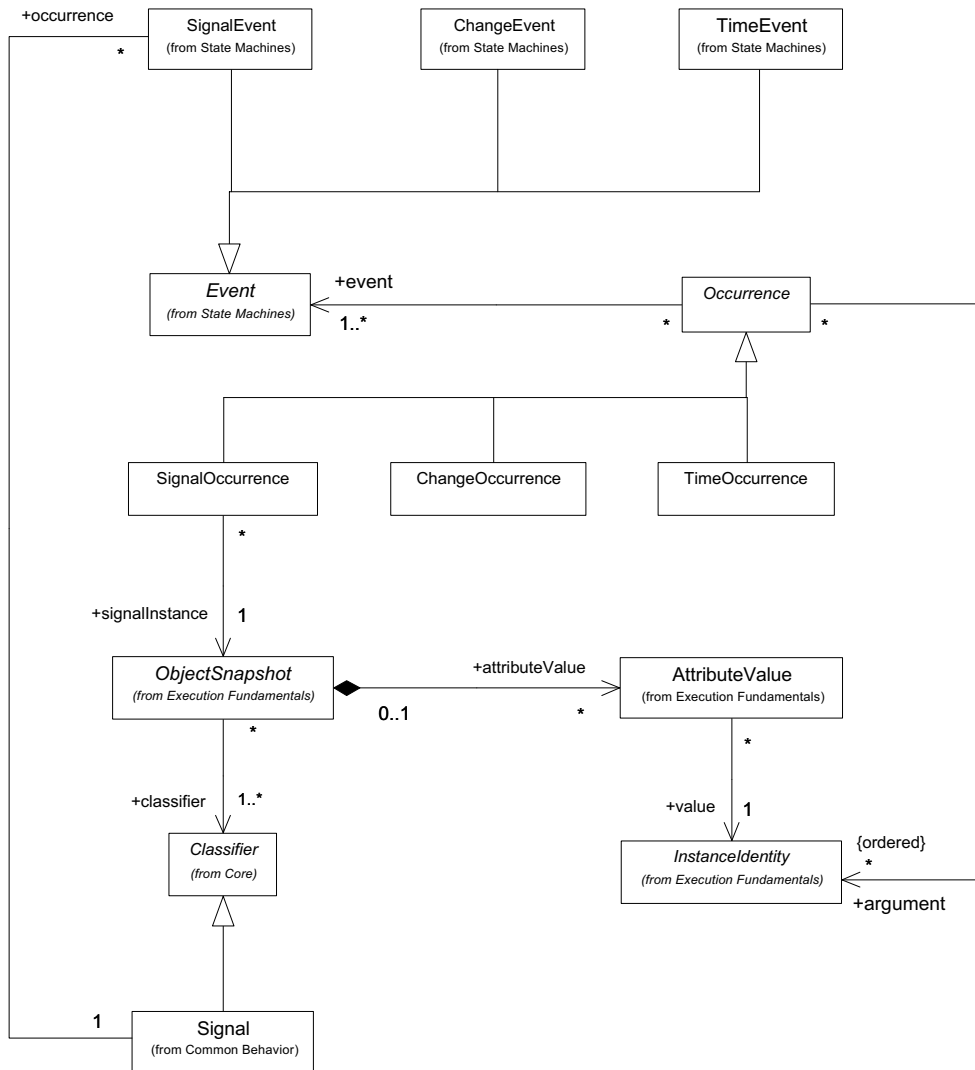


Figure 15. Occurrence Model

5.4. State Machine Execution Classes

ChangeOccurrence

This class represents the occurrence of one or more change events.

Well-formedness Rules

- [1] *Change-event conformance.* All the events of a change occurrence must be change events.
`self.event->forAll(event | event.ocIsKindOf(ChangeEvent))`
- [2] A change occurrence has no arguments.
`[1] self.event->forAll(event | event.ocIsKindOf(ChangeEvent))`
`self.argument->isEmpty`

ObjectSnapshot

If an object snapshot represents an active object, it maintains a set of unhandled occurrences.

Associations

Note. This is in addition to those specified for ObjectSnapshot in Chapter 4, “Execution Fundamentals”.

occurrence	The set of occurrences relevant to this object that have happened but have not yet been handled.
------------	--

Well-formedness Rules

[1] *Active-object conformance.* If no classifier of an object snapshot is active, then the set of occurrences is empty.
 not self.classifier->exists(c : Classifier | c.oclAsType(Class).isActive) implies self.occurrence.isEmpty

Occurrence

This class represents the occurrence of an event.

Associations

event	The set of events that are occurring.
argument	The set of values for arguments of the event.

SignalOccurrence

This class represents the occurrence of a signal event.

Associations

signalInstance	The signal instance that triggered this occurrence.
----------------	---

Well-formedness Rules

[1] *Signal-event conformance.* The signal instance of a signal occurrence must have a signal as its only classifier. The events of a signal occurrence must all be signal events with signals that are the same as the classifier of the signal instance or generalizations of it. The arguments of a signal occurrence are the values of the attributes of the signal instance, in the order given by the order of the attributes of the signal.

```
self.signalInstance.classifier->size = 1
and self.signalInstance.classifier->asSequence->first.oclIsKindOf(Signal)
and self.event->forall(e:Event | e.oclIsKindOf(SignalEvent)
and e.oclAsType(SignalEvent).signal.allSupertypes->includes(self.signalInstance.classifier->asSequence->first)
and self.argument = self.signalInstance.attributeValue.value
```

StateMachineExecutionIdentity

This class represents the immutable identity of a specific execution history for a state machine

Associations

machine	The state machine being executed.
host	The object with which the state machine execution is associated.
occurrence	The set of all occurrences received by the state machine throughout the execution history with this identity.

Well-formedness Rules

[1] *State-machine occurrence consistency.* The events of an occurrence associated with a state-machine-execution identity must be from the state machine for that identity.

```
self.occurrence.event->forAll(event : Event | self.machine.allTransitions->
  collect(t:Transition| t.trigger) ->includes(event))
```

With the following additional operations defined on StateMachine:

```
allTransitions::Set(Transition)
  post: result =
    self.transition->union(self.allSubStates().internal)
allStates::Set(State)
  post : result = self.top.allSubstates()->including(self.top)
```

With the following additional operation defined on StateVertex:

```
allSubstates::Set(StateVertex)
  post: result =
    if self.oclIsKindOf(CompositeState)
    then self.oclAsType(CompositeState).subvertex->union
      self.oclAsType(CompositeState).subvertex ->collect(s:StateVertex|s.allSubstates)
    else Set{ }
  endif
```

StateMachineExecutionSnapshot

This class represents a snapshot at a specific moment in time of the execution history of a state machine.

Associations

executionId	(<i>Derived</i>) The identity of the snapshot, as a state-machine-execution identity.
current	The current occurrence for this snapshot.
configuration	The set of states that compose the current state configuration of the state machine.

Well-formedness Rules

- [1] *Current-occurrence consistency.* The current occurrence of a state-machine-execution snapshot must be in the set of occurrence receptions of the state-machine-execution identity of the snapshot.
self.executionId.occurrence->includes(self.current)
- [2] *State-machine conformance.* The configuration of a state-machine-execution snapshot must include the top state of the state machine of the identity of the execution snapshot. All the other states in the configuration must be contained, directly or indirectly, in this top state.
self.configuration->includes(self.executionId.machine.top)
and self.configuration->forAll(s:State | s <>self.executionId.machine.top implies
self.executionId.machine.top.allSubstates->includes(s))

TimeOccurrence

This class represents the occurrence of a time event.

- [1] *Time-event conformance.* All the events of a time occurrence must be time events. A time occurrence has no arguments.
self.event->forAll(e:Event | e.oclIsKindOf(TimeEvent))
and self.argument->isEmpty

Issue. Should a time occurrence have the actual time of the occurrence as an argument?

Part III. Actions

This part contains descriptions of the various actions that can be included in user models. Related actions are organized into chapters. For each action, a description is given of its model structure, well-formedness rules on its use within a model, and its semantics specified by before-after conditions on the state of the execution model.

Each chapter is structured in two major components. The first is a description of the content in a manner that we hope will be easy to understand and grasp. The second part, starting “<Chapter Name> Classes” is an alphabetically ordered description of the classes. Only this second component is intended to be normative. In turn, the normative description is divided into two parts, one that describes the classes available to the user, and a second part that describes the execution model classes.

The first chapter in this part (Chapter 6, “Action Foundation”) describes the abstract class Action and the various structural classes that are used to describe actions. The remaining chapters in the part describe the various specific action classes that modelers can use to specify systems. Related actions are grouped into chapters.

Chapter 6. Action Foundation

This chapter describes the structure of procedures and actions in the UML metamodel and their run-time behavior. The normative definition of the classes comes in two parts: the first part covers the foundation itself, and the second covers its execution.

The next chapter describes the composite actions that permit large actions to be built out of smaller ones. These two chapters describe the generic execution model for actions. Subsequent chapters describe the structure and execution of particular primitive actions.

6.1. Action Specification

An action is the fundamental unit of behavior specification. An action takes a set of inputs and converts them into a set of outputs, though either or both sets may be empty. The inputs to an action may be obtained from the results of other actions, and the outputs of the action may be provided as inputs to other actions, some of which have the sole purpose of reading or writing object memory..

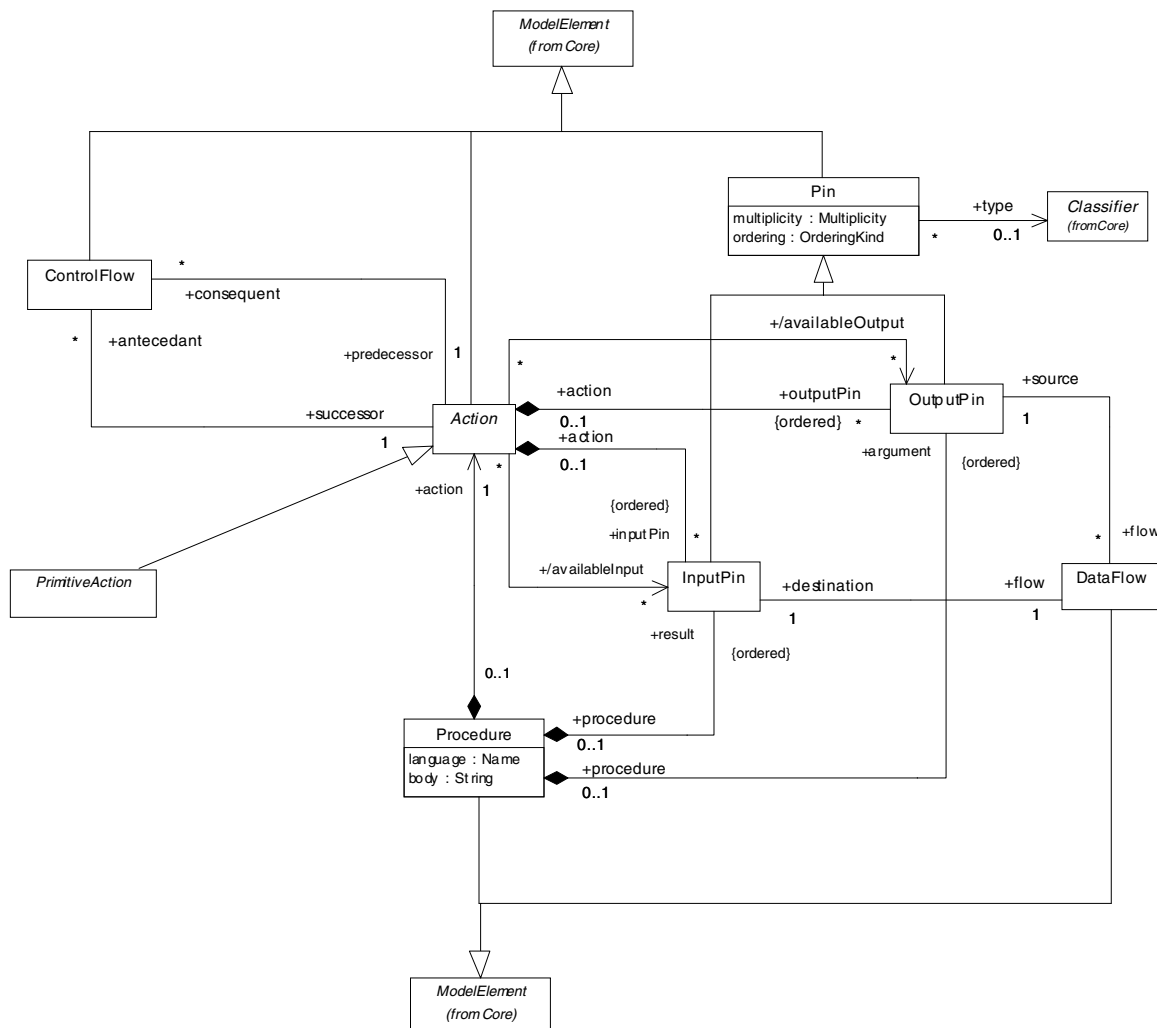


Figure 16. Action foundation model

Actions and Pins

An action takes some input values, performs some processing and produces some set of output values. The required inputs and outputs of an action are represented as *pins* of the action

Input pins are the connection points for delivering the input values to actions. *Output pins* are the connection points for obtaining the output values from actions. The types and multiplicities of the input and output pins of an action are constrained by the *form* of the action. For example, an action that reads a certain attribute has an output pin with the type and multiplicity of that attribute, while an action that writes to an attribute has an input pin with the type and multiplicity of that attribute. An action that calls an operation has (possibly empty) sets of input and output pins with types and multiplicities corresponding to the input and output parameters of the operation.

The entire context for an action is represented in its pins, including the “current object” which is also passed to an action on a pin. There is no “back door” by which an action can access objects implicitly.

Note. The types of the input and output pins of an action form the input and output “signature” of the action. The signature is modeled by derivation from the types of the input and output pins; it is not modeled explicitly. Some actions impose constraints on the types of some of their pins.

Data Flow

A *data flow* from an output pin to an input pin indicates that an output value of one action is used as an input value of another action. Data flows link chains of actions without having to store values temporarily

A single output pin can be connected to zero or more input pins, but each input pin can have at most one connection. Thus, inputs pins are “single assignment” data holders—once they receive a value, this value does not change. In other words, normal dataflow connections allow “fan out”, but not “fan in.”

The input pin that is the destination of a data flow must *conform* to the output pin that is the source of the flow: the type of the output pin is the same as or a descendant of the type of the input pin, and all cardinalities allowed by the multiplicity of the output pin are allowed by the multiplicity of the input pin. For example, if the type of the output pin is money, and the type of the input pin is real, and money is a descendant of real, then the types conform.

Control Flow

A *control flow* indicates an ordering constraint between a predecessor action to a successor action without explicit data flow. A predecessor action must complete execution before its successor action can execute. Such control constraints are often required for actions that affect object memory, such as when a value written to an attribute by one action is to be read by a subsequent action. Control flow specifies the order of actions that require such constraints.

Other actions create or destroy objects, communicate among objects, or have external effects outside the system. Control flow is used to order these actions as well. Control flow represents an implicit potential communications path among actions—through object memory, by signal transmission, or outside of the system.

Traditional programming languages have implicit control flows between each statement, but they overspecify execution order. The model defined here permits control flows, data flows, and concurrent actions as needed.

A data flow implies a control dependency in the sense that an executing action cannot consume an input value during execution until it has been produced by the source action. Control sequencing is implicit between actions connected by data flows; it is unnecessary to include explicit control flows between such actions.

The network of actions connected by data and control flows forms an acyclic directed graph because an action cannot be both a predecessor and successor of another at the same time.

Primitive Actions

A *primitive action* is one that cannot be decomposed into other actions. Primitive actions include purely mathematical functions, such as arithmetic and string functions; actions that work on object memory, such as read actions and write actions; and actions related to object interaction, such as call actions and send actions. Each kind of primitive action has a form that specifically defines sets of input and output pins. Primitive actions are defined as a convenience to provide a common definition of these well-formedness rules for all kinds of primitive actions. The details of the various kinds of primitive actions are discussed in later chapters in this part.

Procedures

Within a user model, actions may be nested at various levels. The highest-level such grouping is the *procedure*. A procedure is a set of actions that may be attached as a unit to other parts of the user model: as the body of a method or the behavior executed on entry to or exit from a state or on the firing of a transition. All uses of procedures may have *arguments*, corresponding to the input parameters. Methods and transitions triggered by invite events may have results; other transitions, as well as entry and exit procedures, do not. See Chapter 5, “State Machine Execution”.

The arguments supply values to the action by *output* pins within the procedure, and each one may be connected to zero or more inputs of the action. Similarly, the results are represented as *input* pins within the procedure, and each one must be connected to exactly one output of the action. The data flow connections then follow the normal connection rules for input and output pins.

As with any action, the pins on a procedure action must match the types and multiplicities of the corresponding parameters supplied to the method, or the supplemental data items on the triggering event.

Procedures can have a textual specification entered in the body attribute, with the kind of specification given in the language attribute.

Issue. We have used the term “arguments” for both procedure arguments and the supplemental data items passed in with an event. There is some utility in separating the two terms, but in this chapter they are truly identical in their treatment, though not their specification.

6.2. Action Execution

Identical action specifications can appear multiple times within a larger action specification, because each appearance may have a different context from other appearances. Except for their context, all such identical action specifications are equivalent. Each action in a procedure may execute zero, one, or more times for each procedure execution, depending on control structures defined in Chapter 7, “Composite Actions”

An action execution corresponds to an individual execution of a particular action. Similarly, a *procedure execution* is an individual run-time execution of a procedure.

Execution is not instantaneous, but takes place over a period of time. Procedures and other groupings of actions must be executed in a number of steps, including control of the execution of nested actions. Thus, procedure and action executions, in general, require the maintenance of state information over time.

As shown in Figure 17, procedure and action execution dynamics are similar to method and state-machine execution. Each procedure and action execution has a unique identity independent of its state information at any time. Time-dependent information is then captured in procedure- and action-execution snapshots.

In this way, the entire execution of a procedure or action is captured by a history of execution snapshots. The behavioral semantics of the procedure or action are then captured in the *well-formedness rules* of the snapshots to the procedure or action specification. However, unlike the well-formedness rules for static entities, the rules for an execution snapshot will determine its values in terms of the values of the previous snapshot in the execution history, capturing the transformational nature of the dynamic semantics of procedures and actions.

This history-based approach makes our definitions of dynamic semantics well-grounded and non-circular. Any execution-engine implementation must provide the same observable effects on system state as specified here.

Procedural Context

As shown in Figure 17, a procedure execution acts as a *context* for all action executions nested directly or indirectly within it. This contextual information is associated with the procedure-execution snapshot. It includes the following.

- *Host.* A procedure may execute as the result of an event occurrence triggering a transition or the invocation of a method (see Chapter 5, “State Machine Execution”, and Chapter 11, “Messaging Actions”). The *host* of the execution is the instance that owns the state machine for the transition or the invoked method. The host is associated with the procedure-execution identity, and it remains fixed throughout the execution. (There is no host instance for procedures associated with a method with classifier scope or for procedures acting as of expressions.)

- *Action executions.* A procedure is the root of a tree of current action executions. Each of these action executions can reference the identity of its parent action execution. However, it is generally not possible to determine statically which actions within the procedure will actually execute, or even at all. The current set of action executions nested within a procedure execution will generally change over time. Nevertheless, this set provides the ability to map from the specification of an action within a procedure to the specific, current execution of that action within the context of a given procedure execution (if, of course, that action is currently executing at all).

Since pins are not referenced independently from the actions (or procedures) that own them, pin values also do not have an independent identity, but are considered properties of the procedure- and action-execution snapshots that own them. When an action execution has completed, its snapshot has pin values for all its pins.

Similarly, it is unnecessary to have explicitly “data-flow executions” and “control-flow executions” inside a procedure execution, because the action executions inside a procedure execution are wired together by pins the same way that the actions are wired together by flows in the procedure specification, so they can be derived.

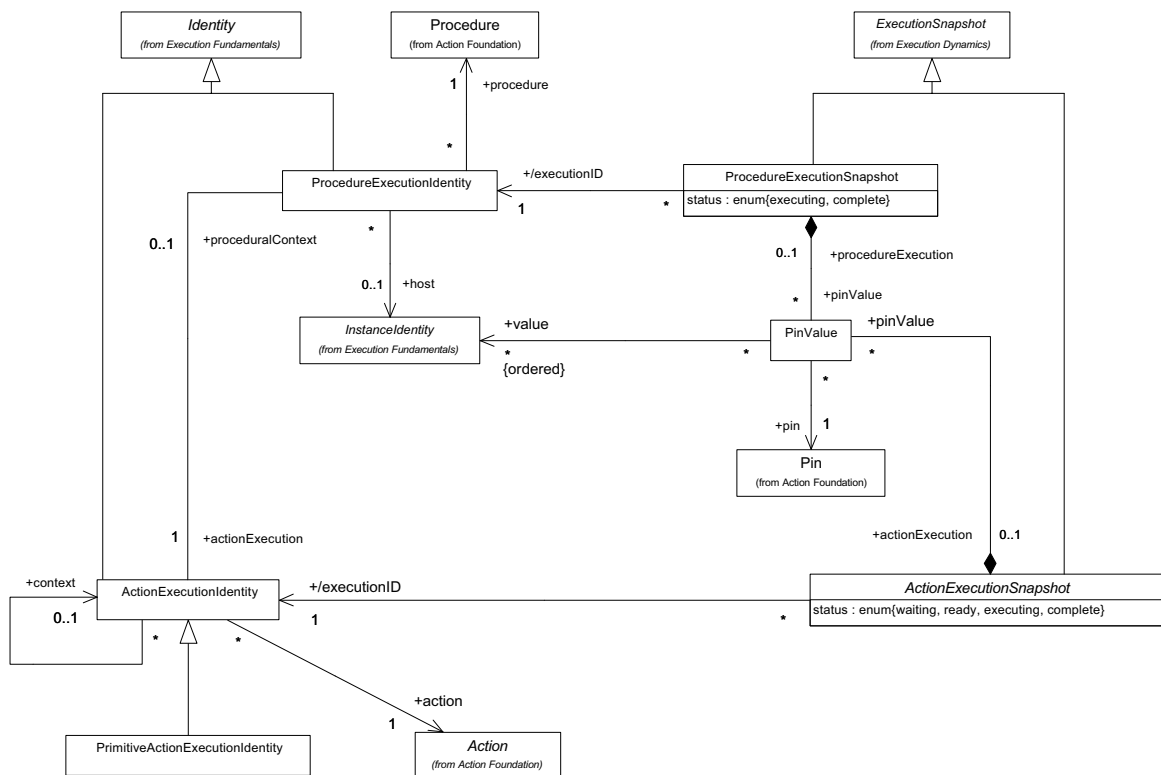


Figure 17. Action foundation execution model

Action Execution Lifecycle

The UML action semantics place no restriction on the relative execution order of two or more actions, unless they are explicitly constrained by data flow or control flow relationships. Hence every action within a procedure may execute at once, though an actual implementation may be sequential or parallel.

The execution of an action proceeds through a life cycle whose stages are given by the *status* attribute of the action execution (see Figure 17). These stages can be understood in terms of the constraints they place on the snapshots in the history of an action execution.

- *Waiting.* After the creation of an action execution, the next snapshot in its history has the status *waiting*. This snapshot has no pin values.
- *Ready.* The next snapshot in the history has the status *ready*. This snapshot must be synchronized with completed snapshots for all *prerequisite* actions (that is all actions that are the sources of data flows or predecessors of control flows into the action becoming ready). This captures the ordering in time between the completion

of prerequisite action executions and the readying of their target. The values of the input pins of the target snapshot are determined by the values of the output pins from the prerequisite action-execution snapshots for actions that are the sources of data flows. (Enclosing composite actions may also place constraints on the execution of nested actions—for instance, conditional execution.)

- *Executing*. The next snapshot has the status *executing*. An action need not begin executing immediately upon being ready and the action semantics does not determine any time delay between the ready and executing snapshots. For a primitive action, there will be only one executing snapshot in the history. However, a composite action may require several steps to complete execution.
- *Complete*. The final snapshot in the history of an action execution must have the status *complete*. This snapshot has pin values for all output pins of the action as well as all input pins. The specific semantics of each kind of action determines how these output pin values are computed.

Figure 18 pictures this life cycle operationally using a statechart diagram. However, this operational depiction is to be understood in terms of snapshot histories, as described above.

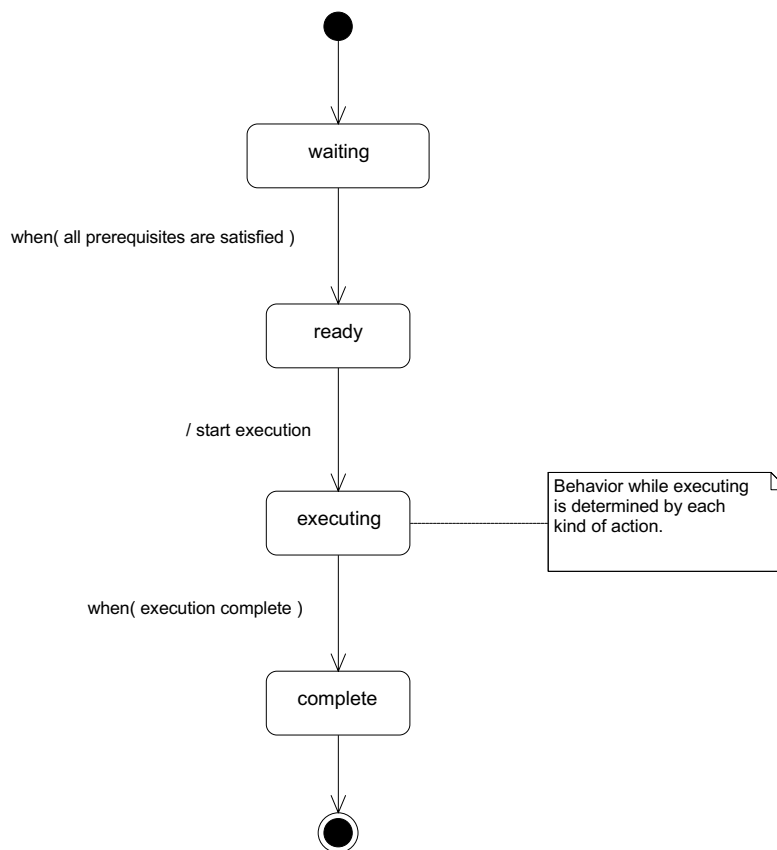


Figure 18. Life cycle for action execution

Procedure Execution

When the procedure execution begins, input parameters are passed to the procedure as values of its argument pins. These are captured as pin values associated with the initial procedure-execution snapshot. Each pin value links a single pin to the set of instance identities that are its value, consistent with the multiplicity of the pin.

The behavior of a procedure execution is determined by the execution of the action nested within the procedure. After its creation, the next step in the procedure-execution history causes the start of the execution of the nested action. The execution of the procedure is completed when the execution of the action is completed.

The completed procedure-execution snapshot has pin values for all the result pins of the procedure. In the specification of the procedure, the result pins are connected by data flows to the output pins of actions within the procedure. This specification and the procedural context of completed action executions within the procedure execution then determine the values for the result pins.

Note. A UML constraint contains an expression that can be modeled using a procedure with a boolean result. However, when such procedures should be executed and what should be done when they fail is not a simple question. Some apply at all times, but most can be violated during the actions of a transition. Some may be valid after certain actions. Currently, the action semantics specification does not attempt to verify or enforce user-defined constraints that are made invalid by actions. In effect, such constraints are considered to be design statements and the system implementer is obliged to ensure that they are not violated.

6.3. Action Foundation Classes

Action

An action is the fundamental unit of behavior specification. An action takes a set of inputs and converts them into a set of outputs, though either or both sets may be empty. If both inputs and outputs are missing, the action must have some kind of fixed, nonparameterized effect on the system state, or be performing some effect external to the system. Actions may access or modify accessible, mutable objects. A reference to an object to read or write is an input of the action. Composite actions may include data-transformation actions as well as object-access actions.

An action may have a set of incoming data flows as well as a set of explicit control flow dependencies to predecessor actions. An action will not begin execution until all of its input values (if any) have been produced by preceding actions *and* all predecessor actions have completed. The completion of the execution of an action may enable the execution of a set of successor actions and actions that take their inputs from the outputs of the action. Actions come in various kinds, each of which has its own formation rules. An action must be one of those kinds.

Associations

<i>antecedent</i>	The set of control flows that must be enabled before this action can execute.
<i>availableInput</i>	<i>(derived)</i> The set of all input pins available to be the destinations of data flows from outside the action. This implies that they are not connected by data flows within the action. The derivation rule is defined for each kind of action.
<i>availableOutput</i>	<i>(derived)</i> The set of all output pins available to be the sources of data flows out of the action. The derivation rule is defined for each kind of action.
<i>consequent</i>	The set of control flows that are enabled when this action finishes executing.
<i>inputPin</i>	The ordered set of input pins owned by the action, which act as connection points for providing values consumed by the action
<i>outputPin</i>	The ordered set of output pins owned by the action, which act as connection points for obtaining values generated by the action.

Additional operations

- [1] This operation returns the set of all immediately nested actions of this action. The actual set returned is defined in concrete descendants of Action.
context Action::nestedActions() : Set(Action)
- [2] This operation returns all nested actions of an action, nested to any depth.
context Action::allNestedActions() : Set(Action)
post: result = self.nestedActions()->union(self.nestedActions.allNestedActions())

- [3] This operation returns the set of immediate subactions of this action, whose output pins may be directly connected to the input pins of actions outside this action. The actual set returned is defined in concrete descendants of Action. (This is only intended to include subactions that are “visible” through a “porous” boundary, which currently includes only the subactions of GroupAction).
context Action::subactions() : Set(Action)
- [4] This operation returns all subactions of an action, nested to any depth.
context Action::allSubactions() : Set(Action)
post: result = self.subactions()->union(self.subactions.allSubactions())
- [5] This operation returns the set of all input and output pins of an action.
context Action::allPins() : Set(Pin)
post: result = self.inputPin->union(self.outputPin)->asSet()
- [6] This operation returns true if the action is a subaction, at any depth, of another given action.
context Action::isSubaction(otherAction: Action): Boolean
post: result = otherAction.allSubactions()->includes(self)
- [7] This operation returns the set of actions whose execution must complete before this action can execute: the source actions in data flows and predecessor actions in control flows.
context Action::prerequisites() : Set(Action)
post: result = self.inputPin.flow.souorce.action->union(self.antecedant.predecessor)
- [8] This operation returns all the actions which are destinations of data flows or successors of control flows leaving an action. A control flow from or to a group action is treated as being a set of control flows from or to each action within the group action.
context Action::successors() : Set(Action)
post: result = self.outputPin.flow.destination.action
->union((self.consequent.successor->union(self.group.consequent.successor))
->collect(a : Action | a.subactions()->including(a)))->asSet()
- [9] This operation returns the transitive closure of all data-flow and control-flow successors (defined as above) of an action.
context Action::allSuccessors() : Set(Action)
post: result = self.successors()->union(self.successors().allSuccessors()->asSet())

Well-formedness rules

- [1] There must be no cycles in the graph of actions and flows, where a cycle is defined as a path that begins and ends at the same action and a path is constructed from directed edges between actions, with control flows traversed from predecessor action to successor action and data flows traversed from the action of the source pin to the action of the destination pin. A control flow from or to a group action is treated as being a set of control flows from or to each action within the group action.
not self.allSuccessors()->includes(self)

Note. This is a necessary but not sufficient condition to prevent ill-formed control cycles. There are additional conditions related to conditional and loop actions that are handled below as well-formedness rules for those kinds of actions.

ControlFlow

A control flow is a sequencing dependency between two actions. The successor action of the flow may not execute until the predecessor action has completed execution.

Associations

<i>predecessor</i>	The action that must finish executing before the successor can execute.
<i>successor</i>	The action that cannot execute until the predecessor completes execution.

DataFlow

A data flow carries values from a source output pin to a destination input pin. When a value is generated on the source pin, it is copied to the destination pin. The source pin must therefore conform in type and multiplicity to the destination pin

Associations

<i>destination</i>	The input pin that receives the data carried by the flow.
<i>source</i>	The output pin that provides the data carried by the flow.

Note. The following operations are additional operations defined for the MultiplicityRange and Multiplicity classes specified in the UML 1.3 section on data types, not for the DataFlow class.

Additional operations

- [1] This operation checks if a given integer is within the range specified by a multiplicity range.
context MultiplicityRange::contains(i : Integer) : Boolean
post: result = if (r.upper=#unlimited) then r.lower <= i else r.lower <= i and i <= r.upper endif
- [2] This operation checks if a given integer cardinality is allowed by a multiplicity.
context Multiplicity::allows(i : Integer) : Boolean
post: result = self.range->exists(r : MultiplicityRange | r.contains(i))
- [3] This operation checks if one multiplicity is compatible with another.
context Multiplicity::compatibleWith(other : Multiplicity) : Boolean
post: result = Integer.allInstances()->forAll(i : Integer | self.allows(i) implies other.allows(i))

Well-formedness rules

Note. Since UML does not provide any standard classifier that is the ancestor of all other classifiers, untyped pins can be used for the purpose of accepting input of “any” type.

- [1] The type of the source pin must be the same as or a descendant of the type of the destination pin. An untyped pin has a type that is an ancestor of any classifier.
self.destination.type->isEmpty()
or (self.source.type->notEmpty()
and self.source.type = self.destination.type
or self.destination.type.allParents()->includes(self.source.type))

Note. The operation “allParents” is defined in the UML 1.3 specification for all generalizable elements.

- [2] All cardinalities allowed by the multiplicity of the source pin must be allowed by the multiplicity of the destination pin.
self.source.multiplicity.compatibleWith(self.destination.multiplicity)

InputPin

An input pin holds input values to be consumed by an action. An input pin may be the destination for exactly one data flow. The input pin receives its values from the source output pin of the data flow.

Associations

<i>action</i>	The action that owns the pin as an input (not used with procedures)
<i>flow</i>	The data flow for which this input pin is the destination.
<i>procedure</i>	The procedure that owns the pin as a result (not used with actions).

Well-formedness rules

- [1] An input pin must be owned by either an action or a procedure but not both.
 $\text{self.action} \rightarrow \text{size}() + \text{self.procedure} \rightarrow \text{size}() = 1$

OutputPin

An output pin holds output values generated by an action. A single output pin may have several data-flow connections to several input pins. In this case, the output pin provides a copy of its value to each of the associated input pins.

Associations

<i>action</i>	The action that owns the pin as an output.
<i>flow</i>	The data flow for which this output pin is the source.
<i>loop</i>	The loop action that owns the pin as a loop variable (see the discussion under Composite Actions).
<i>procedure</i>	The procedure that owns the pin as an argument.

Well-formedness rules

- [1] An output pin must be owned by either an action or a procedure but not both.
 $\text{self.action} \rightarrow \text{size}() + \text{self.procedure} \rightarrow \text{size}() = 1$

Pin

A pin is a connection point for delivering input values to or obtaining output values from an action. Any values passing through the pin must conform to the type of the pin and have cardinalities allowed by the multiplicity of the pin. Pin is completely specialized into input and output pins.

Attributes

<i>multiplicity</i>	A specification of the number of values a pin may hold at any one time.
<i>ordering</i>	Indicates whether the set of values held by this pin is to be considered ordered or not.

Associations

<i>type</i>	A classifier specifying the allowed classifiers of values passing through the pin. The actual classifier of a value must conform to the type specification of the pin.
-------------	--

PrimitiveAction

A primitive action is one that does not contain any nested actions, so all available inputs and outputs of the action are pins directly owned by the action.

Additional operations

- [1] A primitive action has no subactions. (The subactions operation is defined for all actions.)
`context PrimitiveAction:subactions() : Set(Action)`
`post: result = Set{ }`

Well-formedness rules

- [1] The available inputs of a primitive action are the input pins of the action.
 $\text{self.availableInput} = \text{self.inputPin} \rightarrow \text{asSet}()$
- [2] The available outputs of a primitive action are the output pins of the action.
 $\text{self.availableOutput} = \text{self.outputPin} \rightarrow \text{asSet}()$

Note. PrimitiveAction is really only defined as a convenience to provide a common definition of these well-formedness rules for all kinds of primitive actions.

Procedure

A procedure acts as the body of a method or the reaction to an event. The behavior of the procedure is specified using an action. The procedure has an ordered list of arguments, represented as output pins, that may be connected as sources to the available inputs of the body action. It also has an ordered list of results, represented as input pins, which must be connected as destinations to the available outputs of the body action.

Associations

<i>action</i>	The action that provides the behavioral specification of the procedure.
<i>argument</i>	The ordered set of output pins representing procedure arguments.
<i>result</i>	The ordered set of input pins representing procedure results.

Well-formedness rules

- [1] All available inputs of the action of a procedure must be the destinations of flows, the sources of which are arguments of the procedure.
`self.argument->includesAll(self.action.availableInput.flow.source)`
- [2] All results of a procedure must be the destination of flows, the sources of which are available outputs of the action of the procedure.
`self.action.availableOutput->includesAll(self.result.flow.source)`

6.4. Action Foundation Execution Model Classes**ActionExecutionIdentity**

An action-execution identity is the unique identity of a single execution of an action and maintains the properties of such an execution that remain fixed during the execution. An action in a user model may correspond to many action executions over time or at the same time. Each is an independent execution of the action.

Associations

<i>action</i>	The action that specifies the effect of the action execution.
<i>context</i>	The identity of the action execution that is the context of this action execution, if the action of this action execution is directly nested in another action.
<i>proceduralContext</i>	The identity of the procedure execution that contains the action execution, if the action is directly nested in a procedure.

Additional operations

- [1] This operation returns the procedure execution within which this action execution is ultimately nested.
`context ActionExecutionIdentity::procedureExecution() : ProcedureExecutionIdentity`
`post: result = if self.proceduralContext->notEmpty() then self.proceduralContext`
`else self.parent.proceduralExecution() endif`

Wellformedness Rules

- [1] An action-execution snapshot must have either a context or a parent, but not both.
`self.context->notEmpty() xor self.parent->notEmpty()`

ActionExecutionSnapshot

ActionExecutionSnapshot describes the general run-time behavior of executing an action within a specific context, particularly the setup and enabling of an action for execution.

Attributes

<i>status</i>	The state of execution of the action execution. One of:
---------------	---

waiting—created but waiting for a source to complete or produce values
ready—all sources complete and has all its values, may begin execution
executing—in the middle of execution (may be further refined under the specific action)
complete—completed and has produced all its output values

Associations

pinValue The set of pin values that have been determined at the time of this action-execution snapshot (the input pins, if the snapshot has status *ready*, and the input and output pins, if the snapshot has status *complete*).

executionID (*Derived*) The identity of the action-execution snapshot, as an action-execution identity.

Additional operations

- [1] This operation returns the identity of the host instance for this action execution.
 context ActionExecutionSnapshot::host() : InstanceIdentity
 post: result = self.executionID.procedureExecution().host
- [2] This operation returns the set of identities of action executions nested in this action execution that are available to act as prerequisites of other action executions. The actual set of executions is defined in concrete subclasses of ActionExecutionSnapshot.
 context ActionExecutionSnapshot::availableExecutions() : Set(ActionExecutionIdentity)
- [3] This operation returns the set of execution identities of currently executing actions that may act as prerequisites to this one.
 context ActionExecutionSnapshot::peerExecutions() : Set(ActionExecutionIdentity)
 post: result = if self.executionID.context->isEmpty() then self
 else let contextSnapshot=current(self.executionID.context).oclAsType(ActionExecutionSnapshot) in
 contextSnapshot.peerExecutions()->union(contextSnapshot.availableExecutions())
 endif

Note. The set of identities of all current action executions must be determined relative to a specific action-execution snapshot, because this set cannot always be statically determined (for example, the number of executions of the actions nested in a loop action cannot, in general, be statically determined). The above recursive construction of the set is necessary to ensure the selection of the “latest” execution of an action that may already have been executed more than one time, relative to the time of this action-execution snapshot.

- [4] This operation returns the identity of the current execution of the given action.
 context ActionExecutionSnapshot::executionOf(a: Action) : ActionExecutionIdentity
 pre: self.peerExecutions()->select(action=a)->size() = 1
 post: self.peerExecutions()->select(action=a)
- [5] This operation returns the current action-execution snapshot for a given action.
 context ActionExecutionSnapshot::currentSnapshotOf(a : Action) : ActionExecutionSnapshot
 post: result = current(self.executionOf(a)).oclAsType(ActionExecutionSnapshot)
- [6] This operation returns the value of a given pin for this snapshot.
 context ActionExecutionSnapshot::valueOf(p : Pin) : Sequence(InstanceIdentity)
 pre: self.pinValue->exists(pin=p)
 post: result =self.pinValue->select(pin=p)
- [7] This operation returns the source value for a given input pin.
 context ActionExecutionSnapshot::sourceValueFor(p : InputPin) : Sequence(InstanceIdentity)
 post: result = let s = p.flow.source in
 if s.action->notEmpty() then self.currentSnapshotOf(s.action).valueOf(s)
 else current(self.procedureExecution()).oclAsType(ProcedureExecutionSnapshot).valueOf(s)
 endif

[8] This operation checks that the input-pin values of this snapshot have not changed from the previous snapshot.

```
context ActionExecutionSnapshot::inputValuesUnchanged() : Boolean
post: result =
  let inputValues = self.pinValue->select(pin.ocIsType(InputPin)) in
  let preInputValues = self.predecessor().oclAsType(ActionExecutionSnapshot).pinValue->
    select(pin.ocIsType(InputPin)) in
  inputValues.pin = preInputValues.pin and inputValues.value = preInputValues.value
```

[9] This operation returns the value of an output pin of the action being executed, identified by index.

```
context ActionExecutionSnapshot::pinValueAt(i : Integer) : Sequence(InstanceIdentity)
post: result = self.valueOf(self.executionID.action.outputPin->at(i))
```

[10] This operation asserts that the current snapshot of the given action-execution identity has status *complete*.

```
context ActionExecutionSnapshot::completed(e : ActionExecutionIdentity) : Boolean
post: result = (current(e).oclAsType(ActionExecutionSnapshot).status = #complete)
```

Wellformedness Rules

[1] The identity of an action-execution snapshot must be an action-execution identity.

```
self.identity.ocIsKindOf(ActionExecutionIdentity)
and executionID=self.identity.oclAsType(ActionExecutionIdentity)
```

Production 1: An action-execution snapshot is created with status *waiting* with no pin values or action executions.

```
Precondition: self.isNew()
Assertion: self.status = #waiting and self.pinValue->isEmpty()
```

Production 2: An action execution that is waiting becomes ready when all its prerequisites have completed, at which point it obtains input values from its data-flow sources.

```
Precondition: self.status = #waiting
Postcondition: self.status = #ready
and self.executionID.prerequisites()->forall(e : ActionExecutionIdentity | completed(e))
and (self.pinValue.pin->asSet())=(self.executionID.action.inputPin->asSet())
and self.pinValue->forall(value=self.sourceValueFor(p))
```

Production 3: An action execution begins executing after it has become ready.

```
Precondition: self.status = #ready
Postcondition: self.status = #executing and self.inputValuesUnchanged()
```

Production 4: An action continues executing until it has completed.

```
Precondition: self.status = #executing
Postcondition: self.status = #executing or self.status = #complete and self.inputValuesUnchanged()
```

Production 5: An action execution completes with values for all its output pins and can then terminate.

```
Precondition: self.status = #complete
Assertion: (self.pinValue.pin->asSet())=(self.executionID.action.allPins()) and self.terminates()
```

PinValue

A pin value is the value of a specific pin of an action or procedure during a certain execution of that action or procedure.

Associations

<i>actionExecution</i>	The action-execution snapshot that owns the pin value.
<i>procedureExecution</i>	The procedure-execution snapshot that owns the pin value. (A pin value is owned by an action-execution snapshot or a procedure-execution snapshot but not both.)
<i>pin</i>	The pin for which this pin value specifies values.
<i>value</i>	The set of instance identities that represent the values held by the pin.

Wellformedness Rules

- [1] The number of values of a pin value must be consistent with the multiplicity of the pin of the pin value.
`self.pin.multiplicity.allows(self.value->size())`

Note. Because of the possibility of dynamic classification, it is not possible to have a dynamic typing rule based on the identities of the values held by a pin, since dynamic classification introduces the possibility of the type of a value changing over time. However, in the absence of dynamic classification, the static type Wellformedness Rules on data flows ensure that pins always hold values of a proper type, as long as the outputs produced by actions conform to the types of their output pins.

PrimitiveActionExecutionIdentity

A primitive-action execution identity is the identity of the execution of a primitive action. A primitive action has no nested actions, by definition, so a primitive-action execution never has any nested action executions.

Additional operations

- [1] A primitive-action execution has no available action executions.
`context PrimitiveActionExecutionIdentity::availableExecution()`
`post: result = Set{ }`

Wellformedness Rules

- [1] The action of a primitive-action execution identity must be a primitive action.
`self.action.oclIsKindOf(PrimitiveAction)`

ProcedureExecutionIdentity

A procedure-execution identity is the unique identity of a single execution of a procedure and maintains the context for all action executions within the procedure execution. A procedure in a user model may correspond to many procedure executions over time or at the same time. Each is an independent execution of the procedure.

Association

<i>procedure</i>	The procedure that specifies the effect of the procedure execution.
<i>host</i>	The identity of the host instance for this procedure execution.
<i>actionExecution</i>	The identity of the execution of the action executed by this procedure execution.

Wellformedness Rules

- [1] The action execution nested in a procedure-execution identity must be for the action of the procedure.
`self.actionExecution.action=self.procedure.action`

ProcedureExecutionSnapshot

In the execution model, a procedure-execution snapshot is the run-time context for execution established by the invocation of procedure for a method or from a state machine.

Attributes

<i>status</i>	The state of execution of the procedure execution. One of: <i>executing</i> —executing its action <i>complete</i> —completed and has produced all its results
---------------	---

Associations

<i>pinValue</i>	The set of pinValues that have been determined at the time of this procedure-execution snapshot (the argument pins for the initial snapshot and all pins for the final snapshot).
<i>executionID</i>	<i>(Derived)</i> The identity of the procedure-execution snapshot: procedure-execution identity.

Additional operations

- [1] This operation returns the current snapshot of the action executed by the procedure.
 context ProcedureExecutionSnapshot::currentActionSnapshot() : ActionExecutionSnapshot
 post: result = current(self.executionID.actionExecution).oclAsType(ActionExecutionSnapshot)
- [2] This operation returns the value of a given pin for this procedure-execution snapshot.
 context ProcedureExecutionSnapshot::valueOf(p : Pin) : Sequence(InstanceIdentity)
 pre: self.pinValue->exists(pin=p)
 post: result =self.pinValue->select(pin=p)
- [3] This operation returns the source value for a given result pin (as a “pre” value).
 context ProcedureExecutionSnapshot::sourceValueFor(p : InputPin) : Sequence(InstanceIdentity)
 post: result = self.currentActionSnapshot().valueOf(p.flow.source)
- [4] This operation asserts that the execution of the procedure action has been started.
 context ProcedureExecutionSnapshot::actionExecutionStarted()
 post: result = new(self.executionID.actionExecution)

Wellformedness Rules

- [1] The identity of a procedure-execution snapshot must be a procedure-execution identity.
 self.identity.oclIsKindOf(ProcedureExecutionIdentity)
 and self.executionID=self.identity.oclAsType(ProcedureExecutionIdentity)

Production 1: A procedure-execution snapshot is created with values for all the argument pins of its procedure and then begins executing its action.

Precondition: self.isNew()

Assertion: self.pinValue.pin->asSet() = self.executionID.procedure.argument->asSet()

Postcondition: self.status=#executing and self.actionExecutionStarted()

Production 2: Procedure execution continues while the procedure’s action has not completed.

Precondition: self.status=#executing

Postcondition: self.currentActionSnapshot().status<>#complete implies self.status=#executing

Production 3: Procedure execution completes when its action completes, with output values from the action copied to its result pins, otherwise it keeps executing.

Precondition: self.status=#executing

Postcondition: self.currentActionSnapshot().status=#complete implies

(self.status=#complete and self.executionID.procedure.result->forall(p : Pin | p.value=self.sourceValueFor(p)))

Production 4: After it has completed, a procedure-execution snapshot can terminate.

Precondition: self.status=#complete

Assertion: self.terminates()

Group Action

The simplest form of composite action is the *group action*. A group action composes a set of subordinate actions into a higher-level unit.

A group action does not own any pins of its own. Data flows are not connected to a group action, instead they may “cross the boundary” of a group action to connect to input and output pins of actions within the group, making the boundary of the group action “porous”. A group action does not encapsulate its contents, rather this approach to grouping focuses on convenience in organizing a computation, not encapsulation or decomposition. In the example, the group action contains three subordinate actions, each with its own input and output pins. The pins on the subordinate actions are connected to other pins, both within and outside of the group action. Pins must be connected to other pins of the same type.

Group actions as a whole can be predecessors and successors in control flows, so they provide the ability to synchronize the execution of groups of actions. A control flow whose destination is a group action requires that the predecessor must complete before *any* action within the group action may begin. A control flow whose source is a group action requires that *all* actions within the group action must complete before the successor may begin.

Control flows may also cross the boundary of a group action. These control flows are in addition to any control flows to or from the group action itself. A subordinate action may execute if it has all of its data inputs and all of its control flow inputs and if the group action itself has all of its control flow inputs. Similarly, an external successor of a subordinate action must wait until the subordinate action is complete, but it need not wait for the entire group action to complete unless it is a successor of the group action.

Conditional Action

A *conditional action* provides the conditional execution of contained actions depending on the result of test actions.

A conditional action consists of some number of clauses, each of which has an embedded test action and body action. Each clause designates an output pin of its test action as the test output. If it evaluates to true, the body is executed. The body of each clause designates the *same* bank of output pins as outputs, so that the conditional action will produce the same types of output values regardless of which clause body executes. If the body executes, the values of these pins become the values of the output pins of the overall conditional action.

Conditional actions provide the only points of “fan in” of data flow. This fan in within a conditional action does not violate the single assignment principle, since only one of the body actions can execute during any execution of the conditional action, so the each conditional output pin will receive only one value. Also, since exactly one body action must execute, each conditional output pin will always receive a value.

The output pins of a body action may not be directly connected to input pins outside the body action. The output pins of test actions may not be connected to input pins outside the clause. The input data values for all test actions and all body actions come from available output pins in the scope containing the conditional action. There are no data flows among the different test and body actions. The different test and body actions may or may not use the same input values. The conditional action has no explicit input pins. There are no explicit data flows from the output pins of the clauses to the output pins of the overall conditional action. The connection is implicit in the structure of the conditional action itself. Supplying explicit data flow connections violates the well-formedness rules.

Note. In principle, outputs of a test could be connected to inputs of the corresponding body, but it seems simpler to keep the test and body actions separate except for their implicit control link.

The clauses of a conditional action may have noncyclic predecessor-successor relationships among them. Clauses with no predecessor-successor relationships may execute their test actions concurrently. If more than one of these is true, only one body action will execute, but its selection is indeterminate. If the tests are not guaranteed to be exhaustive, the user may provide a default action with a test of “true” as a successor of all other tests. A conditional is not required to have an explicit “else” clause, but the modeler must ensure that at least one test action is true or the model is incorrect.

Note. One exception is allowed to the “exactly one body action must execute” rule. In the case that the conditional action has *no* output pins, then it is allowable for *no* body actions to execute (i.e., for *all* test actions to fail). In this case, any effect of the conditional action is by affecting object memory or the external world. This is equivalent to providing an “else” clause with an empty body.

In cases when the tests in a conditional will be both exhaustive and mutually exclusive by design, the conditional action can be explicitly tagged as being “determinate.” *Exactly one* concurrent test action must evaluate to true.

Determinism is an assertion by the designer—if it is not correct, the model is ill formed. It can be achieved either by complete, explicit sequencing of the test actions or through the designer’s knowledge that tests that are not sequenced are mutually exclusive. The latter case does not imply a need for the run-time system to test that the other tests do indeed evaluate to false—the developer has the responsibility to guarantee mutual exclusion.

Loop Action

The final kind of composite action is the *loop action*. The loop action provides for repeated execution of a contained action so long as a test action results in an appropriate value.

A loop action contains a single clause with a test action and a body action. The body action is executed repeatedly as long as the test action yields “true”. The test and the body actions have access to the values of a set of “loop variables,” which are represented as an ordered set of output pins owned by the loop action.

The loop variables may be connected to input pins of the test and body actions, thus providing the “current values” of the loop variables during a loop iteration. The clause designates a set of output pins within its body subaction. The types of these pins must conform to the types and multiplicities of corresponding loop variables. At the completion of execution of the body action, the values of these pins become the values of the loop variable pins for the next iteration of the loop.

The loop action also has a list of input pins and a list of output pins. Both lists conform to the loop variable pins and the body output pins. Before the first execution of the loop clause, the values of the loop inputs become the values of the loop variables. During each iteration, the test action of the clause is executed. If its designated test output pin is false, then the values of the loop variable pins become the values of the output pins of the loop action, and the execution of the loop is complete. If the test value is true, the body action of the clause is executed. When it is complete, the body output values become the new values of the loop variables.

The loop variables are not explicitly “reassigned” in the body action. Instead, the input and output pins for the body action hold the “old values” and the “new values” of the loop variables, respectively, during a single iteration. This preserves the single-assignment principle. There are no explicit dataflow connections from the loop input pins to the loop variable pins, from the body output pins to the loop variable pins, or from the loop variable pins to the loop output pins. The dataflow is implicit, based on the structure of the loop action itself.

During execution of a loop action, the test action and the body action have access to output pins outside the loop action. For any one execution of the loop, the value on one of these pins will be fixed during all the iterations of the loop. The output pins within the test action or the body action may not be connected to input pins outside each action.

Local Variables

In addition to data flows, it is also possible to pass data between actions using *local variables*. A local variable is a slot for values shared by the actions within a group but not accessible outside it. The output of one action may be written to a variable and used as the input to a subsequent action, providing an indirect communication path.

The inclusion of variables supports both traditional imperative programming and data-flow programming, as well as mixtures of the two styles. In an imperative style, the result of an action is placed in an object attribute or a local variable, and a subsequent action retrieves it. Because there is no explicit relationship between the actions, they must be sequenced by a control flow. And because there may be many reads and writes to a particular attribute or variable, there is a danger of conflict that must be considered and prevented by the specifier.

In a purely imperative style in which each action reads and writes variables, the UML action model uses data flows to connect a single read or write action. The data flows in this case can be considered purely formal, and the different actions are otherwise disconnected and communicate by values in variables.

Each branch of a conditional must have exactly the same data-flow outputs. Using an imperative style, each branch updates the variables that it changes and leaves the others unchanged. There are no data-flow results from any branches, therefore all branches automatically match in results.

7.2. Composite Action Execution

Each kind of composite action has its own kind of execution snapshot. These snapshots maintain relationships, directly or via their identities, with the identities of the executions of actions nested within the composite action. These relationships are used to capture the control and synchronization constraints that the composite execution imposes on the execution of the nested actions, as shown in Figure 20.

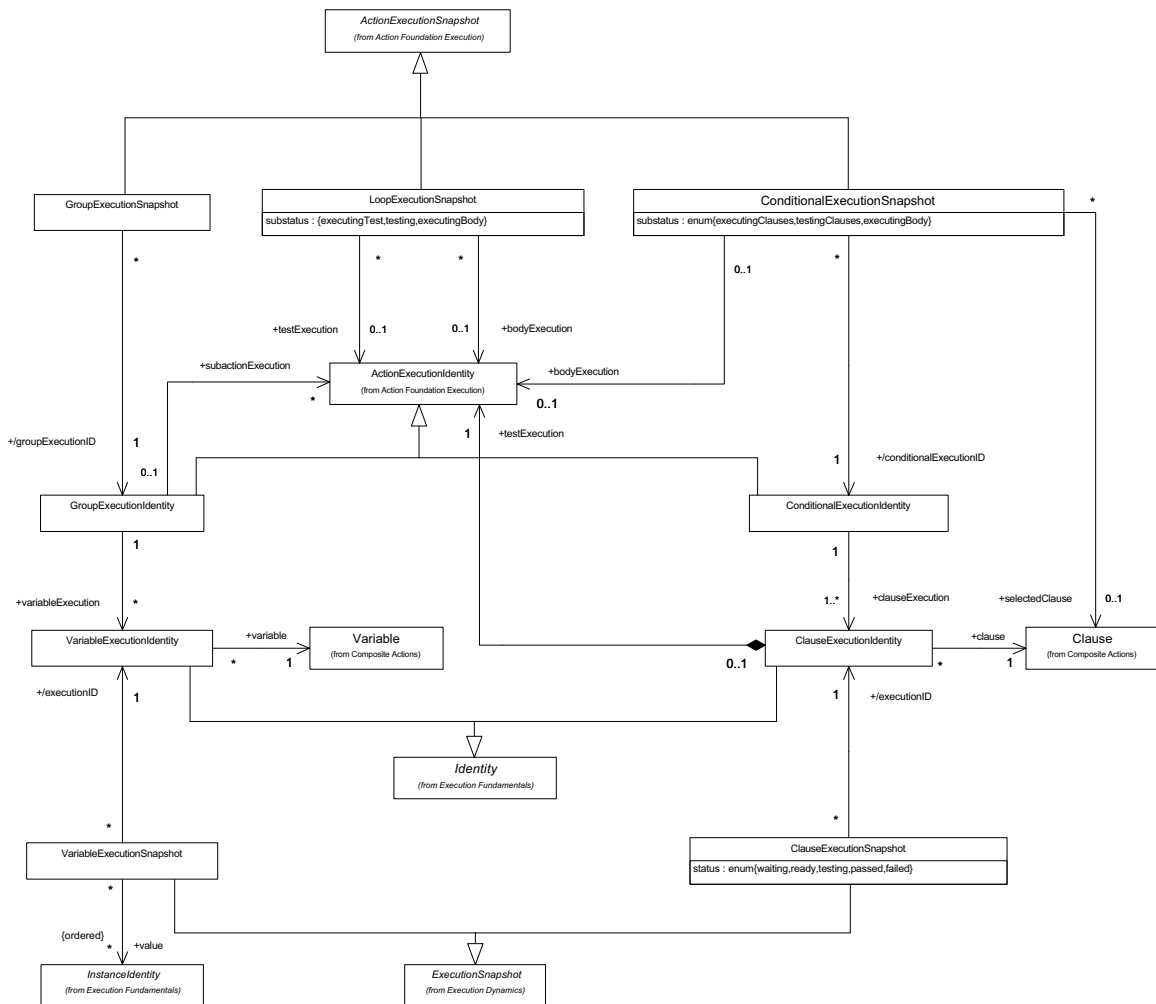


Figure 20. Composite action execution model

In the case of a conditional action, the relationship to the nested action-execution snapshots is provided indirectly via a clause-execution snapshot. This conveniently separates the histories of individual clauses from the histories of the enclosing conditional or loop actions. Clause-execution snapshots are not action-execution snapshots themselves, because clauses are not actions.

Group Action Execution

Figure 21 shows the life cycle for the execution of a group action. As with any action, a group action becomes ready when all the prerequisite actions have completed. Since a group action does not directly own any inputs itself, it can only have control-flow prerequisites. The subactions within a group action may not start executing until the group action as a whole is ready. A subaction may have its own prerequisite data and control flows that must be satisfied before the subaction may execute. In this sense, the group action is an additional prerequisite for all its subactions.

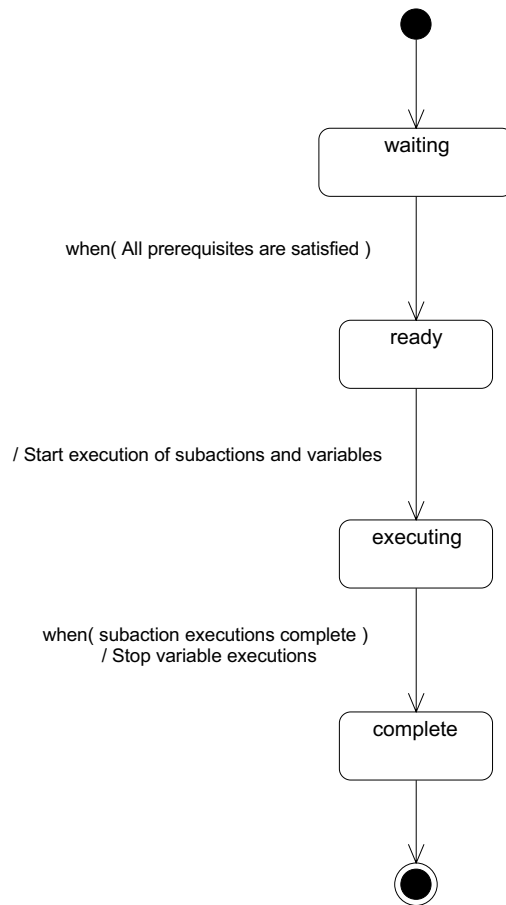


Figure 21. Life cycle for group-action execution

The group action maintains its *executing* status until all its subactions have completed. As individual subactions within the group complete, they will trigger any data or control flows attached directly to them, independently of the completion of the group action as a whole. However, control flows with the group action itself as their source will not be triggered until the group action reaches the *complete* status.

Clause Execution

Figure 22 shows the life cycle for the execution of a clause. Like an action, a clause waits for the completion of its prerequisites. The prerequisites of a clause are the control-flow prerequisites of the test action of the clause, which must be test actions in other clauses in the same enclosing conditional or loop action. Unlike an action, however, a clause only moves out of its *waiting* status if all of its prerequisites are complete *and* they have “false” outputs.

If all the prerequisites of a clause complete with “false” outputs, then the clause’s test action is executed (assuming all data-flow prerequisites of the test action are satisfied). Once the test action has completed, then the clause either passes or fails the test, depending on the result of the test-action execution. The body of a clause is not automatically executed if the clause passes, since multiple clauses may pass in a conditional, but only one body is executed.

The execution of clauses is separately modeled solely to capture the above special behavior associated with clause prerequisites. Since only a clause in a conditional action may have other clauses as predecessors, clause execution need only be explicitly modeled for conditional actions. For loop actions, clauses simply provide a convenient syntactic grouping of the test and body actions of the loop and have no semantic significance..

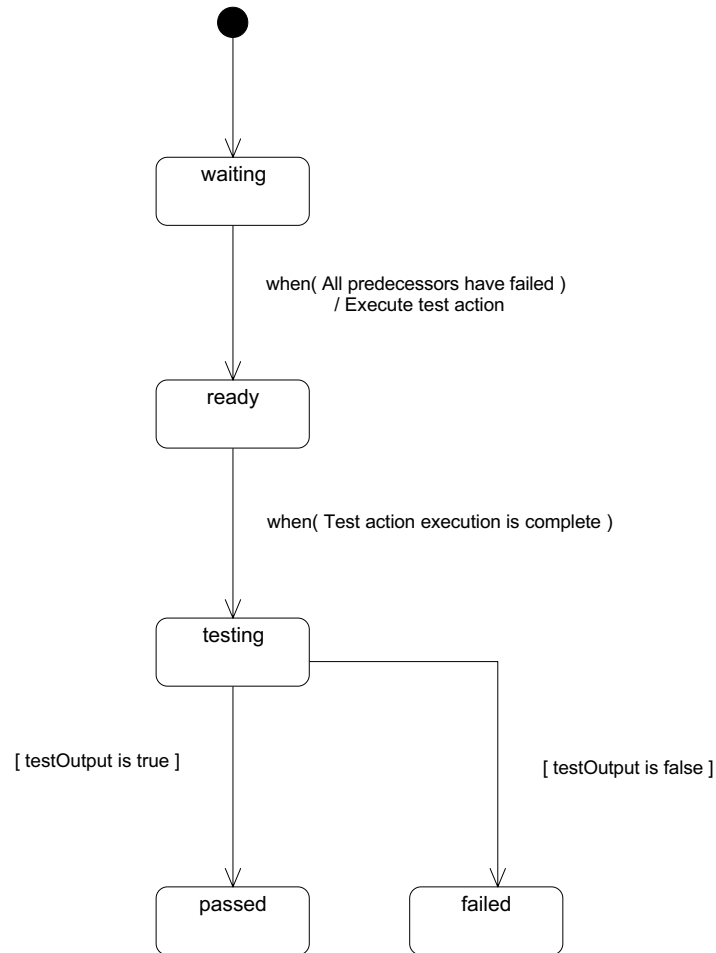


Figure 22. Life cycle for clause execution

Conditional Action Execution

Figure 23 shows the life cycle for the execution of a conditional action. As with a group action, a conditional action may only have control-flow prerequisites. Once these are satisfied, the clauses of the conditional action may begin executing. Once the clauses have completed execution, then the body of exactly one clause with status *passed* is executed, and the outputs of this body action become the outputs of the conditional action. (The one exception is in the case of a conditional action that has no outputs, in which case it is allowable for no clauses to pass.)

We need to interpret “clause execution complete” carefully, as the tests of some clauses may never be executed. Thus, clause execution may be considered complete when every clause satisfies one of these following:

- The clause has the status *passed*.
- The clause has the status *failed*.
- The clause has the status *waiting* and at least one control prerequisite with the status *passed* or *waiting*. (A clause with no control prerequisites cannot meet this condition and therefore must execute.)

Since the execution of the clause tests must complete before a body is executed, the conditional-action-execution history will have multiple snapshots with status *executing*. These correspond to the similarly named substates of the state *Executing*, as shown in Figure 23.

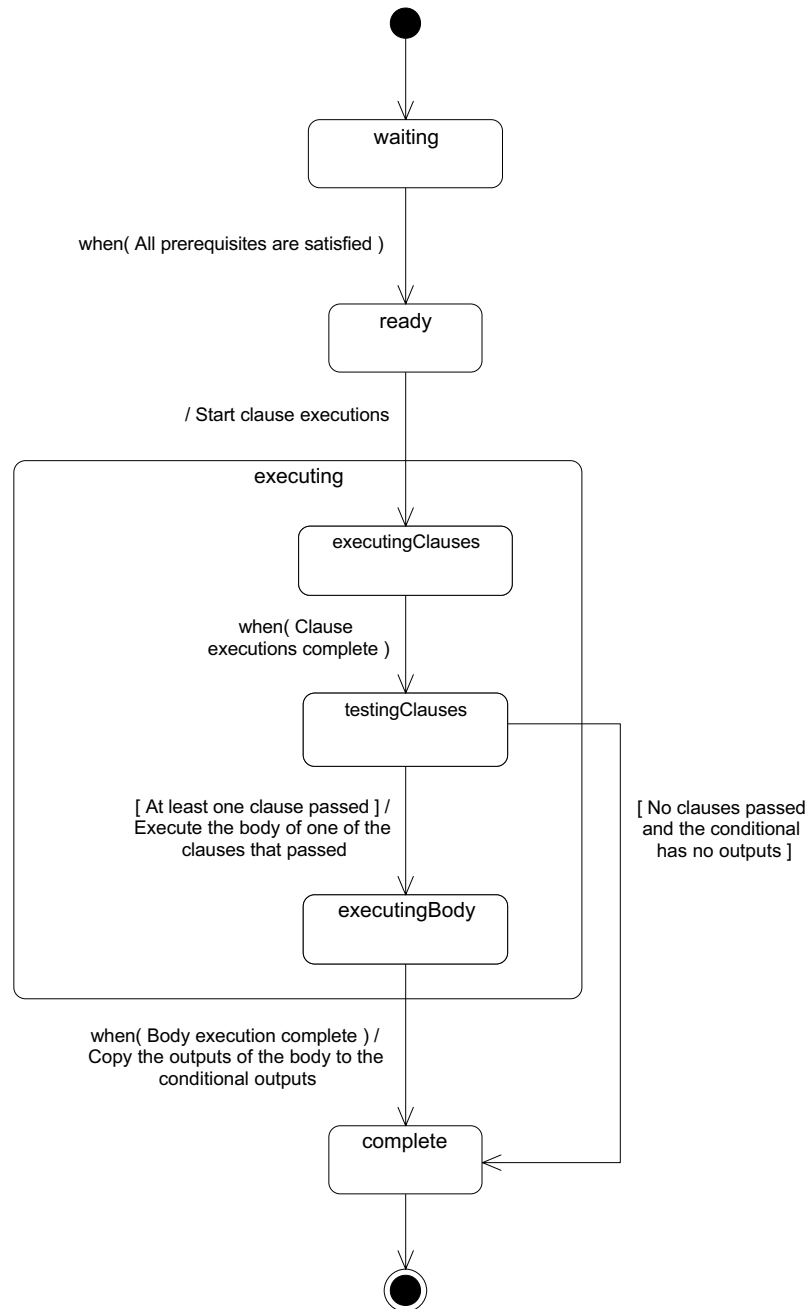


Figure 23. Life cycle for conditional-action execution

Loop Action Execution

Figure 24 shows the life cycle for the execution of a loop action. A loop action may have both control-flow and data-flow prerequisites, since a loop action may directly own input pins. Once these prerequisites have been satisfied, the values of the loop-action input pins are copied to the loop variables as their initial values and the loop clause is executed. If the clause passes its test, then the loop body is executed, its outputs are copied to the loop variables and the

clause is executed again. This continues until the clause test fails, in which case the loop variables are copied to the loop outputs and the loop is complete.

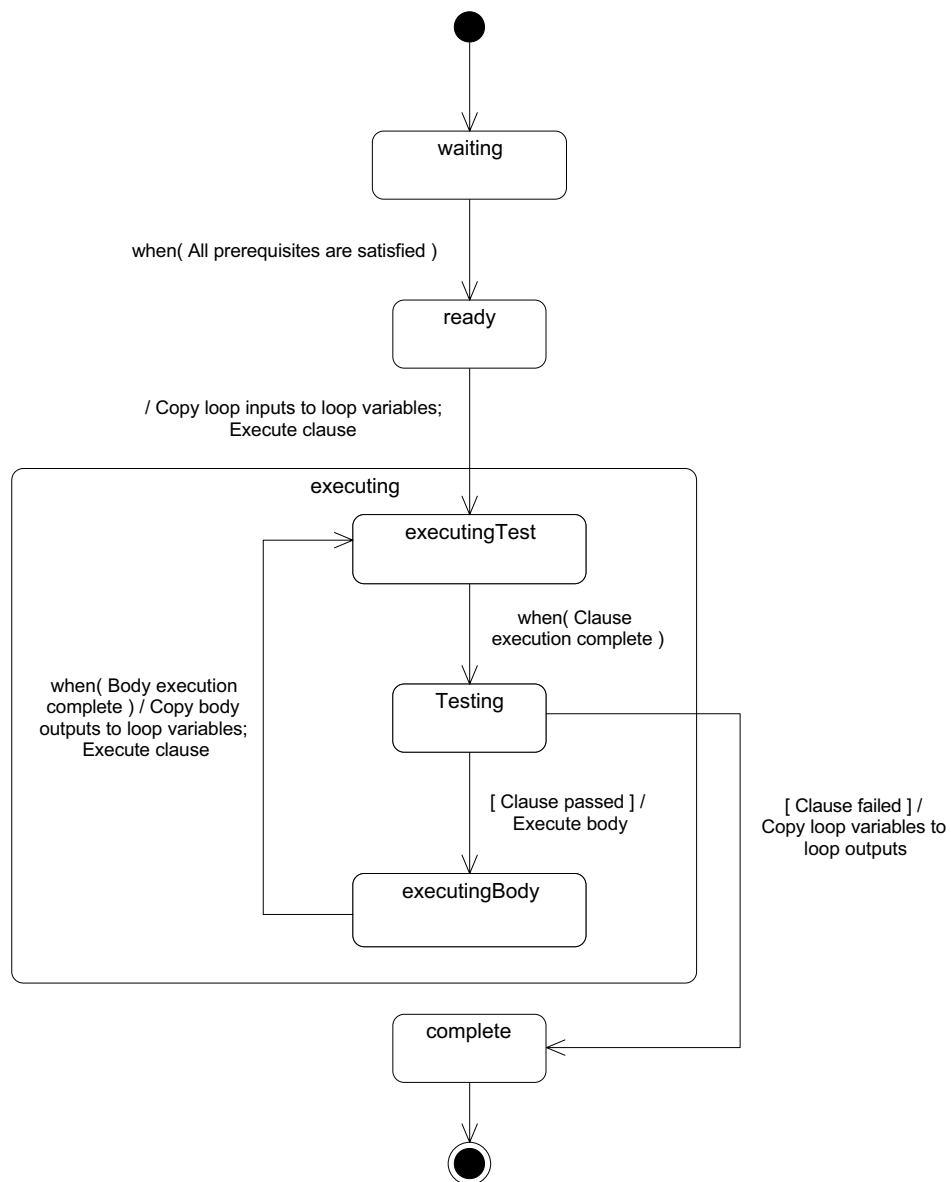


Figure 24. The life cycle for loop-action execution

The iterative nature of a loop action potentially results in a whole sequence of snapshots in its execution history with the status *executing*. These snapshots will cycle between the substatuses *executingTest*, *testing* and *executing-Body* (corresponding to the states in Figure 24), except that the last snapshot with status *executing* will have status *testing* and will be followed by the snapshot with status *complete*

7.3. Composite Action Classes

Clause

A clause is a part of a conditional or loop action. A clause contains a test action and a body action. Both are arbitrary actions (usually group actions) subject to connectivity constraints described under the various composite actions. The execution of the body action is contingent on the corresponding test action producing a “true” value. The clause specifies one output pin of the test action, which must have type Boolean; the body action is only executed if this output produces the value “true” after execution of the test action. (Additionally, for a conditional, only one clause body is executed even if more than one of their tests is true.) The outputs of a condition or loop action are an ordered subset of the outputs of the body action. No other output of a clause may be connected outside the clause.

Clauses within a conditional action may be linked by a set of (noncyclic) predecessor/successor relationships. The test action of a clause may not execute unless all the predecessors of the clause not only have completed execution, but have also “failed”. These relationships thus act, in effect, as specialized kinds of control flows. Neither the test or body actions of a clause can participate in any normal, explicit control flows.

During execution of an enclosing loop action, a test action may not execute for the first time unless all predecessors of the loop action have executed. It may not execute for a subsequent time unless the previous execution of the body action is complete, that is, there is an implicit control flow between the execution of a body action and the next iteration of the test action.

In a conditional or a loop action, a body action may not execute until its test action completes and yields “true”.

Associations

<i>test</i>	The action whose Boolean result (designate by testOutput) must be true for execution of the body action to proceed.
<i>testOutput</i>	The output pin of the test action whose value is the test result. If this value evaluates to “true”, the body action may begin execution.
<i>body</i>	The action whose execution is contingent on the result of the test action being true.
<i>output</i>	The ordered set of outputs of the body that are considered to be results of the clause.
<i>predecessor</i>	The set of clauses that must fail before this clause can execute its test action.
<i>successor</i>	The set of clauses that cannot execute their test actions unless this clause fails.

Additional operations

- [1] This operation returns the transitive closure of all successors of this clause.
 context Clause::allSuccessors() : Set(Clause)
 post: result = self.successor->union(self.successor.allSuccessors()->asSet())

Well-formedness rules

- [1] None of flows attached to the outputs of a clause’s test action may have destinations outside the clause.
 self.test.subactions().availableInputs->union(self.body.subactions().availableInputs)
 ->includesAll(self.test.availableOutput.flow.destination)
- [2] The testOutput of a clause must be an available output of the test action.
 self.test.availableOutput->includes(self.testOutput)
- [3] The testOutput pin must conform to type Boolean and multiplicity 1..1.
 self.testOutput.type = booleanType and
 self.testOutput.multiplicity.range->size = 1 and
 self.testOutput.multiplicity.range->forAll(r : MultiplicityRange | r.lower = 1 and r.upper = 1)

Note. The term “booleanType” is used here to indicate the instance of the class DataType that represents the standard Boolean type. It is not clear what the best way really is to specify such a “global” constant.

- [4] None of the actions within the test action of a clause (if any) may have control-flow connections with actions outside the test action.
`self.test.allSubactions()->forall(action : Action |
action.ancestor.predecessor->union(action.consequent.successor)->forall(a : Action |
a.isSubaction(self.test))`
- [5] None of the actions within the body action of a clause (if any) may have control-flow connections with actions outside the body action.
`self.body->allSubactions()->forall(action : Action |
action.ancestor.predecessor->union(action.consequent.successor)->forall(a:Action |
a.isSubaction(self.body))`
- [6] The test action of a clause may not participate in control flows.
`self.test.ancestor->isEmpty() and self.test.consequent->isEmpty()`
- [7] The body action of a clause may not participate in control flows.
`self.body.ancestor->isEmpty() and self.body.consequent->isEmpty()`
- [8] The outputs of a clause must be available outputs of the body of the clause.
`self.body.availableOutput->includesAll(c.output)`
- [9] There cannot be any cycles in the predecessor/successor relationships among clauses.
`self.allSuccessors()->excludes(self)`

ConditionalAction

A conditional action consists of a set of one or more clauses, exactly one of whose bodies is executed during any execution of the conditional action. If more than one clause has a test that yields “true”, exactly one of the corresponding body actions is selected for execution, but it is unspecified which one. (If the conditional action is determinate, this is an assertion that exactly one concurrent clause test will yield true.) The clause must have an ordered list of output pins that conform to the ordered list of output pins of the conditional. This list must be drawn from accessible outputs of the body action and output pins outside the conditional action that are accessible within it. The only outputs of the conditional action accessible outside it are the output pins directly owned by the conditional action .

Attributes

isDeterminate If true, then whenever the conditional action is executed, the execution of exactly one test action must result in “true”. (This is an assertion, not an executable property. If the assertion is not true, the model is ill formed.)

Associations

clause The set of clauses contained in the conditional action.

Additional operations

- [1] The nested actions of a conditional action are the test and body actions of its clauses.
`context ConditionalAction::subactions() : Set(Action)
post: result = self.clause.test->union(self.clause.body)->asSet()`
- [2] A conditional action has no subactions.
`context ConditionalAction::subactions() : Set(Action)
post: result = Set{ }`

Well-formedness rules

- [1] Each clause of a conditional action must have a number of outputs equal to the number of output pins of the conditional action. Each output of a clause must conform in type and multiplicity to the corresponding output of the conditional.
`self.clause->forall(c : Clause |
c.output->size() = self.outputPin->size())`

```

and Sequence{1..c.output->size()}->forall(i : Integer |
  let cOutput : OutputPin = c.output->at(i) in
  let selfOutput : OutputPin = self.outputPin->at(i) in
    (cOutput.type = selfOutput.type
     or cOutput.type.allParents()->includes(selfOutput.type))
    and cOutput.multiplicity.compatibleWith(selfOutput.multiplicity))

```

- [2] No data flow attached to an available output of the test or body actions of a clause of a conditional action may have a destination outside that action.
 self.clause.test.subactions().availableInput->includesAll(self.clause.test.availableOutput.flow.destination)
 and self.clause.body.subactions().availableInput->includesAll(self.clause.test.availableOutput.flow.destination)
- [3] The predecessors and successors of a clause in a conditional action must be clauses in the same conditional action.
 self.clause->includesAll(self.clause.predecessor->union(self.clause.successor))
- [4] A conditional action owns no input pins.
 self.inputPin->isEmpty()
- [5] The available inputs of a conditional action are the union of the available inputs of the test actions and body actions of all the clauses of the conditional action.
 self.availableInput = self.clause.test.availableInput->union(self.clause.body.availableInput)
- [6] The available outputs of a conditional action are the output pins of the conditional action.
 self.availableOutput = self.outputPin->asSet()
- [7] There may be no path from a conditional action to any action within the conditional action (where a path is defined as in rule [1] for Action and includes both data flows and control flows).
 self.allSuccessors()->excludesAll(self.clause.test.allSubactions()->union(self.clause.body.allSubactions()))

GroupAction

A group action represents a simple composition of a set of subactions. A group action does not own any pins of its own, but data-flow connections may (generally) be made from actions outside the group to pins owned by actions within the group action. The group action as a whole may participate in control flows and actions within the group action may also participate in control flows with actions outside the group action. There is an implicit control flow from the group action to each action within it; this is important only if there is a control flow to the group action. Similarly, there is an implicit control flow from each action in the group action to the group action; this is important only if there is a control flow from the group action. Finally, a set of local variables can be declared in association with a group action. The group action serves as the scope for use of the variables.

Associations

subaction The set of actions contained in the group.
variable The set of variables declared with the group as their scope.

Additional operations

- [1] The nested actions of a group action are its subactions.
 context ConditionalAction::nestedActions() : Set(Action)
 post: result = self.subaction
- [2] A group action has explicit subactions.
 context GroupAction::subactions() : Set(Action)
 post: result = self.subaction

Well-formedness rules

- [1] A group action does not own any pins.
 self.outputPin->isEmpty() and self.inputPin->isEmpty()

- [2] The set of available inputs of a group action is the union of the available inputs of all the subactions of the group action not connected within the group action to an available output of a subaction of the group action.

```
self.availableInput = self.subaction.availableInput->asSet()->reject(i : InputPin |
self.subaction.availableOutput.includes(i.flow.source))
```

- [3] The set of available outputs of a group action is the union of the available outputs of all the subactions of the group action.

```
self.availableOutput = self.subaction.availableOutput->asSet()
```

LoopAction

A loop action contains a single clause whose test action and body action are executed repeatedly as long as the test action yields “true”. A list of output pins acts as “loop variables” for the loop action. The loop variables are not directly available outside the loop. Input pins of the overall loop action provide initial values that are copied into these loop variables before the first iteration of the loop. As output pins, the loop variables may be connected to available inputs of the test and body actions using normal data flows, to provide the “old values” of the loop variables during an iteration. The outputs of the loop clause provide “new values” that are copied to the loop variables at the completion of an iteration. The test is executed after the initial values are copied to the loop variables, and after each execution of the body action. The body action is executed only if the test yields “true”. When the loop terminates, the final values of the loop variables are copied to the regular output pins of the loop action, which are the only available outputs for the loop action as a whole.

Associations

clause The clause that contains the test and body actions for the loop.

loopVariable The set of loop-action output pins that act as loop variables for the loop. These are owned directly by the loop action, but they are not available outside the loop.

Additional operations

- [1] The nested actions of a conditional action are its test and body actions.

```
context ConditionalAction::subactions() : Set(Action)
post: result = Set{self.clause.test, self.clause.body }
```

- [2] A loop action has no subactions.

```
context LoopAction::subactions() : Set(Action)
post: result = Set{ }
```

Well-formedness rules

- [1] The clause of a loop action must have the same number of outputs as the number of loop variables of the loop and each clause output in the ordered list must conform to the corresponding loop variable in type and multiplicity.

```
self.clause.output->size() = self.loopVariable->size()
and Sequence{ 1..clause.output->size() }->forall(i : Integer |
let clauseOutput : OutputPin = clause.output->at(i) in
let v : OutputPin = self.loopVariable->at(i) in
(clauseOutput.type = v.type or clauseOutput.type.allParents()->includes(v.type))
and clauseOutput.multiplicity.compatibleWith(v.multiplicity))
```

- [2] A loop action must have the same number of inputs pins as loop variables and each input pin must conform to the corresponding loop variable in type and multiplicity.

```
self.inputPin->size() = self.loopVariable->size()
and Sequence{ 1..self.inputPin->size() }->forall(i : Integer |
let p : InputPin = self.inputPin->at(i) in
let v : OutputPin = self.loopVariable->at(i) in
(p.type = v.type or p.type.allParents()->includes(v.type))
and p.multiplicity.compatibleWith(v.multiplicity))
```

- [3] A loop action must have the same number of output pins as the number of loop variables and each loop variable in the ordered list must conform to the corresponding output pin (in order) in type and multiplicity.
- ```
self.outputPin->size() =self. loopVariable->size()
 and Sequence{ 1..self.outputPin->size()}->forall(i : Integer |
 let p : OutputPin = outputPin->at(i) in
 let v : OutputPin = loopVariable->at(i) in
 (v.type = p.type or v.type.allParents()->includes(p.type))
 and v.multiplicity.compatibleWith(p.multiplicity))
```
- [4] The clause of a loop action may not have predecessors or successors.
- ```
self.clause.predecessor->isEmpty() and self.clause.successor->isEmpty()
```
- [5] The set of available inputs of a loop action is the union of the loop-action input pins and the available inputs of the test and body actions of the clause of the loop action that are not connected to loop variables.
- ```
self.availableInput = self.inputPin->union(self.clause.test.availableInput
 ->union(self.clause.body.availableInput->reject(i : InputPin | self.loopVariable->includes(i.flow.source)))
```
- [6] The available outputs of a loop action are the output pins of the loop action.
- ```
self.availableOutput = self.outputPin
```
- [7] There may be no path from a loop action to any action within the loop action (where a path is defined as in Rule [1] under Action).
- ```
self.allSuccessors()->excludesAll(self.clause.test.allSubactions()->union(self.clause.body.allSubactions()))
```

## Variable

A variable is the specification of a data slot that represents a local variable shared by the actions within a group. There are actions to write and read variables. These actions are treated as side effect actions, similar to the actions to write and read object attributes and associations. There are no automatic sequencing constraints among actions that access the same variable. Such actions must be explicitly sequenced by control flows (unless their sequencing follows from other constraints anyway).

Any values contained by a variable must conform to the type of the variable and have cardinalities allowed by the multiplicity of the variable.

### Attributes

|                     |                                                                                               |
|---------------------|-----------------------------------------------------------------------------------------------|
| <i>multiplicity</i> | A specification of the number of values a variable execution may hold at any one time.        |
| <i>ordering</i>     | Indicates whether the set of values held by this variable is to be considered ordered or not. |

### Associations

|              |                                                                                                                                                                          |
|--------------|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <i>scope</i> | The group action that owns the variable.                                                                                                                                 |
| <i>type</i>  | A classifier specifying the allowed classifiers of values held by the variable. The actual classifier of a value must conform to the type specification of the variable. |

### Additional operations

- [1] This operations checks whether the given action is within the scope of this variable.
- ```
context Variable::isAccessibleBy(a : Action) : Boolean
post: result = self.scope.nestedActions()->includes(a)
```

7.4. Composite Action Execution Model Classes

ClauseExecutionIdentity

Associations

<i>clause</i>	The clause that is being executed.
<i>testExecution</i>	The identity of the action execution for the test action of the clause.
<i>conditionalExecution</i>	The identity of the execution of the conditional action that contains the clause.

Wellformedness Rules

- [1] A clause execution executes the test action of its clause.
self.testExecution.action = self.clause.test
- [2] The parent of the test execution of a clause-execution identity is the same as the conditional execution.
self.testExecution.parent = self.conditionalExecution

ClauseExecutionSnapshot

Attributes

<i>status</i>	The status of the clause execution. One of: <i>waiting</i> —the control prerequisites for the clause test have not been satisfied <i>ready</i> —the clause test is ready for execution <i>testing</i> —the clause test is executing <i>passed</i> —the clause test resulted in “true”, eligible to execute body <i>failed</i> —the clause test resulted in “false”, successors can be tested
---------------	---

Associations

<i>executionID</i>	(Derived) The identity of this clause-execution snapshot, as a clause-execution identity.
--------------------	---

Additional operation

- [1] This operation returns the set of execution snapshots of the predecessor clauses of the clause being executed.
context ClauseExecutionSnapshot::prerequisites() : Set(ClauseExecutionSnapshot)
post: result = self.executionID.conditionalExecution.clauseExecution
 ->select(c : ClauseExecutionIdentity | self.executionID.clause.predecessor->includes(c.clause))
 ->collect(c : ClauseExecutionIdentity | current(c).oclAsType(ClauseExecutionSnapshot))
- [2] This operation returns the current value of the test output pin of the clause.
context ClauseExecutionSnapshot::testOutputValue() : InstanceIdentity
post: result = current(self.executionID.testExecution).valueOf(self.clause.testOutput)
- [3] This operation asserts that the execution of the test action has been started.
context ClauseExecutionSnapshot::testExecutionStarted() : Boolean
post: result = post(self.executionID.testExecution).isNew()
- [4] This operation asserts that the execution of the test action has completed.
context ClauseExecutionSnapshot::testExecutionStarted() : Boolean
post:
 result = current(self.executionID.testExecution).oclAsType(ActionExecutionSnapshot).status = #complete

Wellformedness Rules

- [1] The identity of a clause-execution snapshot must be a clause-execution identity.
self.identity.oclIsKindOf(ClauseExecutionIdentity) and
self.executionID = self.identity.oclAsType(ClauseExecutionIdentity)
- [2] A clause execution has status *waiting*.

```
self.isNew()
self.status=#waiting
```

Lifecycle

Production 1: A clause execution begins executing its test action when its prerequisites have completed and failed.

Precondition: self.status = #waiting

Postcondition: self.status = #ready and self.prerequisites()->forall(status=#failed) and self.testExecutionStarted()

Production 2: When the test execution is complete, the test output can then be tested.

Precondition: self.status = #ready

Postcondition: self.status = #testing and self.testExecutionCompleted()

Production 3: If the test output of the test execution is true, the clause execution passes.

Precondition: self.status = #testing and self.testOutputValue() = trueValue

Postcondition: self.status = #passed

Production 4: If the test output of the test execution is false, the clause execution fails.

Precondition: self.status = #testing and self.testOutputValue() = falseValue

Postcondition: self.status = #failed

ConditionalExecutionIdentity

Association

clauseExecution The set of identities of the executions of the clauses of the conditional action.

Wellformedness Rules

[1] The action of a conditional-execution identity must be a conditional action.

```
self.action.oclIsKindOf(ConditionalAction)
```

[2] A conditional-execution identity has a unique clause execution for each clause of the conditional action.

```
self.clauseExecution.clause->asSet() = self.action.oclAsType(ConditionalAction).clause
and self.clauseExecution->unique(clause)
```

ConditionalExecutionSnapshot

Attributes

substatus Additional status information on the conditional-action execution, applicable when the status is *executing*. One Of:
executingClauses—the clause tests are being executed
testingClauses—the clause tests have completed execution and the results are being tested
executingBody—the body of a clause that passed its test is being executed

selectedClause The passing clause selected to have its body action executed.

bodyExecution The identity of the action execution for the body action of the selected clause.

conditionalExecutionID

(*Derived*) The identity of the conditional-execution snapshot, as a conditional-execution identity.

Additional operations

[1] A conditional-execution snapshot has no available executions. (The test and body executions cannot be explicit prerequisites to any other action executions.)

```
context ConditionalExecutionSnapshot::nestedExecutions() : Set(ActionExecutionIdentity)
```

```
post: result = Set{ }
```

- [2] This operation returns the current snapshots for the clause executions of this conditional-action execution.
 context ConditionalExecutionSnapshot::clauseSnapshots() : Set(ClauseExecutionSnapshot)
 post: result = self.conditionalExecutionID.clauseExecution->
 collect(e : ClauseExecution | current(e).oclAsType(ClauseExecutionSnapshot))->asSet()
- [3] This operation returns the “pre” snapshots for the clause executions of this conditional-action execution.
 context ConditionalExecutionSnapshot::preClauseSnapshots() : Set(ClauseExecutionSnapshot)
 post: result = self.conditionalExecutionID.clauseExecution->
 collect(e : ClauseExecution | pre(e).oclAsType(ClauseExecutionSnapshot))->asSet()
- [4] This operation asserts that the clause executions have been started.
 context ConditionalExecutionSnapshot::clauseExecutionsStarted() : Boolean
 post: result = self.clauseExecution->forall(e : ClauseExecutionIdentity | new(e))
- [5] This operation asserts that all the clause executions have completed or are unable to execute.
 context ConditionalExecutionSnapshot::clausesExecuted(): Boolean
 post: result = self.clauseSnapshots()->forall(s : ClauseExecutionSnapshot |
 s.status = #passed or s.status = #failed
 or (s.status = #waiting and s.prerequisites()
 ->exists(p : ClauseExecutionSnapshot | p.status = #passed or p.status = #waiting)))
- [6] This operation asserts that the body of the selected clause has been started.
 context ConditionalExecutionSnapshot::bodyExecutionStarted() : Boolean
 post: result = (self.bodyExecution->notEmpty() and self.bodyExecution.action=self.selectedClause.body
 and new(self.bodyExecution))
- [7] This operation asserts that the execution of the body action has completed.
 context ConditionalExecutionSnapshot::bodyExecutionCompleted() : Boolean
 post: result =
 (self.bodyExecution=self.predecessor().oclAsType(ClauseExecutionSnapshot).bodyExecution
 and completed(self.bodyExecution))
- [8] This operation returns the indexed output pin of the clause whose body is being executed.
 context ConditionalExecutionSnapshot::clauseOutput(i : Integer) : OutputPin
 post: result = self.predecessor().oclAsType(ClauseExecutionSnapshot).selectedClause.output->at(i)

Wellformedness Rules

- [1] The identity of a conditional-execution snapshot must be a conditional-execution identity.
 self.identity.oclIsKindOf(ConditionalExecutionIdentity) and
 self.conditionalExecutionID = self.identity.oclAsType(ConditionalExecutionIdentity)
- [2] The parent of the body execution (if any) of a conditional-execution snapshot is the conditional execution.
 self.bodyExecution->forall(parent=self.executionID)

Lifecycle

Production 1: When a conditional-action execution starts executing, it begins executing its clauses.

Precondition: self.status = #ready

Postcondition: self.substatus = #executingClauses and self.clauseExecutionsStarted()

Production 2: When its clauses have completed execution, the conditional-action execution tests the results.

Precondition: self.status = #executingClauses

Postcondition: self.clausesExecuted()
 and self.status = #testingClauses

Production 3: If no clause passes and the conditional action has no output pins, then the execution completes.

Precondition: self.substatus = #testingClauses and self.clauseSnapshots()->forall(status<>#passed)
 and self.executionID.action.outputPins->isEmpty()

Postcondition: self.status = #complete

Production 4: If at least one clause passes, then the conditional-action execution starts executing a body action.

Precondition:

self.substatus = #testingClauses and self.clauseSnapshots()->exists(status=#passed) and
(self.isDeterminate implies self.clauseSnapshots()->select(status=#passed)->size() = 1)

Postcondition: self.substatus = #executingBody
and self.preClauseSnapshots()->exists(s : ClauseExecutionSnapshot |
s.status = #passed and self.selectedClause=s.executionID.clause
and self.bodyExecutionStarted())

Production 5: When the body execution is complete, the clause outputs are copied to the output pins of the conditional action.

Precondition: self.substatus = #executingBody

Postcondition: self.bodyExecutionCompleted()
and self.status = #complete and Sequence{ 1..self.executionID.action.outputPin->size() }
->forall(i : Integer | self.pinValueAt(i) = self.bodySnapshot().valueOf(self.clauseOutput(i)))

GroupExecutionIdentity

Associations

subactionExecution The set of identities of the executions of the subactions of the group action.

variableExecution The set of identities of the executions of variables declared in the group action.

Wellformedness Rules

- [1] The action of a group-execution identity must be a group action.
self.action.oclIsKindOf(GroupAction)
- [2] A group-execution identity must have one unique subaction execution for each subaction of its group action.
self.subactionExecution->unique(action)
and self.subactionExecution.action->asSet{ } = self.action.oclAsType(GroupAction).subaction
- [3] The subaction executions of a group-execution identity have the group-execution identity as their parent.
self.subactionExecution->forall(parent=self)
- [4] A group-execution identity must have one unique variable execution for each variable of its group action.
self.variableExecution->unique(variable)
and self.variableExecution.variable->asSet{ } = self.action.oclAsType(GroupAction).variable

GroupExecutionSnapshot

Associations

groupExecutionID (*Derived*) The identity of the group-execution snapshot, as a group-execution identity.

Additional operations

- [1] The available executions of a group-execution snapshot include all subaction executions and all their available executions.
context GroupExecutionSnapshot::availableExecutions() : Set(ActionExecutionIdentity)
post: result = self.executionID.subactionExecution
->union(self.executionID.subactionExecution.availableExecutions()->asSet())
- [2] This operation asserts that the subaction executions have been started.
context GroupExecutionSnapshot::subactionExecutionsStarted() : Boolean
post: result = self.groupExecutionID.subactionExecution->forall(e : ActionExecutionIdentity | new(e))

- [3] This operation asserts that the subaction executions have completed.
 context GroupExecutionSnapshot::subactionExecutionsCompleted() : Boolean
 post: result = self.groupExecutionID.subactionExecution->forall(e : ActionExecutionIdentity | completed(e))
- [4] This operation asserts that the variable executions have been started.
 context GroupExecutionSnapshot::variableExecutionsStarted() : Boolean
 post: result = self.groupExecutionID.variableExecution->forall(e : VariableExecutionIdentity | new(e))
- [5] This operation asserts that the variable executions have been stopped.
 context GroupExecutionSnapshot::variableExecutionStopped() : Boolean
 post: result = self.groupExecutionID.variableExecution->forall(e : VariableExecutionIdentity | destroyed(e))

Wellformedness Rules

- [1] The identity of a group-execution snapshot must be a group-execution identity.
 self.identity.oclIsKindOf(GroupExecutionIdentity)
 and self.groupExecutionID = self.identity.oclAsType(GroupExecutionIdentity)

Lifecycle

Production 1: When a group executes, it executes all its subactions and variables.

Precondition: self.status = #ready

Postcondition: self.status = #executing and self.subactionExecutionsStarted()
 and self.variableExecutionsStarted()

Production 2: A group execution completes when all its subaction executions have completed, which causes its variable executions to complete.

Precondition: self.status = #executing

Postcondition: self.subactionExecutionsCompleted()
 and self.status = #complete and self.variableExecutionsStopped()

LoopExecutionSnapshot

Associations

<i>substatus</i>	Additional status information on the loop-action execution, applicable when the status is <i>executing</i> . One Of: <i>executingTest</i> —the test action is being executed <i>testing</i> —the test action has completed execution and the result is being tested <i>executingBody</i> —the body action is being executed
<i>testExecution</i>	The identity of the execution of the loop test for the iteration of the loop corresponding to this snapshot.
<i>bodyExecution</i>	The identity of the execution of the loop test for the iteration of the loop corresponding to this snapshot.

Note. Since loop variables are represented as pins, their values are given by pin values, linked to a loop-execution snapshot in the same way as input and output pin values (that is, via the inherited pinValue association). This allows the accessing of loop variables via data flows to be handled uniformly with the accessing of other output pins.

Additional operations

- [1] The only available execution of a loop-execution snapshot is itself (since the loop variables are available to the test and body actions of the loop).
 context GroupExecutionSnapshot::availableExecutions() : Set(ActionExecutionIdentity)
 post: result = Set{self}
- [2] This operation returns the current value of the indexed loop variable.

```
context LoopExecutionSnapshot::currentValueOfVariable(i : Integer) : Sequence(InstanceIdentity)
post: result = self.valueOf(self.executionID.action->oclAsType(LoopAction).loopVariable->at(i))
```

- [3] This operation returns the “pre” value of the indexed loop variable.

```
context LoopExecutionSnapshot::preValueOfVariable(i : Integer) : Sequence(InstanceIdentity)
post: result = self.predecessor().valueOf(self.executionID.action->oclAsType(LoopAction).loopVariable->at(i))
```
- [4] This operation returns the current value of the indexed output of the loop clause.

```
context LoopExecutionSnapshot::clauseOutput(i : Integer) : Boolean
pre: self.bodyExecution->notEmpty()
post: result = current(self.bodyExecution->oclAsType(ActionExecutionSnapshot)
    .valueOf(self.executionID.action.oclAsType(LoopAction).clause.output->at(i)))
```
- [5] This operation asserts that the loop variables have been initialized from the loop inputs.

```
context LoopExecutionSnapshot::loopVariablesInitialized() : Boolean
pre: self.status = #executing
post: result = Sequence{1..self.loopVariableValue->size()}->forall(i : Integer |
    self.currentValueOfVariable(i)=self.valueOf(self.executionID.action.inputPin->at(i)))
```
- [6] This operation asserts that the loop variables have not changed their values.

```
context LoopExecutionSnapshot::loopVariablesUnchanged() : Boolean
pre: self.status = #executing
post: result = Sequence{1..self.loopVariableValue->size()}->forall(i : Integer |
    self.currentValueOfVariable(i)=self.preValueOfVariable(i))
```
- [7] This operation asserts that the loop variables have been updated from the body action outputs.

```
context LoopExecutionSnapshot::loopVariablesUpdated() : Boolean
pre: self.status = #executing
post: result = Sequence{1..self.loopVariableValue->size()}->forall(i : Integer |
    self.currentValueOfVariable(i)=self.clauseOutput(i))
```
- [8] This operation asserts that the loop outputs have been set from the loop variables.

```
context LoopExecutionSnapshot::loopOutputsSet() : Boolean
pre: self.status = #complete
post: result = Sequence{1..self.loopVariableValue->size()}->forall(i : Integer |
    self.valueOf(self.executionID.action.outputPin->at(i))=self.preValueOfVariable(i))
```
- [9] This operation asserts that the execution of the test action has been started.

```
context LoopExecutionSnapshot::testExecutionStarted() : Boolean
post: result = (self.testExecution->notEmpty() and new(self.testExecution))
```
- [10] This operation asserts that the execution of the test action has completed.

```
context LoopExecutionSnapshot::testExecutionCompleted() : Boolean
post: result = (self.testExecution->notEmpty()
    and self.testExecution=self.predecessor().oclAsType(LoopExecutionSnapshot).testExecution
    and completed(self.testExecution))
```
- [11] This operation asserts that the execution of the body action has been started.
Constraints: context LoopExecutionSnapshot::bodyExecutionStarted() : Boolean

```
post: result = (self.bodyExecution->notEmpty() and new(self.bodyExecution))
```
- [12] This operation asserts that the execution of the body action has completed.

```
context LoopExecutionSnapshot::bodyExecutionCompleted() : Boolean
post: result = (self.bodyExecution->notEmpty()
    and self.bodyExecution=self.predecessor().oclAsType(LoopExecutionSnapshot).bodyExecution
    and completed(self.bodyExecution))
```

Wellformedness Rules

- [1] The action of a loop-execution snapshot must be a loop action.

```
self.executionID.action.ocIsKindOf(LoopAction)
```

[2] The test and body executions of a loop-execution snapshot have the the loop execution as their parent.

```
self.testExecution->forAll(parent=self.executionID) and self.bodyExecution->forAll(parent=self.executionID)
```

Lifecycle

Production 1: When a loop action is executing, it must have a value for every loop variable.

Precondition: status=#executing

Assertions: self.pinValue.pin->includes(self.executionID.action.ocAsType(LoopAction).loopVariable)

Production 2: When a loop-action execution starts executing, it begins executing its test action, with the loop inputs copied to the loop variables.

Precondition: self.status = #ready

Postcondition: self.substatus = #executingTest and self.loopVariablesInitialized() and self.testExecutionStarted()

Production 3: When the test action has completed execution, the loop-action execution tests the result.

Precondition: self.substatus = #executingTest

Postcondition: self.testExecutionCompleted()

and self.substatus = #testing and self.loopVariablesUnchanged()

Production 4: If the test action results in false, the loop-action execution completes, with the values of the loop variables copied to the loop output pins.

Precondition: self.substatus=#testing and self.testOutput()=falseValue

Postcondition: self.status=#complete and self.loopOutputsSet()

Production 5: If the test action results in true, the loop-action execution executes its body action.

Precondition: self.substatus=#testing and self.testOutput()=trueValue

Postcondition: self.substatus=#executingBody and self.loopVariablesUnchanged() and self.bodyExecutionStarted()

Production 6: When the body action has completed execution, the loop-action executes the test action again, with the outputs of the body action copied to the loop variables.

Precondition: self.substatus=#executingBody

Postcondition: self.bodyExecutionCompleted()

and self.substatus=#executingTest and self.loopVariablesUpdated() and self.testExecutionStarted()

VariableExecutionIdentity

A variable-execution identity represents a run-time data slot that holds temporary values within a procedure execution.

Associations

variable The variable that is being instantiated by this variable execution.

VariableExecutionSnapshot

A variable-execution snapshot specifies the value of a variable at certain point in time.

Association

value The set of instance identities that represent the value of the variable for this variable-execution snapshot.

executionID (*Derived*) The identity of the variable-execution snapshot, as a variable-execution identity.

Wellformedness Rules

[1] The identity of a variable-execution snapshot must be a variable-execution identity.

```
self.identity.ocIsKindOf(VariableExecutionIdentity)
```

```
and self.executionID = self.identity.ocAsType(VariableExecutionIdentity)
```

- [2] The number of values of a variable-execution snapshot must be consistent with the multiplicity of the variable of the snapshot.
`self.executionID.variable.multiplicity.allows(self.value->size())`

Chapter 8. Read and Write Actions

Objects, attributes, links, and variables have values that are available to actions. Objects have classifiers and existence, attributes and variables have values, links have object ends, qualifier values, and existence that are also available to actions. Read actions get values, while write actions modify values and create and destroy objects and links. Attributes, links, and variables each have their own subsection in this chapter. Read and write actions share the structures for identifying the attributes, links, and variables that they are accessing.

Read actions do not modify the values they access, while write actions, as a general policy, have the most minimal effect possible. For example creating an object does not execute constructors. Languages requiring additional semantics can define higher-level actions on the primitive ones given here. Unlike read actions, write actions may leave semantics unspecified in some cases. No semantics is usually given when an action violates those aspects of static UML modeling that constrain runtime behavior. For example, no semantics is given to creating an instance of an abstract class. The only exception is minimum multiplicity, which is given a semantics equivalent to the lower multiplicity being zero. No semantics is given in situations that require classes as values at runtime. Developers of language bindings can assign their own semantics for undefined cases.

Association actions define semantics for associations that is omitted in UML, and clarify semantics that may be ambiguous or unclear. See Section 8.3 for more information.

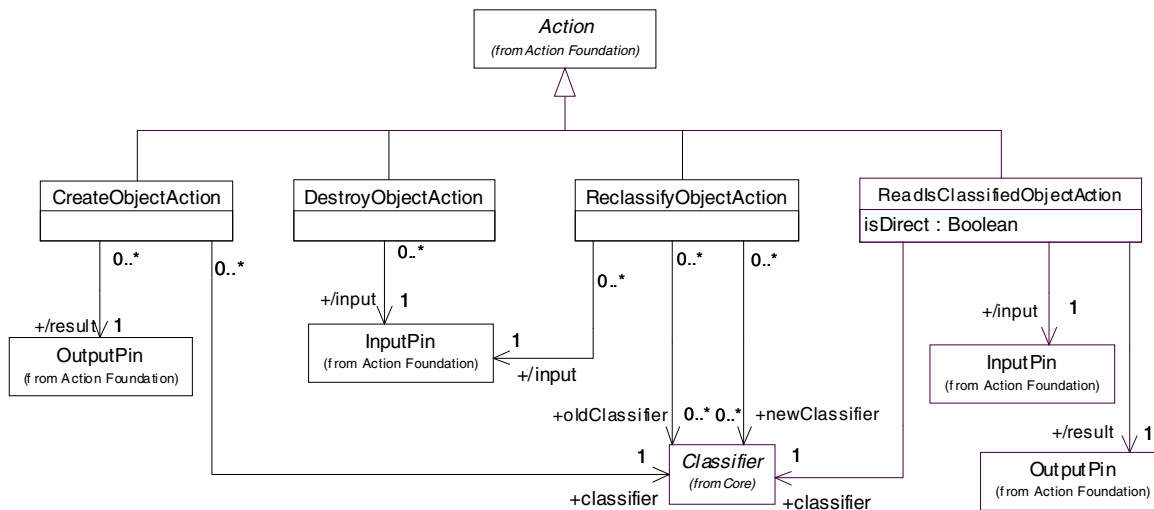


Figure 25. Object Action metamodel

8.1. Object Actions

The only properties an object has directly are its classes and whether it exists or not. All the other aspects of an object are handled through other elements, such as attributes, and associations. This section covers the direct properties of an objects. *CreateObjectAction* requires a classifier to be statically specified, and puts the new object on its output pin at runtime. *DestroyObjectAction* takes the object to be destroyed on its input pin. It includes the semantics of *DestroyLinkAction* when operating on link objects. *ReclassifyObjectAction* requires the classifiers to be added and removed from the object to be specified statically, and the object to reclassify given on its input pin at runtime. *ReclassifyObjectAction* supports adding and removing multiple classifiers at a time, in case some features are inherited from both the old and new classifiers that have values stored in the object. No constructors or destructors are executed by the object actions, nor is there any effect on the state machine of the object. *ReadIsClassifiedObject* determines whether an object is classified by a classifier that is specified at modeling time, either as a direct instance or indirect descendant of the classifier. See descriptions of classes for more information on semantics.

8.2. Attribute Actions

Attributes have values, which may be ordered or unordered if the multiplicity allows more than one value. Figure 26 shows the metamodel for a attribute actions. Both read and write actions use the abstract class *AttributeAction*. It specifies the attribute being accessed, and the particular object on which to access the attribute. The attribute is specified statically by referring to an element in the user model. The object to access is specified dynamically, by referring to an input pin on which the object will be placed at runtime. The type of this pin is the classifier that owns the specified attribute. The multiplicity of the pin is 1..1. When a read-attribute action is executed, the value of the attribute of the input object is placed on the output pin of the action. The type and ordering of the output pin are the same as the specified attribute. All values of the attribute are read regardless of violations in multiplicity.

No semantics is given to adding a value that violates the maximum multiplicity of the attribute. The semantics of removing a value that violates the minimum multiplicity is that same as of the minimum were zero, that is, the value is removed. The insertion point for writing new values in ordered attributes is specified at runtime by an additional input pin, which is required for ordered attributes and omitted for unordered attributes. The insertion point is a positive integer giving the position to insert the value, or the special value *end*, to insert at the end. Reinserting an existing value at a new position moves the value to that position.

The action has no input pin for attributes with an ownerScope of classifier, since the attribute is associated with the classifier itself, rather than any one instance. However, if the classifier has children, it is not defined whether the value is taken from the classifier or its descendants. Semantics is not defined for attributes with targetScope of classifier.

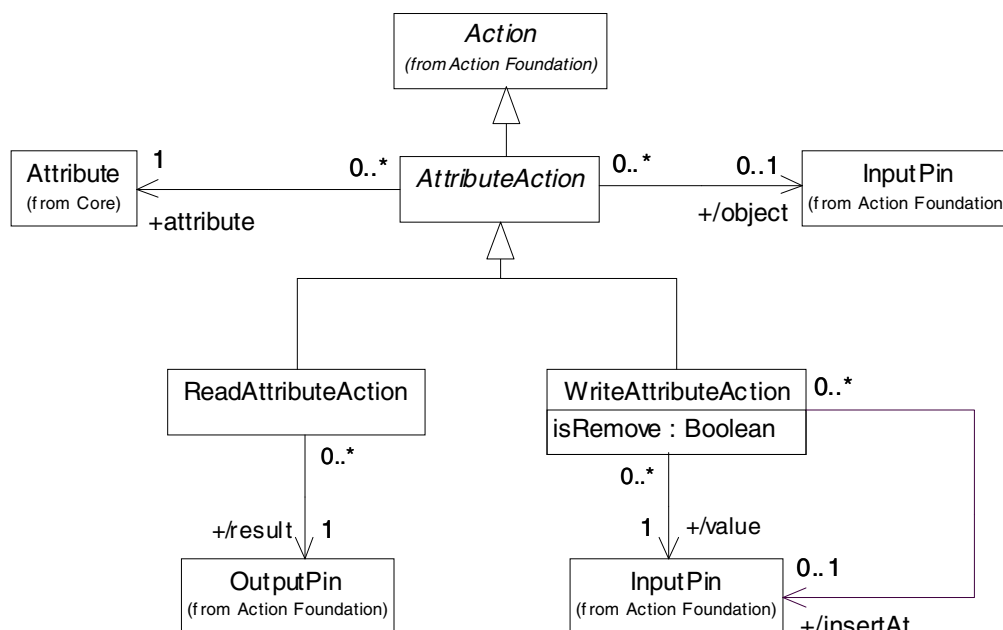


Figure 26. Attribute action metamodel

The attributes of an object at any point in time are determined by the classifiers of that object. The classifiers of an object may change over time due to dynamic classification.

Objects are referenced by their “identity,” which does not change over time. Passing an object as a “value” is really a shorthand for passing a reference to the identity of that object. The attribute specification is part of the metamodel of the action itself, not a value provided dynamically on an input pin.

8.3. Association Actions

Semantics of Associations

Association actions give semantics for areas that are not covered in UML:

Links are not mutable once they are created, except that they can be destroyed and reordered. Qualifier values and end objects cannot be changed once a link is created.

For associations of arity higher than two, the semantics for ordering, changeability, navigability, and visibility of association ends is defined by analogy to n-ary multiplicity: the constraint on an end is applied to the objects on the other association ends, holding those objects and qualifiers on all ends constant. See link action classes for more information.

The multiplicity of qualifier attributes is always 1..1.

Navigability applies only to read actions, and only those that do not operate on link objects.

Visibility applies to both read and write actions, but reading can only be restricted on the end being read, whereas writing can be restricted on all ends. Visibility does not apply to reading link objects. See link action classes for more information.

Conversely, UML gives some semantics for associations that is not covered by this specification:

Associations that have any ends with classifier `targetScope`.

It is noted by this submission that associations with arity greater than two with ends of both classifier and instance `targetScope` are ill-formed. For example, suppose a ternary association has one end with classifier `targetScope` and the others with instance `targetScope`. Those with instance `targetScope` are incorrectly marked as such, because the ends opposite them have both class and instance values. This is an additional well-formedness rule for associations in UML:

```
[1] self.connections→size() = 2 or self.connections→forall(end1, end2 | end1.targetScope = end2.targetScope)
```

Finally, the UML is ambiguous on some semantics of associations that is not defined in this specification either:

Complete semantics is not defined for features and association ends with `ownerScope` of classifier declared on classifiers that have children. It is not defined whether this means a single value for the classifier and all its descendants, or a value for the classifier and each descendant.

Identifying a Link

A link is an instance of an association, but unless the association is a class, the link cannot be passed as a value to or from an action (see). Such a link can only be identified by its end objects and qualifier values. Figure 27 shows the metamodel for identifying a link. These are passed to actions that operate on a link. The *LinkEndData* class gathers together all the information about a single link end. An input pin identifies the end object by being given a value at runtime, and input pins also provide the qualifier values through the *QualifierValue* class. A pin for an end object has the type of the association end and multiplicity of 1..1, since links always have exactly one object at their ends. A pin for a qualifier attribute has a type given by that qualifier, and multiplicity of 1..1. The input pin for object ends is optional, so the metamodel can be used for read actions (see Subsection ””). For write actions, all association ends must have an object input pin, so that all end objects are specified when creating or deleting a link.

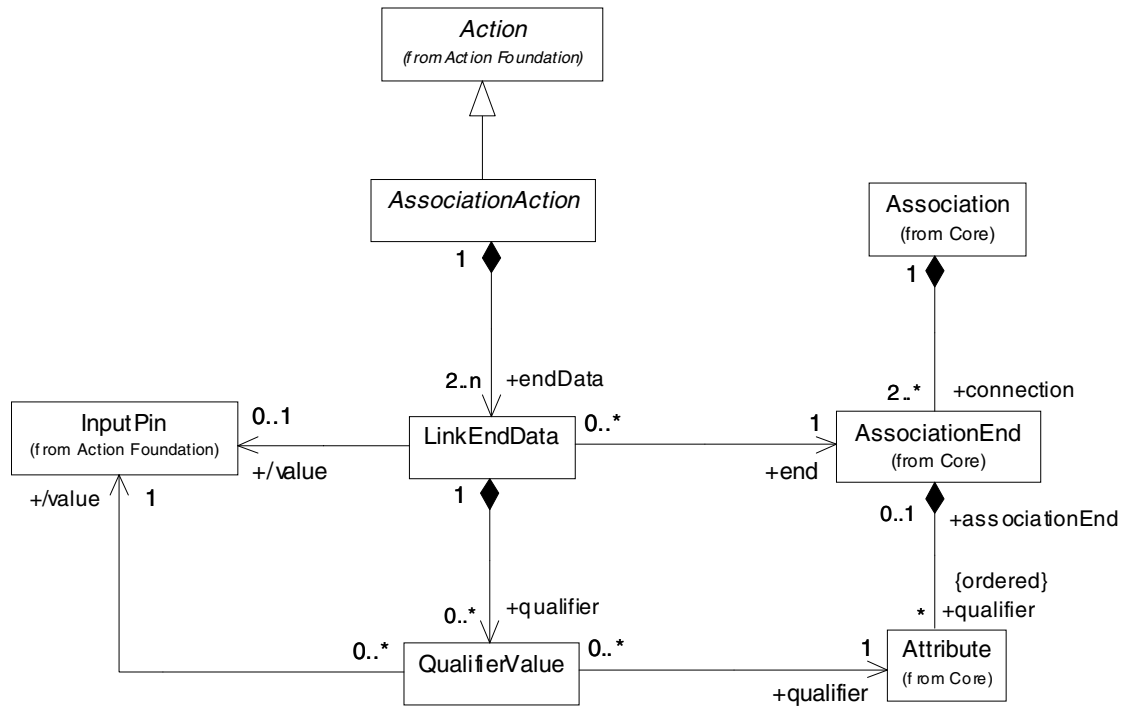


Figure 27. Link identification metamodel

Navigating Across an Association

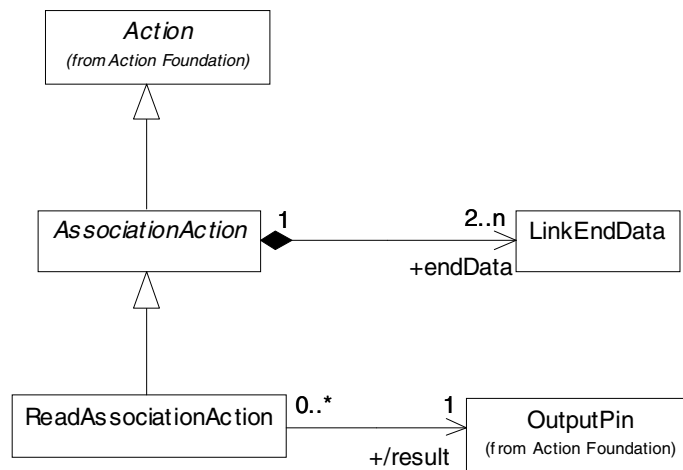


Figure 28. Read-association action metamodel

For associations that are not classes, one can only navigate to a specific end by giving objects for the other ends of the association, and qualifier values for all the ends. The results are the objects at the specified end. These inputs identify

a subset of all the existing links of the association that match the end objects and qualifier values. Returned is a collection of objects for the remaining end (the end being navigated towards), one object from each identified link.

Figure 28 shows the metamodel for a *ReadAssociationAction*. One of the link-end data inherited from *AssociationAction* must have an unspecified object value (the “open” end). The result of the action is a collection of objects on the open end of links of the association, such that the links have the given objects and qualifier values for the other ends and the given qualifier values for the open end. This result is placed on the output pin of the action, which has a type and ordering given by the open end. All objects navigated to are returned regardless of violations of multiplicity. The open end must be navigable, and visible to the host object of the action, in order for semantics to be defined.

The metamodel applies to binary and n-ary associations. Navigation of a binary association is done with a read-association action in which the source end of the link is specified and the target end is not.

Reading Link Objects

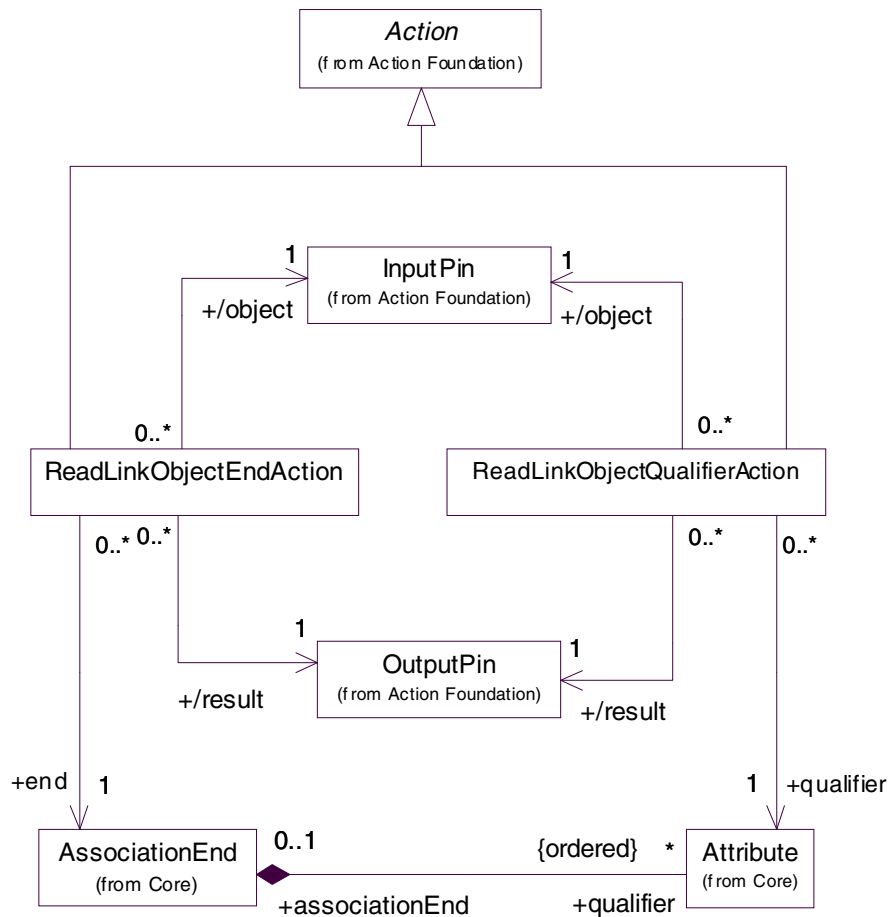


Figure 29. Read link object actions

Link objects are instances of association classes. They have individual identity, since they are objects as well as links. It is possible to read the end objects and qualifier values of a link object in a more direct fashion than links of associations that are not classes, like reading an attribute. Figure 29 shows the metamodel for link object reading actions. Both require a link object as input at runtime, on an input pin that with the type of the association. An association end is given statically to a *ReadLinkObjectEndAction* to specify which end of the link to retrieve the object from. *ReadLinkObjectQualifierAction* determines the association end through a qualifier attribute, which UML restricts to being on exactly one association end. *ReadLinkObjectQualifierAction* returns the value of the qualifier

attribute. In both cases, the input and output pins have multiplicity 1..1. Links have exactly one object at each end in UML. The Action Semantics further specifies that qualifier attributes must have exactly one value. The type of the output pin is the type of the specified association end for *ReadLinkObjectAction*, and the type of the qualifier attribute for *ReadLinkObjectQualifierAction*.

Writing Links

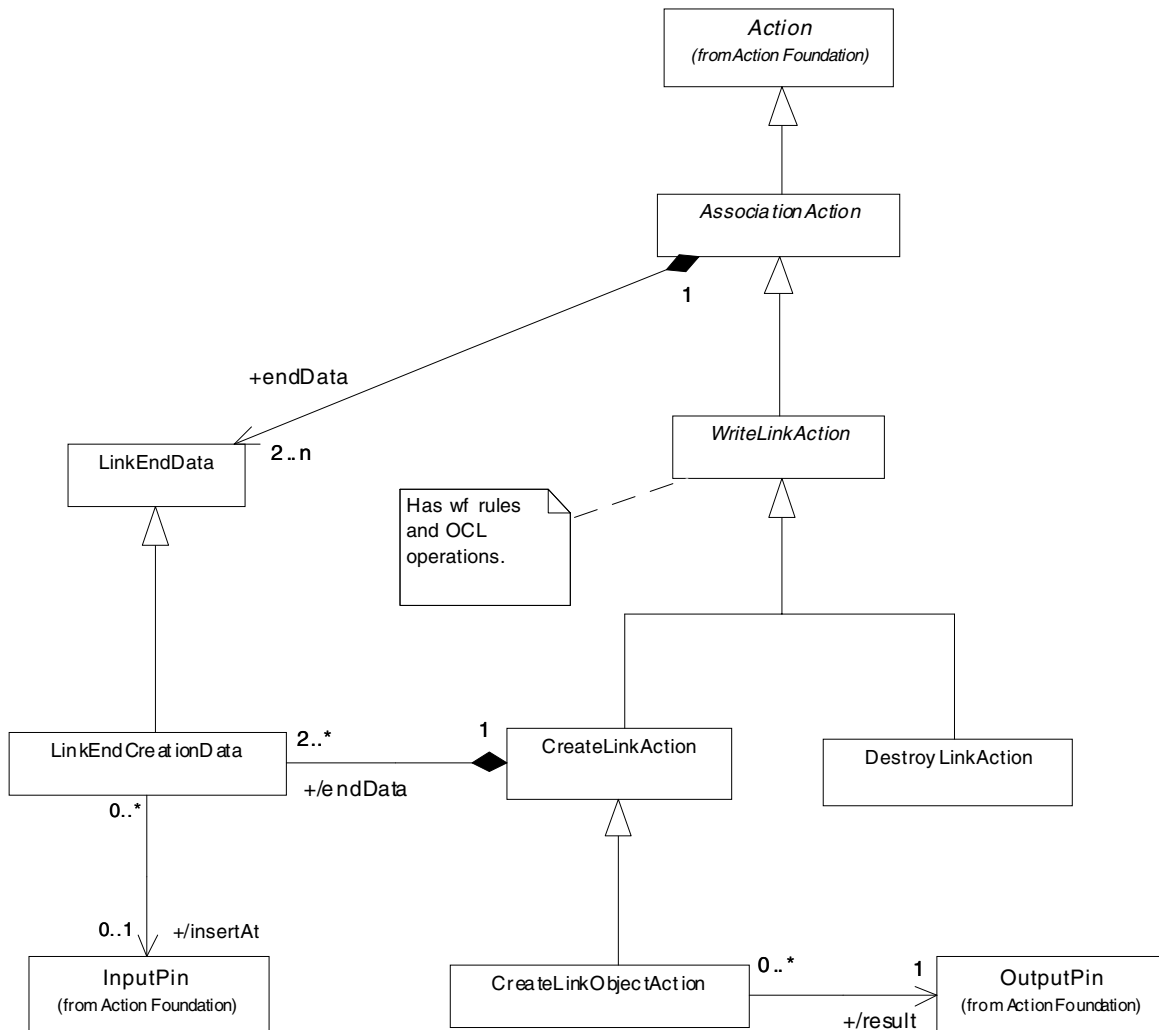


Figure 30. Write link action metamodel

Since links are not mutable, the only write actions that can be taken on them are creation and destruction. Figure 30 shows the classes for creating and destroying links. Both inherit the elements for identifying associations from *AssociationAction* (see Subsection "Identifying a Link"). Both inherit well-formedness rules and OCL operations from *WriteLinkAction*, such as the requirement that all link end data have an input pin, and all ends be visible to the object hosting the action.

CreateLinkAction can be used to create links for associations that are or are not classes, but the created link object is not returned, even for association classes. This allows actions to remain unchanged when their associations change their class status. *CreateLinkObjectAction* is exclusively for creating links of association classes. It returns the created link object. No semantics is given to creating a link that violates the maximum multiplicities of the association.

CreateLinkAction uses a specialization of *LinkEndData* called *LinkEndCreationData*, to support ordered associations. The insertion point is specified at runtime by an additional input pin, which is required for ordered association ends and omitted for unordered ends. The insertion point is a positive integer giving the position to insert the link, or the special value *end*, to insert at the end. Reinserting an existing end at a new position moves the end to that position.

DestroyLinkAction also uses *LinkEndData* to identify associations. It will delete link objects. The semantics of deleting a link that violates the minimum multiplicities of the association is that same as of the minima were zero, that is, the link is destroyed.

8.4. Variable Actions

Variables have values, which may be ordered or unordered if the multiplicity allows more than one value. Figure 31 shows the metamodel for variable actions. Both inherit the variable on which they operate from *VariableAction*, which specifies this statically. The insertion point for new values in ordered variables is specified at runtime by an additional input pin, which is required for ordered variables and omitted for unordered variables. The insertion point is a positive integer giving the position to insert the value, or the special value *end*, to insert at the end. Reinserting an existing value at a new position moves the value to that position. Variable actions can only access variables within the procedure of which the action is a part. The input and output pins have the type, multiplicity, and ordering of the variable.

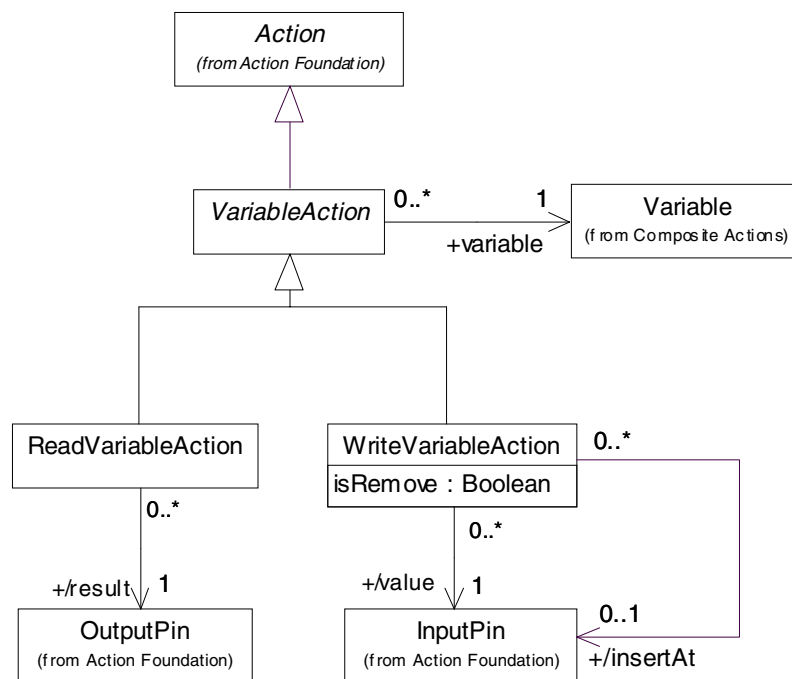


Figure 31. Variable Actions

8.5. Other Actions

Additional actions support navigation to the host object and reading of the extent of a classifier. Figure 32 shows the metamodel for these actions. Every action is ultimately a part of some procedure, which is in turn is attached in some way to the specification of a classifier (e.g., as the body of a method or as part of a state machine). When the procedure executes, it does so in the context of some specific “host” instance of that classifier (see Chapter 6, “Action Foundation”). *ReadSelfAction* produces this host instance on its output pin. The type of the output pin is the classifier

to which the procedure is statically associated. ReadExtentAction reads the current extent of a given classifier. Note that there is no action to read an event or method parameter, since parameters are passed into and out of procedures using pins (see Chapter 5, “State Machine Execution”).

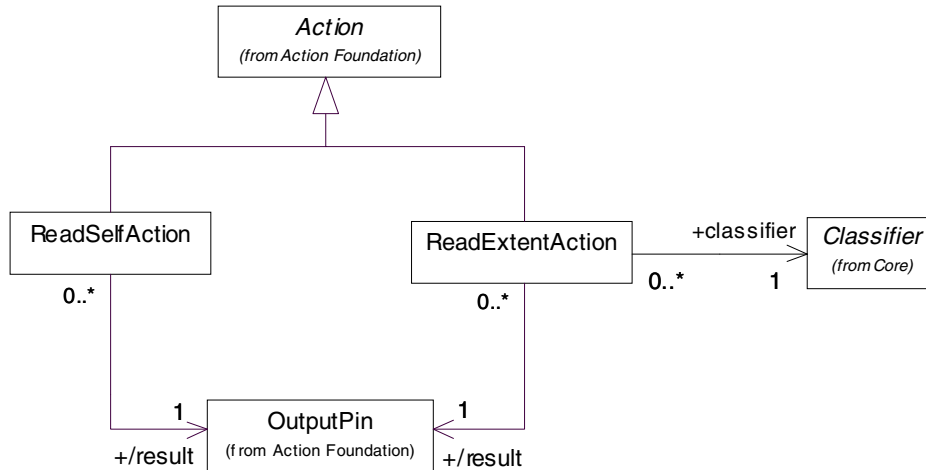


Figure 32. Other action metamodel

Note. The extent of a classifier is the set of all instances of a classifier that exist at any one point in time. However, it is not generally practical to require that reading the extent produce all the instances of the classifier that exist in the entire universe. Rather, any real execution engine will manage only a limited subset of the theoretical extent of any classifier and may actually manage multiple distributed extents for any one classifier. It is not formally specified in general by the execution semantics which managed extent is actually read by a read-extent action.

8.6. Additional OCL Operations for Read and Write Actions

Some additional OCL operations are as follows:

The datatype *SequenceIndex* is defined as a positive integer, or the special value *end*. The operation `<=` on *SequenceIndex* takes an integer as input and determines whether the sequence index is less than or equal to the integer. If the sequence index is *end*, then it is never less than the integer.

```

context SequenceIndex :: <= (i : Integer) : Boolean
post: result = self <= i or not self = #end
  
```

procedure operates on *Action*. It returns the procedure containing the action.

```

context Action::procedure() : Procedure
post: result = if self.Procedure->size() > 0 then self.Procedure else self.group.procedure() endif
  
```

hostClassifier operates on *Procedure*. It returns the classifier hosting the procedure. The navigation between Procedure and host classifier is not formalized yet. It is not the same as the host association between Procedure and InstanceIdentity, because *hostClassifier* returns the classifier on which the procedure is defined as a method or action in a state machine. Also *hostClassifier* will return a value for procedures used by operations with classifier *ownerScope*.

```

context Procedure::hostClassifier() : Classifier
  
```

hostElement operates on *Procedure*. It returns the “innermost” element in the user model that is hosting the procedure. This will be either a *Method*, *Transition*, or *State*. The navigation is not formalized yet. See comments above at *hostClassifier*.

```
context Procedure::hostElement() : ModelElement
```

8.7. Read and Write Action Classes

AssociationAction (abstract)

This class provides metaassociations and well-formedness rules for all association actions. No semantics is defined for associations that have *targetScope* equal to *classifier* on any end.

Associations

```
endData : LinkEndData [2..*]      Data identifying link ends.
```

Well-formedness rules

- [1] The association ends of the link-end data must all be from the same association and include all the association ends of that association.

```
self.endData→forall(end.Association = self.association())
    and self.endData→collect(end)→includesAll(self.association()→collect(connection))
```

- [2] The association ends must have *targetScope* of *instance*.

```
self.endData→forall(end.targetScope = #instance)
```

OCL operations

association operates on *AssociationAction*. It returns the association of the action.

```
context AssociationAction : Association
post: result = self.endData→asSequence().first().end.association
```

allLinkEndObjects operates on *AssociationAction* execution snapshots. It takes a *LinkEndData* as input and returns the *ObjectIdentity*'s participating at that end in links matching the other end objects and all qualifiers as specified by the action execution, for the snapshot in which the operation is called, in order if the link end is ordered.

```
context WriteLinkAction::allLinkEndObjects (targetled : LinkEndData) : Collection
-- Select one other end so it works with read actions and get its input object ...
post: let otherle : LinkEndData=self.executionID.action.endData→select(
    not end = targetleSSd.end)→asSequence→first
    let sourceobject : ObjectIdentity = self.pinValue→select(pin = otherle.value).value in -- ... and link ends.
    let sourcelinkendvalues : Set = sourceobject.linkEnd→select(associationEnd = otherle.end)→select(
    link.linkEndValue→forall(lev |-- All ends of link identity match execution inputs except target.
    let ed : LinkEndData = self.executionID.action.endData→select(end = lev.associationEnd) in
    not targetled→includes(ed) -- Source linked object matches ...
    or  lev.value = self.pinValue→select(pin = ed.value).value)
    and ed.qualifier.forall(edq | lev.qualifierValue→exists(
    attribute=edq.qualifier -- ... qualifiers match on all ends.
    and value = self.pinValue→select(pin = edq.value)))
    let unsortedresultlev : Collection = sourcelinkendvalues→collect(link |
    link.linkEndValue→select(associationEnd = targetled.end)→asSequence→first)) in
    result = if targetled.end.ordering = #unordered
    then unsortedresultlev→collect(value)
    else unsortedresultlev→sortedBy(lev | lev.at())→collect(value)
endif
```

AttributeAction (abstract)

This class provides the general metaassociations and well-formedness rules for all attribute actions. No semantics is defined for accessing an attribute that violates its visibility. No semantics is defined for attributes with `targetScope` equal to `classifier`.

Associations

`attribute` : Attribute [1..1] Attribute to be read.
 <derived> `object` : InputPin [1..1] Derived from `Action:inputPin`. Gives the input pin from which the object whose attribute is to be read or written is obtained.

Well-formedness rules

- [1] If the attribute has a *ownerScope* of *instance*, then the action must have a single input pin. Otherwise the action must have no input pins.
 (self.attribute.ownerScope = #classifier and self.object→size() = 0) or
 (self.attribute.ownerScope = #instance and self.object→size() = 1)
- [2] The attribute must have a *targetScope* of *instance*.
 self.attribute.targetScope = #instance.
- [3] If the action has an input pin, then the type of the input pin is the same as the type of the host object of the attribute.
 self.object→forall(type = self.attribute.owner)
- [4] If the action has an input pin, then its multiplicity must be 1..1.
 self.object→forall(multiplicity.is(1,1))
- [5] Visibility of attribute must allow access to the object performing the action. TBD: OCL needs to handle nested classes.
 let host : Classifier = self.executionID.action.procedure().hostClassifier() in
 self.attribute.visibility = #public
 or host = self.owner.type
 or (self.attribute.visibility = #protected and host.allSupertypes→includes(self.owner.type))

Lifecycle

The lifecycle is specified by subclasses of this action.

Execution Semantics

The semantics is specified by subclasses of this action.

Additional OCL operations

attributeValues operates on *AttributeAction* execution snapshots. It returns the *Identity*'s that are values of the attribute of the object specified by the action execution, for the snapshot in which the operation is called, in order if the attribute is ordered.

```
context AttributeAction::attributeValues() : Collection
post: let unsortedav : Collection = self.pinValue→select(
  pin = self.executionID.action.object).value.attributeValue→select(
  attribute = self.executionID.action.attribute) in
  result = if self.executionID.action.attribute.ordering = #unordered
  then unsortedav→collect(value)
  else unsortedav→sortedBy(av | av.at())→collect(value)
endif
```

isAttributeValueExist operates on *AttributeAction* execution snapshots. It takes a boolean flag indicating whether to test order, if any, and determines whether the attribute value specified by the action execution exists for the snapshot in which the operation is called. -- TBD: Need classifier snapshots to formalize attributes with class ownerscope.

```
context AttributeAction::isAttributeValueExist (isOrderTest : boolean) : boolean
post: let inputValue = self.pinValue→select(pin = self.executionID.action.value).value in
result = self.attributeValues()→includes(inputValue)
and
(not isOrderTest -- ... not testing order ...
or -- ... or ...
self.executionID.action.attribute.ordering = #unordered -- ... the attribute is not ordered ...
or -- ... or ...
(let av : AttributeValue = ... position of the value is the same as insertAt
self.pinValue→select(pin = self.executionID.action.object).value.attributeValue@pre→select(
attribute = self.executionID.action.attribute)→select(value = inputValue)→Sequence→first in
let insertAt : SequenceIndex = self.pinValue→select(pin = self.executionID.action.insertAt).value in
av.at() = insertAt or (insertAt = #end and av.successor→size() = 0)))
```

CreateLinkAction

This class represents the creation of a link or link object for an association.

This action has no output pin, because links are not always values that can be passed to and from actions. The action can be used to create link objects also, but the created object is not returned. This allows actions to remain unchanged when their associations change status.

Creating a link that already exists has no effect. TBD: The formalization of this assumes there are constraints in the link identity model that disallow duplicate *LinkEndValues* on an *ObjectSnapshot*.

No semantics is defined for creating a link for an association that is an abstract class. No semantics is defined for creating a link that violates the upper multiplicity of one of its association ends. No semantics is defined for creating a link that has an association end with changeability *frozen* after initialization of the other end objects, unless the link being created already exists. Objects participating in the association across from an unfrozen end can have links created as long as the objects across from the frozen ends are still being initialized. This means that objects participating in links with 2 or more frozen ends cannot have links created unless all the linked objects are being initialized.

Creating ordered association ends requires an insertion point for a new link using the *insertAt* input pin of *LinkEndCreationData*. The pin is of type *SequenceIndex* with multiplicity of 1..1. A pin value that is a positive integer less than or equal to the current number of links means to insert the new link at that position in the sequence of existing links. A value of *end* for *insertAt* means to insert the new link at the end of the sequence. No semantics is defined for an integer greater than the number of existing links. The *insertAt* input pin does not exist for unordered association ends. Reinserting an existing end at a new position moves the end to that position.

Associations

<derived> endData : *LinkEndCreationData* [2..*] Derived from *AssociationAction*:endData. Specifies ends of association and inputs.

Inputs

endData.insertAt : *SequenceIndex* [1..1] Gives insertion point for ordered association ends. Omitted for unordered ends.

<inherited>endData.value : *ObjectIdentity* [2..*] Inherited from *AssociationAction*:endData. Gives the object at the association end. It is of the same type as the end.

<inherited>endData.qualifier.value : *Identity* [0..*] Inherited from *AssociationAction*:endData. Gives the qualifier value of an association end if the end is qualified. It is the same type as the qualifier

attribute. See Class `LinkEndData`.

Outputs

None.

Well-formedness rules

[1] The association cannot be an abstract class.

```
not (if self.association().oclIsKindOf(Classifier) then (true = self.association().isAbstract) else false endif)
```

Lifecycle

The lifecycle is the same as the generic lifecycle.

Execution Semantics

```
context self: CreateLinkActionExecutionSnapshot
```

Assertion: No production is applicable if any link end is at the maximum multiplicity for its association end in the predecessor snapshot, or if changeability is *frozen* after initialization of the other end objects, unless the link being created already exists. No production is applicable if any association end is ordered and the *insertAt* pin value of the corresponding *LinkEndCreationData* is greater than the number of *LinkEndValue*'s for that end in the predecessor snapshot.

```
(self.existingLink(false)→size() > 0 -- The link to be created either already exists, without order test, or ...
  or self.executionID.action.endData→forall(ed | -- ...no end violates the upper multiplicity...
    self.allLinkEndObjects(ed)→size() < ed.end.multiplicity.upperBound -- ... and ...
    and not ed.end.changeability = #frozen -- ...no end is frozen after initialization of the other end objects.
    or self.executionID.action.endData→forall(oed |
      oed = ed or (self.pinValue→select(pin = oed.value).value).isBeingInitialized()))))
and ...and insertAt is less than or equal to the cardinality for ordered ends.
  self.executionID.action.endData→forall(ed | ed.end.ordering = #unordered
    or ed.insertAt.value <= self.allLinkEndObjects(ed)→size())
```

Production 1: A link for the association exists in the successor snapshot with the specified ends, qualifier values, and insertion point. If the association is an association class, then a corresponding link object identity exists as one of its instances in the successor snapshot. If the link did not exist in the predecessor snapshot, then the link object identity is new in the successor snapshot, has the association class as its single classifier, has no attribute values, and does not participate in any associations.

Precondition:

```
self.status = ready
```

Postcondition:

```
self.status = completed
```

```
and
```

```
self.existingLink(true)→size() > 0 -- Testing order, if any.
```

```
and
```

```
((self.executionID.action.association().oclIsKindOf(Classifier) and self.existingLink(false)@pre→size() = 0)
  implies
```

```
  let link : LinkObjectIdentity = self.existingLink(true)→asSequence→first.oclAsType(LinkObjectIdentity)
```

```
  in
```

```
    link.classifier→size() = 1 and link.classifier→includes(self.executionID.action.association())
```

```
    and link.linkEnd→size() = 0 and link.attributeValue→size() = 0))
```

CreateLinkObjectAction

This class represents the creation of a link object for an association. The action has the same meaning the same as *CreateLinkAction*, except that it is only for creating or finding links on association classes, and it returns the created

or found link object. There are no additional semantics over *CreateLinkAction*, except that the new or found link object is put on the output pin. If the link already exists, then the only effect is that it is put on the output pin.

Associations

<derived> result : OutputPin [1..1] Derived from Action:outputPin. Gives the output pin on which the result is put.

<inherited> endData : LinkEndCreationData [2..*] Inherited from CreateLinkAction. Specifies ends of association and inputs.

Inputs

*<inherited>*endData.value: ObjectIdentity [2..*] Inherited from AssociationAction.endData. Gives object at association end. It is of the same type as the end.

*<inherited>*endData.qualifier.value : Identity [0..*]Inherited from AssociationAction.endData. Gives qualifier values of an association end. They are the same type as the qualifier attribute. See Class LinkEndData.

*<inherited>*endData.insertAt : SequenceIndex [1..1]Inherited from CreateLinkAction. Gives insertion point for ordered association ends. Omitted for unordered ends.

Outputs

result : ObjectIdentity [1..1] The link object created of the same type as the association of the action.

Well-formedness rules

- [1] The association must be an association class.
self.association().oclIsKindOf(Classifier)
- [2] The type of the result pin must be the same as the association of the action.
self.result.type = self.association()
- [3] The multiplicity of the output pin is 1..1.
self.result.multiplicity.is(1,1)

Lifecycle

The lifecycle is the same as the generic lifecycle.

Execution Semantics

context self: CreateLinkObjectActionExecutionSnapshot

Production 1: A link object identity as an instance of the association with the specified ends and qualifier values is put on the output pin.

Precondition:

self.status = ready

Postcondition:

self.status = completed

and self.isMatchingLinkIdentity(self.pinValue→select(pin = self.executionID.action.result).value)

CreateObjectAction

This class represents the instantiation of a classifier. Identity for a new object is created, storage for the object of the given classifier is allocated, and the classifiers of the object are set to the given classifiers. The new object is returned as the value of the action. The action has no other effect. In particular, no constructors are executed, no initial expressions are evaluated, and no state machines transitions are triggered.

No semantics is defined for creating objects from abstract classifiers or association classes.

Associations

class : Classifier [1..1] Classifier to be instantiated.

<derived> result : OutputPin [1..1] Derived from Action:outputPin. Gives the output pin on which the result is put.

Inputs

None.

Outputs

result : ObjectIdentity [1..1] The created object. The type of identity must be the classifier specified for the action.

Well-formedness rules

- [1] The classifier cannot be abstract.
not (self.classifier.isAbstract = true)
- [2] The classifier cannot be an association class
not self.classifier.ocIsKindOf(AssociationClass)
- [3] The classifier of the result pin must be the same as the classifier of the action.
self.result.type = self.classifier
- [4] The multiplicity of the output pin is 1..1.
self.result.multiplicity.is(1,1)

Lifecycle

The lifecycle is the same as the generic lifecycle.

Execution Semantics

context self: CreateObjectActionExecutionSnapshot

Production 1: The result has exactly one classifier in the successor snapshot, which is the same as the classifier specified as a parameter of the action. The result has no attribute values in the successor snapshot and does not participate in any associations.

Precondition:

self.status = ready

Postcondition:

let newidentity : ObjectIdentity = self.pinValue→select(pin = self.executionID.action.result).value in
 self.status = completed and newidentity.new() and newidentity.classifier→size() = 1
 and newidentity.classifier→includes(self.executionID.action.classifier)
 and newidentity.linkEndValue→size() = 0 and newidentity.attributeValue→size() = 0)

DestroyLinkAction

This class represents the destruction of a link, which may be a link object. Link objects can also be destroyed with *DestroyObjectAction*.

The link is specified in the same way as link creation, even for link objects. This allows actions to remain unchanged when their associations are transformed from ordinary ones to association classes and vice versa.

Destroying a link that does not exist has no effect. Destroying a link that has a link object includes the semantics of *DestroyObjectAction*.

No semantics is defined for destroying a link that has an association end with changeability *addonly*, or *frozen* after initialization of the other end objects, unless the link being destroyed does not exist. Objects participating in the association across from an unfrozen end can have links destroyed as long as the objects across from the frozen ends are still being initialized. This means that objects participating in links with 2 or more frozen ends cannot have links destroyed unless all the linked objects are being initialized.

Associations

<inherited> endData : LinkEndData [2..*] Inherited from AssociationAction. Specifies ends of association and inputs.

Inputs

*<inherited>*endData.value: ObjectIdentity [2..*] Inherited from AssociationAction.endData. Gives the object at the association end. It is of the same type as the end.

*<inherited>*endData.qualifier.value : Identity [0..*]Inherited from AssociationAction.endData. Gives the qualifier value of an association end if the end is qualified. It is the same type as the qualifier attribute. They are the same type as the qualifier attribute. See Class LinkEndData.

Outputs

None.

Lifecycle

The lifecycle is the same as the generic lifecycle.

Execution Semantics

context self: DestroyLinkActionExecutionSnapshot

Assertion: No production is applicable if the changeability of any end has changeability *addonly*, or *frozen* after initialization, except if the link does not exist in the predecessor snapshot.

self.existingLink(false)→size() = 0 -- *The link being destroyed does not exist ...*

or -- ... or ...

(self.executionID.action.endData→forall(ed | -- *...no end is frozen after initialization of the other end objects.*
(not ed.end.changeability = #frozen

or self.executionID.action.endData→forall(oed |

oed = ed or (self.pinValue→select(pin = oed.value).value).isBeingInitialized())

and -- *... and no end changeability is addonly.*

not self.executionID.action.endData→exists(end.changeability = #addonly)

Production 1: A link for the association does not exist in the successor snapshot with the specified ends and qualifier values. If the association is an association class, then a corresponding link object identity does not exist as one of its instances in the successor snapshot. If the link existed in the predecessor snapshot, then the link object identity is destroyed in the successor snapshot.

Precondition:

self.status = ready

Postcondition:

self.status = completed and not self.existingLink(false)→size() > 0

and ((self.executionID.action.association().oclIsKindOf(Classifier) and self.existingLink(false)@pre→size() > 0)

implies -- *Need association extents to say that the assoc instance not in extent.*

self.existingLink(false)@pre→asSequence→first.oclAsType(LinkObjectIdentity).destroyed())

DestroyObjectAction

This class represents the destruction of an object, which may be a link object. Link objects can also be destroyed with *DestroyLinkAction*.

The classifiers of the object are removed as its classifiers, storage for the object is deallocated, and the identity of the object is destroyed. The action has no other effect. In particular, no destructors are executed, no state machines transitions are triggered, and references to the objects are unchanged.

Destroying an object that is already destroyed has no effect. Destroying a link object includes the semantics of destroying a link with *DestroyLinkAction*. This is not expressed formally.

Associations

input : InputPin [1..1] The input pin providing the object to be destroyed.

Inputs

input : ObjectIdentity [1..1] The object to be destroyed. There is no restriction on its type.

Outputs

None.

Well-formedness rules

[1] The multiplicity of the input pin is 1..1.

```
self.input.multiplicity.is(1,1)
```

[2] The input pin has no type.

```
self.input.type→size() = 0
```

Lifecycle

The lifecycle is the same as the generic lifecycle.

Execution Semantics

```
context self: DestroyObjectActionExecutionSnapshot
```

Assertion: No production is applicable if the input is not an object identity.

```
self.pinValue→select(pin = self.executionID.action.input).value.ocIsKindOf(ObjectIdentity)
```

Production 1: The input object identity is destroyed in the successor snapshot.

Precondition:

```
self.status = ready
```

Postcondition:

```
self.status = completed and (self.pinValue→select(pin = self.executionID.action.input).value.destroyed())
```

LinkEndCreationData

This class identifies one end of a link to be created by *CreateLinkAction* or *CreateLinkObjectAction*. It is required for specifying ordered association ends. See semantics defined at Class *CreateLinkAction*.

Associations

insertAt : InputPin [0..1] Specifies where the new link should be inserted for ordered association ends, or where an existing link should be moved to. The type of the input is *SequenceIndex*. This pin is omitted for association ends that are not ordered.

Well-formedness rules

[1] Link end data for ordered association ends must have a single input pin for the insertion point with type *SequenceIndex* and multiplicity of 1..1, otherwise the link end data has no input pin for the insertion point.

```
let insertAtPins : Collection = self.insertAt in
```

```
  if self.end.ordering = #unordered
```

```
    then insertAtPins→size() = 0
```

```
    else let insertAtPin : InputPin = insertAts→asSequence→first in
```

```
      insertAtPins→size() = 1 and insertAtPin.type = SequenceIndex and insertAtPin.multiplicity.is(1,1)
```

```
    endif
```

Execution Semantics

See execution semantics defined at Class *CreateLinkAction*.

LinkEndData

This class identifies one end of a link to be read by a read action or written by a write action.

Associations

- end : AssociationEnd [1..1] Association end for which this link-end data specifies values.
- value : InputPin [0..1] Input pin that provides the specified object for the given end. This is omitted if the link-end data specifies an “open” end.
- qualifier : QualifierValue [0..*] Specifies qualifier attribute/value pairs of the given end.

Well-formedness rules

- [1] The association end must be an end of association being read.
self.AssociationAction.association().connection→includes(self.end)
- [2] The qualifiers include all of the qualifiers of the association end.
self.qualifier→collect(qualifier)→includesAll(self.end.qualifier)
- [3] The type of the end object input pin is the same as the type of the association end.
self.value.type = self.end.type
- [4] The multiplicity of the end object input pin must be “1..1”.
self.value.multiplicity.is(1,1)
- [5] The end object input pin is not also a qualifier value input pin.
self.value→excludesAll(self.qualifier.value)

QualifierValue

This class identifies a single qualifier within a link-end data specification.

Associations

- qualifier : Attribute [1..1] Attribute representing the qualifier for which the value is to be specified.
- value : InputPin [1..1] Input pin from which the specified value for the qualifier is taken.

Well-formedness Rules

- [1] The qualifier attribute must be a qualifier of the association end of the link-end data.
self.LinkEndData.end→collect(qualifier)→includes(self.qualifier)
- [2] The type of the qualifier value input pin are the same as the type of the qualifier attribute.
self.value.type = self.qualifier.type
- [3] The multiplicity of the qualifier value input pin is “1..1”.
self.value.multiplicity.is(1,1)

ReadAssociationAction

This class represents navigating an association towards one end. All objects navigated to are returned regardless of violations of multiplicity.

No semantics is defined for reading a link that violates the navigability or visibility of the open association end.

Associations

- <derived> result : OutputPin [0..*] The pin on which are put the objects participating in the association at the end not specified by the inputs.
- <inherited> endData : LinkEndData [2..*] Inherited from AssociationAction. Specifies ends of associa-

tion and inputs.

Inputs

*<inherited>*endData.value: ObjectIdentity [2..*] Inherited from AssociationAction.endData. Gives the object at an association end. It is of the same type as the end. See Class LinkEndData.

*<inherited>*endData.qualifier.value : Identity [0..*]Inherited from AssociationAction.endData. Gives the qualifier value of an association end if the end is qualified. It is the same type as the qualifier attribute. See Class LinkEndData.

Outputs

result : ObjectIdentity [0..*] The objects participating in the association at the end not specified by the inputs. The type of the identities are the same as the type of the open association end. The number of identities is not restricted by the multiplicity of the association end.

Well-formedness Rules

- [1] Exactly one link-end data specification (the “open” end) must not have an end object input pin.
self.endData→select(ed | ed.value→size() = 0)→size() = 1
- [2] The type and ordering of the result output pin are same as the type and ordering of the “open” association end.
let openend : AssociationEnd = self.endData→select(ed | ed.value→size() = 0)→asSequence→first.end in
self.result.type = openend.type
and self.result.ordering = openend.ordering
- [3] The multiplicity of the result output pin is 0..*.
self.result.multiplicity.is(0,#unlimited)
- [4] The ends opposite the open one must be navigable.
let openend : AssociationEnd = self.endData→select(ed | ed.value→size() = 0)→asSequence→first.end in
self.endData→forall(oed | oed.end = openend or oed.end.isNavigable = true)
- [5] Visibility of the open end must allow access to the object performing the action. TBD: OCL needs to handle nested classes.
let host : Classifier = self.executionID.action.procedure().hostClassifier() in
let openend : AssociationEnd = self.endData→select(ed | ed.value→size() = 0)→asSequence→first.end in
openend.visibility = #public
or self.endData→forall(oed | oed.end = openend
or host = oed.end.type
or (openend.visibility = #protected
and
host.allSupertypes→includes(oed.end.type)))

Lifecycle

The lifecycle is the same as the generic lifecycle.

Execution Semantics

context self: ReadAssociationActionExecutionSnapshot

Production 1: The objects put on the result output pin are the ones participating in the association in the predecessor snapshot at the end which has no object input pin, conforming to any specified qualifiers, in order if the end is ordered.

Precondition:

self.status = ready

Postcondition:

self.status = completed

and

```

let openlinkenddata : LinkEndData = self.endData→select(ed | ed.value→size() = 0)→asSequence→first in
let openendobjs : Collection = self.allLinkEndObjects(openlinkenddata)@pre in
  self.pinValue→select(pin = self.executionID.action.result).value =
  if openlinkenddata.end.ordering = #unordered
  then openendobjs.asSet
  else openendobjs.asSequence
endif

```

ReadAttributeAction

This class represents a reading an attribute. All values are read regardless of violations of multiplicity.

Associations

<inherited> attribute : Attribute [1..1] Attribute to be read.

<inherited> object : InputPin [1..1] Inherited from AttributeAction. Gives the input pin from which the object whose attribute is to be read is obtained.

Inputs

object : ObjectIdentity [1..1] Object whose attribute is to be read. The type of the identity is the same as the type of the owner of the attribute.

Outputs

result : Identity [0..*] Value of the attribute. It is of the same type as the attribute. The number of values is not restricted by the multiplicity of the attribute.

Well-formedness Rules

- [1] The type and ordering of the result output pin are the same as the type, multiplicity, and ordering of the attribute.
self.result.type = self.attribute.type
and self.result.ordering = self.attribute.ordering
- [2] The multiplicity of the result output pin is 0..*.
self.result.multiplicity.is(0,#unlimited)

Lifecycle

The lifecycle is the same as the generic lifecycle.

Execution Semantics

context self: ReadAttributeActionExecutionSnapshot

Production 1: The values put on the result output pin are those of the attribute in the predecessor snapshot, in order if the attribute is ordered.

Precondition:

self.status = ready

Postcondition:

self.status = completed

and

```

let attrvals : Collection = self.attributeValues() in
  self.pinValue→select(pin = self.executionID.action.result).value =
  if self.executionID.action.attribute.ordering = #unordered
  then attrvals.asSet
  else attrvals.asSequence
endif

```

ReadExtentAction

This class represents reading the extent of a classifier.

Association

classifier : Classifier [1..1] The classifier whose extent is to be read.

Inputs

None.

Outputs

result : Identity [0..*] The instances of the classifier.

Well-formedness Rules

- [1] The action has no input pins.
self.pinValue→size() = 0
- [2] The type of the result output pin is the classifier.
self.result.type = self.classifier
- [3] The multiplicity of the result output pin is “0..*”.
self.result.multiplicity.is(0,#unlimited)

Lifecycle

The lifecycle is the same as the generic lifecycle.

Execution Semantics

context self: ReadExtentActionExecutionSnapshot

Production 1: The instances of the classifier in the predecessor snapshot are put on the result output pin.

```
self.pinValue→select(pin = self.executionID.action.result).value =
self.executionID.action.classifier.extent.ClassifierSnapshot.instance
```

ReadIsClassifiedObjectAction

This class represents testing an object’s classification, either as a direct instance of , or with intervenining classifiers. It returns true if the specified classifier classifies the input object.

No semantics is defined for testing indirect classification in cases where generalization relationships between classifiers can be modified at runtime.

Attributes

isDirect : Boolean [1..1] Indicates whether the classifier must directly classify the input object.

Associations

classifier : Classifier [1..1] The classifier to test for classification of the input object.

input : InputPin [1..1] The input pin on which to test classification.

result : OutputPin [1..1] The output pin on which the result is put.

Inputs

input : ObjectIdentity [1 ..1] The object on which to test classification. There is no restriction on its type.

Outputs

result : Boolean [1..1] The result of testing the classification of the input object.

Well-formedness rules

[1] The multiplicity of the input pin is 1..1.

self.input.multiplicity.is(1,1)

[2] The input pin has no type.

self.input.type→size() = 0

[3] The multiplicity of the output pin is 1..1.

self.result.multiplicity.is(1,1)

[4] The type of the output pin is Boolean

self.result.type = Boolean

[5] If isDirect is false, then generalization between classifiers must be statically defined.

This rule is not formalized.

Lifecycle

The lifecycle is the same as the generic lifecycle.

Execution Semantics

context self: ReadIsClassifiedObjectActionExecutionSnapshot

Assertion: No production is applicable if the input is not an object identity.

self.pinValue→select(pin = self.executionID.action.input).value.ocIsKindOf(ObjectIdentity)

Production 1: The result is true if the object input to the action is classified by the specified classifier with no intervening classes between the object and the specified classifier. The result is true if the isDirect attribute is false and the object input to the action is classified by the specified classifier, either directly or with intervening classifiers.

Precondition:

self.status = ready

Postcondition:

self.status = completed

and let classifierToTest : Classifier = self.executionID.action.classifier in

let object : Identity = self.pinValue→select(pin = self.executionID.action.input).value in

let result : Boolean = self.pinValue→select(pin = self.executionID.action.input).value in

if object.classifier→includes(classifierToTest)

then result = true

else if isDirect = false and object.classifier→exists(c | c.allSupertypes→includes(classifierToTest))

then result = true

else result = false

endif

ReadLinkObjectEndAction

This class represents reading an end of a link object.

Associations

end : AssociationEnd [1..1] Link end to be read.
 <derived> object : InputPin [1..1] Derived from Action:inputPin. Gives the input pin from which the link object is obtained.

Inputs

object : ObjectIdentity [1..1] Link object being read. The type of the identity is the same as the association owning the association end being read.

Outputs

result : ObjectIdentity [1..1] Object participating in the link at the specified end.

Well-formedness Rules

- [1] The association of the association end must be an association class.
self.end.Association.oclIsKindOf(AssociationClass)
- [2] The ends of the association must all have instance targetScope.
self.end.Association.connections→forall(targetscope = #instance)
- [3] The type of the object input pin is the association class that owns the association end.
self.object.type = self.end.Association
- [4] The multiplicity of the object input pin is “1..1”.
self.object.multiplicity.is(1,1)
- [5] The type of the result output pin is the same as the type of the association end.
self.result.type = self.end.type
- [6] The multiplicity of the result output pin is 1..1.
self.result.multiplicity.is(1,1)

Lifecycle

The lifecycle is the same as the generic lifecycle.

Execution Semantics

context self: ReadLinkObjectEndActionExecutionSnapshot

Production 1: The object put on the result output pin is the one participating in the link in the predecessor snapshot at the end specified.

Precondition:

self.status = ready

Postcondition:

self.status = completed
 and
 self.pinValue→select(pin = self.executionID.action.result).value =
 self.pinValue→select(pin = self.executionID.action.object).value@pre.linkEndValue→select(
 associationEnd = self.executionID.action.end).value

ReadLinkObjectQualifierAction

This class represents reading a qualifier on an end of a link object.

Associations

- qualifier : Attribute [1..1] The attribute that represents the qualifier to be read.
- <derived> object : InputPin [1..1] Derived from Action:inputPin. Gives the input pin from which the link object is obtained.

Inputs

- object : ObjectIdentity [1..1] The link object being read. The type of the identity is the same as the association owning the association end of the qualifier attribute being read.

Outputs

- result : Identity [1..1] Value of the qualifier attribute on its end of the link. The value has the same type as the qualifier attribute.

Well-formedness Rules

- [1] The qualifier attribute must be a qualifier attribute of an association end.
self.qualifier.associationEnd→size() = 1
- [2] The association of the association end of the qualifier attribute must be an association class.
self.qualifier.associationEnd.Association.oclIsKindOf(AssociationClass)
- [3] The ends of the association must all have instance targetScope.
self.qualifier.associationEnd.Association.connections→forall(targetscope = #instance)
- [4] The type of the object input pin is the association class that owns the association end that has the given qualifier attribute.
self.object.type = self.qualifier.associationEnd.Association
- [5] The multiplicity of the object input pin is “1..1”.
self.object.multiplicity.is(1,1)
- [6] The type of the result output pin is the same as the type of the qualifier attribute.
self.result.type = self.qualifier.type
- [7] The multiplicity of the result output pin is “1..1”.
self.result.multiplicity.is(1,1)

Lifecycle

The lifecycle is the same as the generic lifecycle.

Execution Semantics

context self: ReadLinkObjectQualifierActionExecutionSnapshot

Production 1: The qualifier value of the link in the predecessor snapshot at the end specified is put on the result output pin.

Precondition:

self.status = ready

Postcondition:

self.status = completed

and

self.pinValue→select(pin = self.executionID.action.result).value =
self.pinValue→select(pin = self.executionID.action.object).value.linkEndValue@pre→select(lev |
lev.qualifierValue→select(attribute = self.executionID.action.qualifier)).qualifierValue.value

ReadSelfAction

This class represents getting the host object of an action.

Associations

<derived> result : OutputPin [1..1] Derived from Action:outputPin. Gives the output pin on which the hosting object is placed.

Inputs

None.

Outputs

result : ObjectIdentity [1..1] Object hosting the action.

Well-formedness Rules

- [1] The action must be contained in a procedure that has a host classifier.
self.procedure().hostClassifier()→size() = 1
- [2] The action must be contained in a procedure that is acting as the body of a method with an ownerScope of instance.
let hostelement : Element = self.procedure().hostElement() in
 not hostelement.oclIsKindOf(Method)
 or hostelement.oclAsType(Method).ownerScope = #instance
- [3] The action may not be contained in a procedure that has a host classifier with the stereotype «utility».
self.procedure().hostClassifier().stereotype→forall(s | not s.name = “utility”
 and not s.allSuperTypes→exists(name = “utility”))
- [4] The type of the result output pin is the host classifier.
self.result.type = self.procedure().hostClassifier()
- [5] The multiplicity of the result output pin of a read-self action is “1..1”.
self.result.multiplicity.is(1,1)

Lifecycle

The lifecycle is the same as the generic lifecycle.

Execution Semantics

context self: ReadSelfActionExecutionSnapshot

Production 1: The object hosting the action is put on the result output pin.

Precondition:

self.status = ready

Postcondition:

self.status = completed

and self.pinValue→select(pin = self.executionID.action.result).value = self.context.host

ReadVariableAction

This class represents reading a variable.

Associations

<inherited> variable : Variable [1..1] Variable being read.

<derived> result : OutputPin [1..1] Derived from Action:outputPin. Gives the output pin on which the val-

ues of the variable are placed.

Outputs

result : Identity [0..*] Value of the variable. The type of the value is the same as the type of the variable.
The number of values is restricted by the multiplicity of the variable.

Well-formedness Rules

[1] The type, multiplicity, and ordering of the result output pin of a read-variable action are the same as the type, multiplicity, and ordering of the variable.

```
self.result.type = self.variable.type
and self.result.multiplicity = self.variable.multiplicity
and self.result.ordering = self.variable.ordering
```

Lifecycle

The lifecycle is the same as the generic lifecycle.

Execution Semantics

context self: ReadVariableActionExecutionSnapshot

Production 1: The value of the variable in the predecessor snapshot is put on the result output pin.

Precondition:

```
self.status = ready
```

Postcondition:

```
self.status = completed
and self.pinValue→select(pin = self.executionID.action.result).value = self.getVariableValues
```

ReclassifyObjectAction

This class represents changing classifiers for an object. Identity for the object is preserved, storage for the object is changed, and the specified change of classifiers is made. The action has no other effect. In particular, no constructors or destructors are executed and no initial expressions are evaluated. New classifiers can replace existing classifiers in one action, so that information in attributes is not lost by intermediate stages of classification.

Adding a classifier that duplicates one already existing, or removing a classifier that is not there, has no effect. Adding and removing the same classifier has no effect.

No semantics is defined if any of the new classifiers are abstract. No semantics is defined if all classifiers are removed from an identity.

Associations

<i><derived></i> input : ObjectIdentity [1..1]	Derived from Action:inputPin. Gives the object to be reclassified.
newClassifier : Classifier [0..*]	Classifiers to add to the classifiers of the object.
oldClassifiers : Classifier [0..*]	Classifiers to remove from classes of the object.

Inputs

input : ObjectIdentity [1..1] *Object to be reclassified.*

Outputs

None.

Well-formedness rules

- [1] None of the new classifiers may be abstract.
not self.newClassifier→exists(isAbstract = true)
- [2] The multiplicity of the input pin is 1..1.

```
self.input.multiplicity.is(1,1)
```

[3] The input pin has no type.

```
self.input.type→size() = 0
```

Lifecycle

The lifecycle is the same as the generic lifecycle.

Execution Semantics

```
context self: ReclassifyObjectActionExecutionSnapshot
```

Assertion: No production is applicable if the input is not an `ObjectIdentity` or all the classifiers of the input object are removed.

```
self.pinValue→select(pin = self.executionID.action.input).value.ocIsKindOf(ObjectIdentity)
```

```
and self.resultClassifiers()→size() > 0
```

Production 1: The object input to the action is classified by the classifiers in the predecessor snapshot plus the new classifiers and minus the old classifiers.

Precondition:

```
self.status = ready
```

Postcondition:

```
self.status = completed
```

```
and self.pinValue→select(pin = self.executionID.action.input).value.classifier = self.resultClassifiers()
```

Additional OCL operations

resultClassifiers operates on *ReclassifyObjectAction* execution snapshots. It returns the classifiers of the identity being reclassified plus the new classifiers minus the old classifiers, as specified in the action.

```
context ReclassifyObjectAction::resultClassifiers(): Set
```

```
post: result = self.pinValue→select(pin = self.executionID.action.input).value.
```

```
classifier@pre→union(self.executionID.action.newClassifier)
```

```
- self.executionID.action.oldClassifier
```

VariableAction (abstract)

This class provides metaassociations and well-formedness rules for all variable actions.

Associations

```
variable : Variable [1..1]      Variable being accessed.
```

Well-formedness rules

[1] The action must be in the scope of the variable.

```
self.variable.isAccessibleBy(self)
```

Lifecycle

The lifecycle is specified by subclasses of this action.

Execution Semantics

The semantics is specified by subclasses of this action.

Additional OCL operations

getVariableValues operates on *VariableAction* execution snapshots. It returns the values of the variables in the context of the action execution corresponding to the variable of the action, in the snapshot in which the operation is called.

```
context WriteVariable::getVariableValues() : Sequence
```

```
post: result = self.context.variableExecution→select(variable = self.executionID.action.variable).value
```

isVariableValueExist operates on *VariableAction* execution snapshots. It takes boolean flag indicating whether to test order and determines whether the variable value specified by the action execution exists for the snapshot in which the operation is called.

```
context WriteVariableAction::isVariableValueExist (isOrderTest : boolean) : Boolean
post: result = self.getVariableValues→includes(self.pinValue→select(
  pin = self.executionID.action.value).value)
  and
  (not isOrderTest
  or
  self.pinValue→select(pin = self.executionID.action.insertAt).value
```

WriteAttributeAction

This class represents the modification of attribute values. Creating an attribute value that already exists has no effect. Removing an attribute value that does not exist has no effect.

No semantics is defined for adding a value that violates the upper multiplicity of the attribute. No semantics is defined for removing an existing value for an attribute with changeability *addonly*. No semantics is defined for adding a new value or removing an existing value for an attribute with changeability *frozen* after initialization of the owning object.

Adding values to ordered attributes requires an insertion point for a new value using the *insertAt* input pin of *WriteAttributeAction*. The pin is of type *SequenceIndex*. A positive integer less than or equal to the current number of values means to insert the new value at that position in the sequence of existing values. A value of *end* for *insertAt* means to insert the new value at the end of the sequence. No semantics is defined for an integer greater than the number of existing values. The *insertAt* input pin does not exist for unordered attributes. Reinserting an existing value at a new position moves the value to that position.

Attributes

isRemove : Boolean [1..1] Tells whether the action is adding or removing a value from the attribute.

Associations

value : InputPin [1..1] The attribute value being added or removed.

<derived> insertAt : InputPin [0..1] Derived from Action:inputPin. Gives the position at which to insert a new value or move an existing value in ordered attributes. Omitted for unordered attributes.

<derived> object : InputPin [1..1] Derived from Action:inputPin. Gives the input pin from which the object whose attribute is to be written is obtained.

<inherited> attribute : Attribute [1..1] Inherited from AttributeAction. Gives the attribute to be read.

Inputs

value : Identity [1..1] Value of attribute to add or remove. Its type is the same as the type of the attribute.

object : ObjectIdentity [1..1] Object to access. Its type is the same as the type of the owner of the attribute being written.

Outputs

None.

Well-formedness rules

[1] The type input pin is the same as the owner of the attribute.
self.value.owner = self.attribute.type

- [2] The multiplicity of the input pin is 1..1
`self.value.multiplicity.is(1,1)`
- [3] An action adding a value on ordered attributes must have a single input pin for the insertion point with type *SequenceIndex* and multiplicity of 1..1, otherwise the action has no input pin for the insertion point.
`let insertAtPins : Collection = self.insertAt in`
 `if self.attribute.ordering = #unordered or isRemove = false`
 `then insertAtPins→size() = 0`
 `else let insertAtPin : InputPin = insertAts→asSequence→first in`
 `insertAtPins→size() = 1`
 `and insertAtPin.type = SequenceIndex`
 `and insertAtPin.multiplicity.is(1,1)`
 `endif`

Lifecycle

The lifecycle is the same as the generic lifecycle.

Execution Semantics

context self: WriteAttributeActionExecutionSnapshot

TBD: The semantics for classifier ownerscope is not formalized yet.

Assertion: No production is applicable if *isRemove* is *false* and the attribute is at the maximum multiplicity in the predecessor snapshot, or the attribute changeability is *frozen* after initialization of the owning object, unless the value being added already exists. No production is applicable if *isRemove* is *false* and the attribute is ordered and the *insertAt* attribute is greater than the number of values in the predecessor snapshot. No production is applicable if *isRemove* is *true* and attribute changeability is *addonly*, or the attribute changeability is *frozen* after initialization, unless the value being removed does not exist.

```
(self.isRemove = false
and (self.isAttributeValueExist(false)
or (self.attributeValues()→size() < self.executionID.action.attribute.multiplicity.upperBound
and (not self.executionID.action.attribute.changeability = #frozen
or (self.pinValue→select(pin = ed.object).value).isBeingInitialized()))
-- and insertAt is less than or equal to the cardinality for ordered attributes
and (self.executionID.action.attribute.ordering = #unordered
or self.pinValue→select(pin = self.executionID.action.insertAt).value <= self.attributeValues()→size()))
or
(self.isRemove = true
and (not self.isAttributeValueExist(false)
or (not self.executionID.action.attribute.changeability = #addonly
and (not self.executionID.action.attribute.changeability = #frozen
or (self.pinValue→select(pin = ed.object).value).isBeingInitialized()))))
```

Production 1: The value specified for the attribute exists in the successor snapshot if *IsRemove* is *false*, in the position specified if any, otherwise it does not.

Precondition:

`self.status = ready`

Postcondition:

```
self.status = completed
and ((self.isRemove = true and not self.isAttributeValueExist(false))
or
(self.isRemove = false and self.isAttributeValueExist(true))
```

WriteLinkAction (abstract)

This class provides the general metaassociations and well-formedness rules for all link actions. No semantics is defined for writing a link that violates the visibility of any association end.

Associations

<inherited> endData : LinkEndData [2..*] Inherited from AssociationAction. Data identifying link ends.

Well-formedness rules

[1] All end data must have exactly one input object pin.

```
self.endData.forall(value→size() = 1)
```

[2] Visibility of all ends must allow access to the object performing the action. TBD: OCL needs to handle nested classes.

```
let host : Classifier = self.executionID.action.procedure().hostClassifier() in
self.endData→forall(ed | ed.end.visibility = #public
  or (self.endData→forall(oed | oed = ed
    or host = oed.end.type
    or (ed.end.visibility = #protected
      and host.allSupertypes→includes(oed.end.type))))
```

Lifecycle

The lifecycle is specified by subclasses of this action.

Execution Semantics

The semantics is specified by subclasses of this action.

Additional OCL operations

isMatchingLinkIdentity operates on *WriteLinkAction* execution snapshots. It takes a *LinkIdentity* as input, and a boolean flag indicating whether to test order, if any. It determines whether the link identity matches the link specified by the write link action. Compare to *allLinkEndObjects*.

```
context WriteLinkAction::isMatchingLinkIdentity(li : LinkIdentity, isOrderTest : Boolean) : Boolean
post: result = li.linkEndValue→forall(lev |-- All ends of link identity match corresponding execution inputs.
  let ed : LinkEndData = self.executionID.action.endData→select(end = lev.association-
```

End) in

```
  lev.value = self.pinValue→select(pin = ed.value).value -- Linked object matches ...
  and -- ... and ...
  ed.qualifier.forall(edq | lev.qualifierValue→exists(
    attribute=edq.qualifier -- ... qualifier matches.
    and
    value = self.pinValue→select(pin = edq.value))
  and -- ... and ...
  (not isOrderTest -- ... not testing order ...
  or -- ... or ...
  lev.associationEnd.ordering = #unordered -- ... the link end is not ordered ...
  or -- ... or ... position of the link end object is the same as insertAt
  lev.at() = ed.insertAt.value or (ed.insertAt.value = #end and lev.successor→size() = 0)))
```

existingLink operates on *WriteLinkAction* execution snapshots. It takes a boolean flag indicating whether to test order, if any. It returns a set that contains the link identity specified by the action execution, or an empty set of the link does not exist.

```
context WriteLinkAction::existingLink(isOrderTest : Boolean) : Set
post: let endData1 : LinkEndData = self.executionID.action.endData→asSequence→first in
```

```

let inputobject : ObjectIdentity = self.pinValue→select(pin = endData1.value).value in
let linkendvalues : Set = inputobject.linkEnd→select(associationEnd = endData1.end) in
result = linkendvalues→select(self.isMatchingLinkIdentity(link, isOrderTest))→collect(link)

```

isBeingInitialized operates on *ObjectSnapshot*. It determines whether the objectsnapshot is referring to a point in time when the object is being initialized. No OCL is given because initialization is not formalized.

```
context ObjectSnapshot::isBeingInitialized() : Boolean
```

WriteVariableAction

This class represents the modification of a variable. Variables are potentially multi-valued. This action supports adding and removing values from a variable. Creating a variable value that already exists has no effect. Removing an variable value that does not exist has no effect.

Adding values to ordered variables requires an insertion point for a new value using the *insertAt* input pin of *WriteVariableAction*. The pin is of type *SequenceIndex*. A positive integer less than or equal to the current number of values means to insert the new value at that position in the sequence of existing values. A value of *end* for *insertAt* means to insert the new value at the end of the sequence. No semantics is defined for an integer greater than the number of existing values. The *insertAt* input pin does not exist for unordered variables. Reinserting an existing value at a new position moves the value to that position.

No semantics is defined for adding a value that violates the upper multiplicity of the variable.

Attributes

isRemove : Boolean [1..1] Tells whether the action is adding or removing a value from the variable.

Associations

<derived> value : InputPin [1..1] Derived from Action:inputPin. Value to be added or removed from the variable.

<derived> insertAt : InputPin [0..1] Derived from Action:inputPin. The input pin giving the position at which to insert values into an ordered variable. Omitted for unordered variables.

<inherited> variable : Variable [1..1] Inherited from Action:inputPin. Gives the variable being written.

Inputs

value : Identity [1..1] Value to add or remove. Its type is the same as the type of the variable.

insertAt : SequenceIndex [0..1] The position at which to insert values into an ordered variable. Omitted for unordered variables.

Outputs

None.

Well-formedness rules

[1] The type of the input pin is the same as the type of the variable.

```
self.value.type =self.variable.type
```

[2] The multiplicity of the input pin is 1..1.

```
self.value.multiplicity.is(1,1)
```

[3] Write actions on ordered variables must have a single input pin for the insertion point with type *SequenceIndex* and multiplicity of 1..1, otherwise the action has no input pin for the insertion point.

```

let insertAtPins : Collection = self.insertAt in
if self.variable.ordering = #unordered
then insertAtPins→size() = 0
else let insertAtPin : InputPin = insertAts→asSequence→first in
insertAtPins→size() = 1

```

```

and insertAtPin.type = SequenceIndex
and insertAtPin.multiplicity.is(1,1)
endif

```

Lifecycle

The lifecycle is the same as the generic lifecycle.

Execution Semantics

```
context self: WriteVariableActionExecutionSnapshot
```

Assertion: No production is applicable if *isRemove* is *false* and the variable is at the maximum multiplicity in the predecessor snapshot, unless the value being added already exists. No production is applicable if the variable is ordered and the *insertAt* attribute is greater than the number of values in the predecessor snapshot.

```

self.isRemove = true
or ((self.isVariableValueExist(false)
    or self.getVariableValues→size() < self.executionID.action.variable.multiplicity.upperBound))
    -- and insertAt is less than or equal to the cardinality for ordered variables
and (self.executionID.action.variable.ordering = #unordered
    or self.pinValue→select(pin = self.executionID.action.insertAt).value<= self.getVariableValues→size()))

```

Production 1: The value specified for the variable exists in the successor snapshot.

Precondition:

```
self.status = ready
```

Postcondition:

```

self.status = completed
and ((self.isRemove = true and not self.isVariableValueExist(false))
    or (self.isRemove = false and self.isVariableValueExist(true)))

```


Chapter 9. Computation Actions

These actions transform a set of input values to produce a set of output values. These actions work on the input values only to produce output values, so they embody mathematical functions. They do not read or write attribute or link values, nor do they otherwise interact with object memory or other objects, so their control is entirely self-contained. These actions supply the primitive functions out of which computations are constructed.

9.1. Computation actions

Computation actions evaluate various mathematical functions. They take input values and produce output values. The output values depend only on the input values and not on the state of the memory or the state of the control.

This submission does not define a set of primitive functions. Rather, we assume that any particular implementation of the action semantics will define a set of primitive functions. For modeling purposes, users must be able to define new primitive functions, but the mechanisms of the definition are outside of the UML and action semantics. All that is required is for each primitive function to have a name and lists of input and output types.

This approach has the drawback that users must agree on the set of primitive functions, but it is almost certain that different groups of users would prefer different sets of functions anyway, so little is lost in not providing a default set of functions.

The following model shows the computation action classes

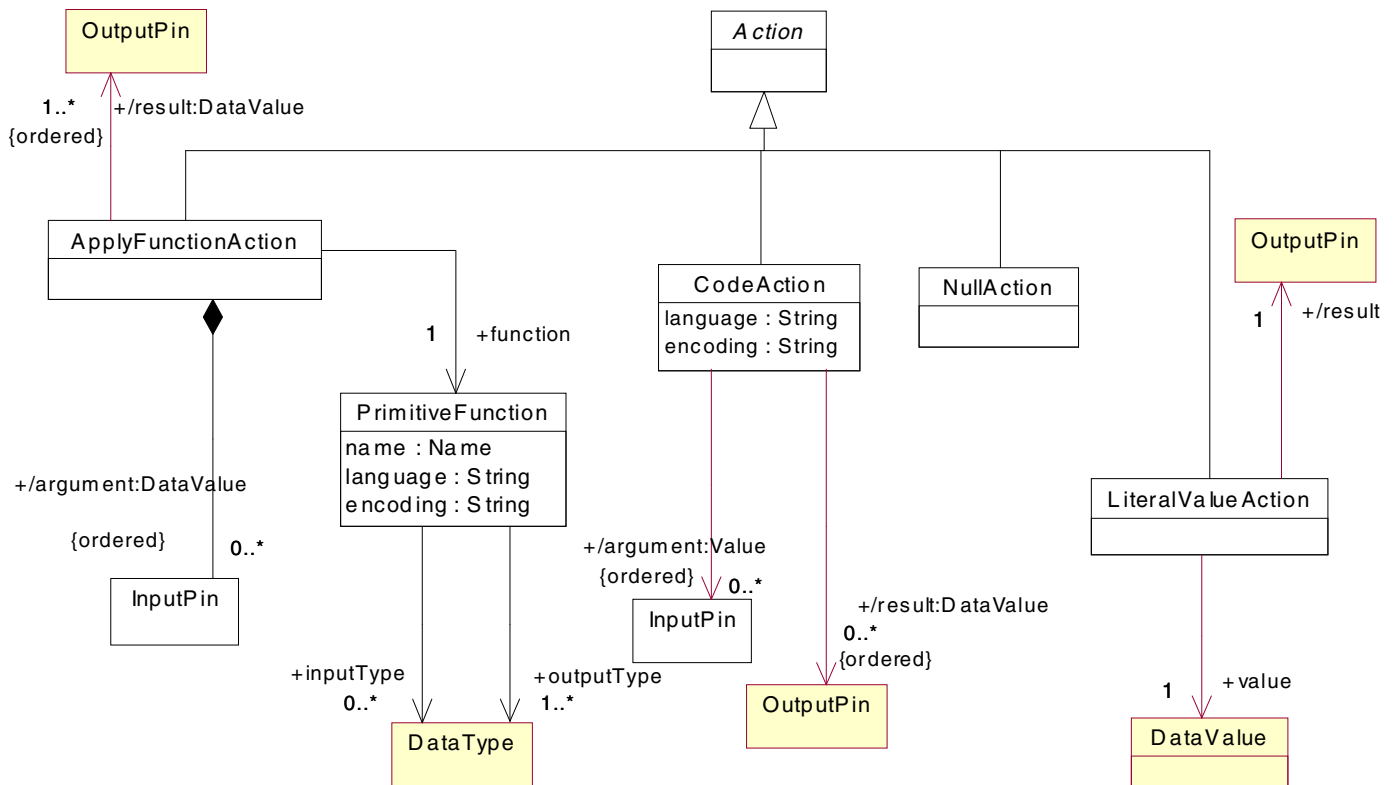


Figure 33. Computation classes metamodel

Literal value actions are broken out as a separate kind of action. These are to be regarded as special kinds of primitive functions, with zero inputs and one output.

Code actions are included here, although they might possibly have external effects and therefore not be pure mathematical functions. However, because they are explicitly implementation-dependent, it is hard to say much more about them within UML itself.

The following model shows the computation execution classes

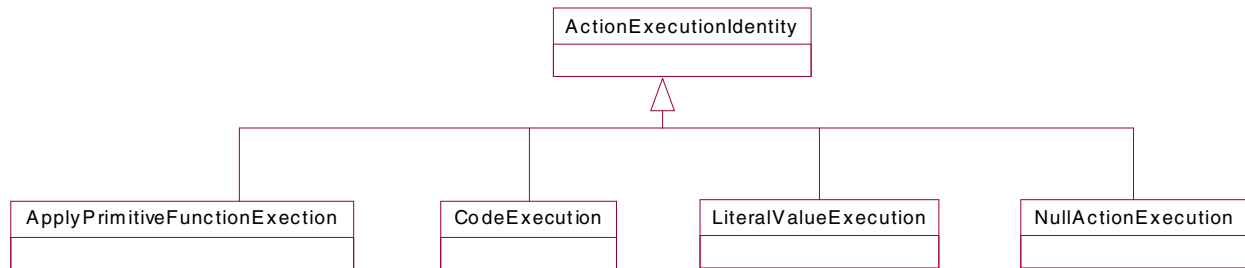


Figure 34. Computation execution classes

9.2. Computation Classes

ApplyFunctionAction

This action computes a primitive predefined mathematical function that depends only on the input values, with no access to object memory or to any objects. The execution and results of this action depend only on the function and the input values. There are absolutely no side effects of this action and it therefore cannot conflict with anything. All it does it produce result values using a mathematical function.

New primitive functions may be defined (outside of UML) as mathematical functions of input values to output values. All usual primitive operations should be considered as primitive functions, e.g., addition, nand, square root, finding a substring, Bessel function, etc.

A list of defined primitive functions may be supplied as part of a modeling profile. UML does not provide a mechanism to define primitive functions, as their definition is outside its scope. It is expected that users will agree on environments containing such definitions.

Attributes

None

Associations

function: PrimitiveFunction The function to execute

Inputs

argument: DataValue[*]

Outputs

result: DataValue[*]

Well-formedness rules

[1] The number and types of the input argument and output result pins must be compatible with the number and types of the parameters of the function.

```

self.input_argument()->size( ) = self.function.inputType->size( )
Sequence { 1..self.input_argument()->size( ) } -> forAll (i:Integer |
  let argumenti = self.input_argument (i)
  let inparameteri = self.function.inputType->at(i)

```

```

        argumenti.type.isCompatibleWith (inparameteri.type))
self.output_result()->size( ) = self.function.outputType->size( )
Sequence { 1..self.output_result()->size( ) } -> forAll (i:Integer |
    let resulti = self.output_result (i)
    let outparameteri = self.function.outputType->at(i)
    outparameteri.type.isCompatibleWith (resulti.type))

```

Therefore there are no functions on collections, but operations on collections can be constructed as actions at a higher level out of functional pieces.

Functions have no effect on and may not access object state.

Lifecycle

ready -> completed / set the output values to the given function of the input values

Production 1: Execute action

Precondition: self.status = ready

Assertions:

The number and types of actual arguments must match the number and types of the declared parameters of the function. Normal conformance rules apply in matching types (that is, the actual types can be descendants of the declared types).

```

self.action.function.inputType->size( ) = self.action.input_argument()->size( )
Sequence { 1 .. self.action.input_argument()->size( ) }->forAll (i:Integer |
    let argumenti = self.input_argument (i) and
    let inparameteri = self.action.function.inputType->at (i) and
    argumenti.class.isCompatibleWith (inparameteri.type) and -- the type of the actual argument must match
    argumenti->size( ) = 1 -- cardinality must be 1 for primitive function arguments
}

```

[2] The types and cardinalities of the result values must match the declaration of the operation.

```

self.action.function.outputType->size( ) = self.action.output_result()->size( )
Sequence { 1 .. self.action.output_result()->size( ) } ->forAll (i:Integer |
    let resulti = self.output_result (i) and
    let outparameteri = self.action.function.outputType->at (i) and
    outparameteri.class.isCompatibleWith (resulti.type) and -- contrapositive match
    outparameteri.multiplicity->inRange (resulti->size( )) -- cardinality must match multiplicity
}

```

Postcondition: self.status = completed

```

let m = self.action.operation.out_parameter->size( ) in
Sequence { 1 .. m }->forAll (i:Integer |
    let n = self.action.function.inputType->size( ) in
    self.output_result (i) = fj(self.input_argument->at(1), input_argument->at(2), ..., input_argument->at(n)) and
    assert (self.output_result(i).type.isCompatibleWith (self.action.function.outputType->at(i))
)

```

Where the function is defined as (for $j=1..m$) $b[i] = f_j(a_1, a_2, \dots, a_n)$ by appropriate mathematical functions f_j

The definition of the mathematical functions is outside of UML.

CodeAction

A code action performs an action that is defined outside of UML. This may involve external interactions, so it is not a mathematical transformation like a primitive function action. The action may have inputs and outputs. Code actions are must not alter object memory state, although it is hard to prevent such things. The semantics of a code action that alters object memory are undefined, together with the semantics of all subsequently executed actions.

Attributes

language: String -- the language in which the code action is specified
 encoding: String -- a string that identifies the action in the given language. Not meaningful to UML.

Associations

none

Inputs

argument: DataValue[*]

Outputs

result: DataValue[*]

Well-formedness rules

Clearly each kind of code action has constraints on its input and output values, but as the whole purpose of the code action is to do something that is outside of UML, it is not possible to specify the constraints within UML.

LiteralValueAction

Generates a literal value on the output pin. The value can be of any pure data type.

Attributes

None

Associations

value: DataValue

Inputs

none

Outputs

result: DataValue

Well-formedness rules

self.output_result().type = self.value.type

NullAction

An action that has no effect.

Attributes

None

Associations

none

Inputs

none

Outputs

none

Well-formedness rules

None

9.3. Execution Classes

ApplyPrimitiveFunctionExecution

Apply actions are specified as pure functions from a list of input values to a list of output values. This separates functional computation from data manipulation and flow of control issues. By necessity, a call on a method cannot be a functional step, as it includes all of these aspects.

We permit arbitrary definition of functions.

Functions operate on scalar values, because collections are objects. Primitive functions operation on data values.

CodeExecution

Lifecycle

ready -> executing / do the code action set the output values to the given function of the input values

Production 1: Execute action

Precondition: self.status = ready

There are constraints on the input values, but these must be defined outside of UML.

Postcondition: self.status = completed

let m = self.output_result->size() in

Sequence { 1 .. m }->forall (i:Integer |

let n = self.input_argument->size() in

self.output_result (i) = g_j(self.input_argument[1], input_argument[2], ..., input_argument[n], S_t) and
assert (self.output_result(i).type.isCompatibleWith (self.action.function.outputType->at(i))

)

Where the function is defined as (for i=1..m) b[i] = g_j(a[1], a[2], ..., a[n], S_t) by appropriate mathematical functions g_j, and where S_t represents the state of the system (and possibly the external environment) at the time when the action is executed..

LiteralValueExecution

The execution of a literal value action delivers a predefined value to the output pin.

Semantics

Production 1: Complete execution

Precondition: self.status = ready

Postcondition: self.status = completed

self.output_result().value = self.action.value

NullActionExecution

The execution of a null action has no observable impact.

Semantics

Production 1: Complete execution

Precondition: self.status = ready

Postcondition: self.status = completed

Chapter 10. Collection Actions

Collection actions permit the application of an action to a set of data elements, possibly in parallel. They avoid the need for explicit indexing and extracting of elements from collections, avoiding the overspecification of control that would otherwise be necessary. Each collection action contains a subaction that is executed once for each element in the input collection.

The motivation for collection actions is that various tasks yield collections, say finding the accounts owned by a customer, and now there is a need to do something with that collection: summing the balances, picking the account with the largest balance, counting the number of accounts, selecting accounts with more than a certain balance, etc.

Collection actions provide the mechanisms for acting on collections as a unit, rather than building loops, counters, iterators and the like to manipulate each element. There are four kinds of collection action, as follows:

Filter	The filter action selects a subset of the elements in a collection into a new collection based on a boolean result of the subaction applied to each element; for example, a selection of accounts above a certain balance from the collection of accounts associated with the customer.
Iterate	The iterate action applies a subaction repeatedly to each of the elements in a collection, accumulating the effects in loop variables. For example, counting the number of accounts.
Map	The map action applies a subaction in parallel to each of the elements of a collection of data resulting in an output that is a collection of the same size and shape; for example, paying interest on each account.
Reduce	The reduce action repeatedly applies a binary subaction to pairs of adjacent elements in a collection, implicitly replacing a pair of elements by the result, until the final result is a single element of the type in the collection. An example is summing up balances of all accounts.

10.1. General Rules for Collection Actions

Collection actions repeat an action (the *subaction*) once for each element in a collection of elements. The number of repetitions is governed by the size of the collection, rather than a programmed test condition required in a loop action. This section describes some general principles required for collection actions.

Collections have shape and size. By *shape*, we mean that they are all the same kind of collection. UML supports two kinds of collection: sets (unordered collections) and lists (ordered collections), but the concept could be generalized in the future.

By the same *size*, we mean that the contents of two collections can be put into one-to-one correspondence using some scanning rules provided by the collection. For UML collections, they must have the same number of elements.

Two collections are *congruent* if they have the same shape and size. For example, a list of five integers is congruent to a list of five bank accounts but not to a set of five integers or a list of six integers. A list of collections is congruent if all of the collections in the list are congruent to each other.

If a collection action takes more than one collection as input, they must all be congruent. Similarly, multiple collections produced by the same action must be congruent. The output collections must also be the same shape as the input collections because they contain one element per execution of the subaction, though the size may be smaller.

A list of collections is conceptually equivalent to a single collection of tuples, in which each tuple contains one element, at the same position, from each collection. Such a tuple, comprising one element from each collection, is called a *slice*. Each execution of the subaction acts on a slice, so that each execution receives a tuple of elements, one element from the corresponding positions in their respective collections. This is why the collections must be congruent—they all govern the same repetition of execution.

During each execution of the subaction, the execution receives a slice, represented in the action model as a list of output pins (output, because they represent values that can be accessed within the subaction), called the *subinput* pins. The number of subinput pins must equal the number of input collections to the collection action and the type of each subinput pin must match the type of element contained in the corresponding collection. For example, if the input to

the collection action is a collection of bank accounts, then the subaction receives a bank account during each execution.

The constraint *correspond* on two associations involving the same collection action means that the lists of pins in the two associations must have the same size and the type of each value in the client association matches the type of element contained in each corresponding collection in the supplier.

The constraint *match* on two associations from the same collection action means that two links from the same collection action instance have the same cardinality and that each pair of corresponding pins has the same type (but different positions may have different types); such pins do not represent collections of values. For example, the list of loop variables inside an iterate action must match the list of loop variable inputs of the iterate action; neither list involves collections.

10.2. Collection Action Classes

The following model shows the collection actions:

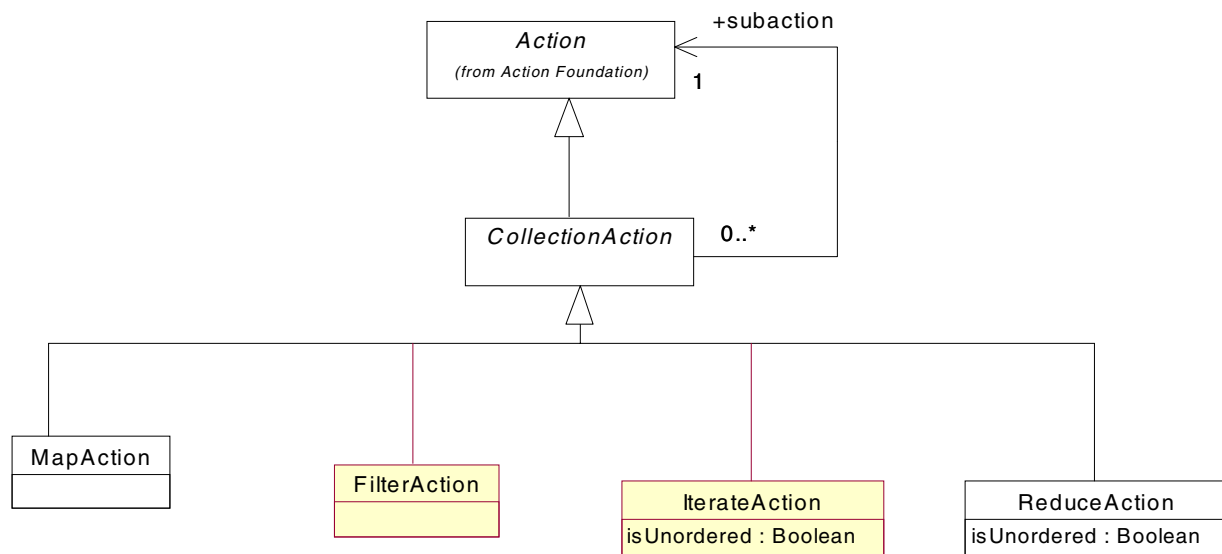


Figure 35. Collection actions

FilterAction

The filter action applies an action concurrently and independently to each of the elements of a collection. The action must yield a boolean output whose value is used to decide whether the input elements are passed through to the output collection. The result of the filter action is a collection of the same shape as the input collection, consisting of all the elements of the input collection for which the action yields a true value. Missing elements are removed from the output collection and the gaps in the collection are closed up.

Description

The subaction is executed once for each slice of input values, yielding a boolean test value. If the value is true, the slice of values is copied into the output collections; if it is false, the slice is omitted from the output collections and the gaps are closed up. All executions are concurrent.

A filter action has one or more input pins, each of which hold a collection type of the same shape, but each collection may contain a different type of element. All the input pins must be the same type of collection. At run time, the size of the collection on each pin must be the same. The output pins of a filter action are the same in number and type of each as the input pins. A filter action contains an embedded subaction which has a designated output pin of type Boolean. The subaction has access to a list of “subinput” output pins owned by the Filter action, equal in number

to the number of inputs and outputs of the Filter action. Each subinput pin has a type that is the type of element held within the collection on the corresponding input (and output) pin on the filter action; that is, the input pins hold collections and the subinput pins hold corresponding scalars. The subinput pins are connected to actions within the subaction; they may not be connected outside the filter action. The subaction may also access output pins within the wider containing scope. Output pins within the subaction may not be connected to pins outside the subaction.

The input pins and output pins of the filter action are not explicitly connected to anything inside the filter action; the data flow is implicit. The subaction is executed once for each slice of values in the input collections. During each execution, the values at a given position in each collection are made available to the subaction on the subinput pins. Each concurrent execution of the subaction gets a different slice of scalar values of the subinput pins. If the subaction accesses values from outside the filter action, such values will be the same for all the concurrent subaction executions during one execution of the filter action. The execution of each subaction yields a Boolean value on the designated testOutput pin. If the value is true, the slice of input values goes through as a slice of values in collections on the output pins. If the value is false, the slice of input values is omitted from the output collections, which are “closed up” to eliminate the missing position.

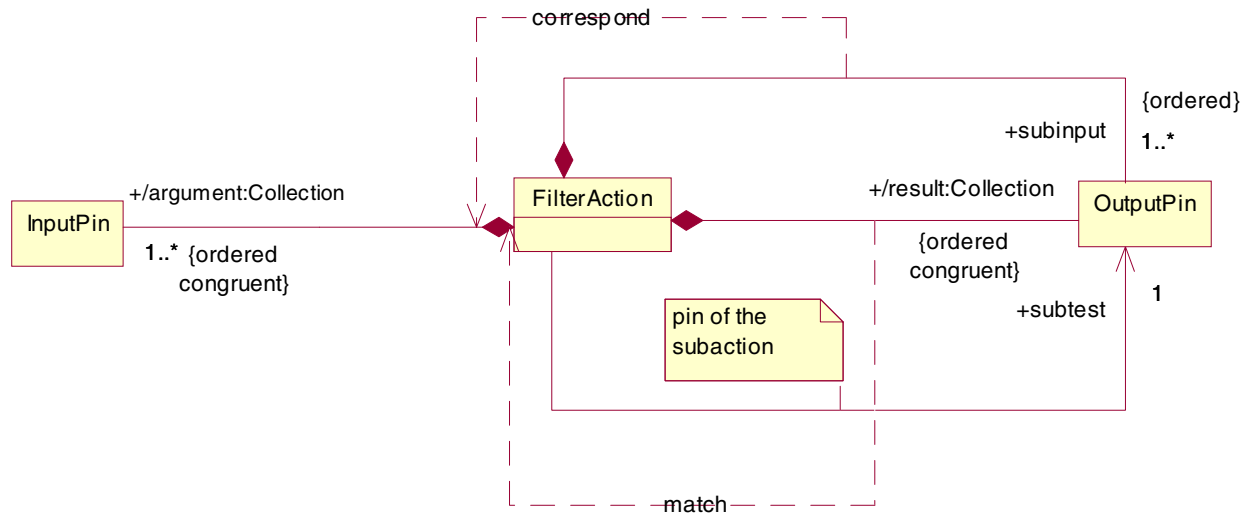


Figure 36. Filter action

Attributes

None

Associations

subaction: Action An action that tests whether to keep the corresponding collection elements in the result. A slice of values, one from each input collection, is available to the subaction. It must have an output pin that yields a Boolean value (see subtest association). For each execution yielding a true value, the corresponding elements from the input collections are copied to the output collections of the filter action.

subinput: OutputPin[1..*]

An ordered list of output pins owned by the filter action itself. Each output pin has a type that matches the type of element in the input collection at the same position. These output pins are accessible to the subaction. During each execution of the subaction, a value from each input collection is copied to the corresponding subinput pin. The subaction is executed once for each position in the input collections.

subtest: OutputPin References an accessible output pin of type Boolean owned by the subaction. During each execution of the subaction, the value on this output pin determines whether the corresponding values from the input collections are copied to the output collections. If the value is false, the elements do not appear in the output collections and the gap is closed up.

Inputs

argument: Collection[1..*] of Value

An ordered list of collections of values. All the collections must have the same shape and size, but each collection may contain a different type of element. The subaction is executed once for each slice of values.

Outputs

result: Collection[1..*] of Value

An ordered list of collections of result values. Each collection must have the same shape as the input collections. The number of result collections must equal the number of argument collections. Each result must have the same element type as the corresponding argument. After execution of the filter action, each result collection will contain a (possibly empty) subset of the elements in the corresponding input collection, with the elements in the same order as in the inputs.

IterateAction

Applies an action repeatedly, once for each slice of values from a list of input collections. A tuple of loop variables accumulates the result of the iteration and is eventually passed to the output of the iterate action. This action represents a loop controlled by the elements of a collection, rather than an explicit index variable, collection index actions, and explicit termination condition, and therefore avoids overspecifying iteration mechanics for many common operations

Description

The iterate action is a kind of loop executed once for slice of elements in a list of input collections. During each execution, the slice of values is made available to the subaction. The slices are presented from first to last in the scan order for the collection (for a list, from first element to last). If the input collection shape is a set, or if the *isUnordered* attribute is true, then the slices are presented in an indeterminate (and not necessarily repeatable) order; this option should be used only if the order of computation does not affect the result. Like a loop, an iterate action has loop variables that accumulate the effects of the iteration. The subaction accesses the previous values of the loop variables and computes new values for the next execution. The initial values of the loop variables are supplied by inputs to the overall iterate action, and the final values of the loop variables become the results of the overall iterate action. It is not necessary to have loop variables, but if they are absent, the action can have an effect only by writing memory values.

An iterate action has two banks of input pins: one bank (collectionInput) is a list of collections, each the same type of collection, but they contain different types of elements. On any given execution, all the collections must be the same size. The second bank of input pins are scalar types; these values are used to initialize the loop variables. The iterate action has one bank of output pins, whose number and types match the loop variable input pins.

The loop variable contains a subaction, an action that is executed once for each slice of values from the input collections. The iterate action owns a bank of internal OutputPins, matching the bank of loop variable input pins. On the initial execution of the subaction, these pins get the values from the loop variable input pins. The iterate action also owns a bank (labeled subinput) of internal OutputPins, equal in number to the collection input pins. The type of each subinput pin matches the type of element containing in the corresponding collection input pin. During each execution of the subaction, these pins hold one slice of values from the collections. The iterate action also designates (as suboutput) a bank of OutputPins owned by the subaction. At the conclusion of the execution of the subaction, the values on these pins become the new values of the loop variable pins.

The subaction has access to the subinput values and the loop variable values. These values change during each execution of the subaction. It may also access available OutputPins in the containing scope. Such values are fixed during the executions of the subactions for any one execution of the iterate action. During one execution, the subaction computes values for the suboutput pins. The values on the suboutput pins become the new values of the loop variable

pins on the next iteration of the subaction. When all slices of the collections have been processed, the final values of the loop variables become the values on the result output pins of the overall iterate action.

No output pins of the subaction may be connected outside the iterate action. The input pins and output pins of the iterate action are not explicitly connected to anything inside the iterate action; the data flow is implicit. The subaction is executed once for each slice of values in the input collections. During each execution, the values at a given position in each collection are made available to the subaction on the subinput pins. Each successive execution of the subaction gets a different slice of scalar values of the subinput pins. If the subaction accesses values from outside the filter action, they will be the same for all the concurrent subaction executions in an execution of the filter action.

The *isUnordered* attribute states that the order of execution of slices is irrelevant, even though the ordering of elements in each collection is still used to match corresponding elements into slices. The purpose of this attribute would be to remove overspecification of ordering and permit optimization within an implementation, especially if values are not all computed at the same time (such as lazy evaluation). If the input collection shape is a set, then the slices are processed in an indeterminate order.

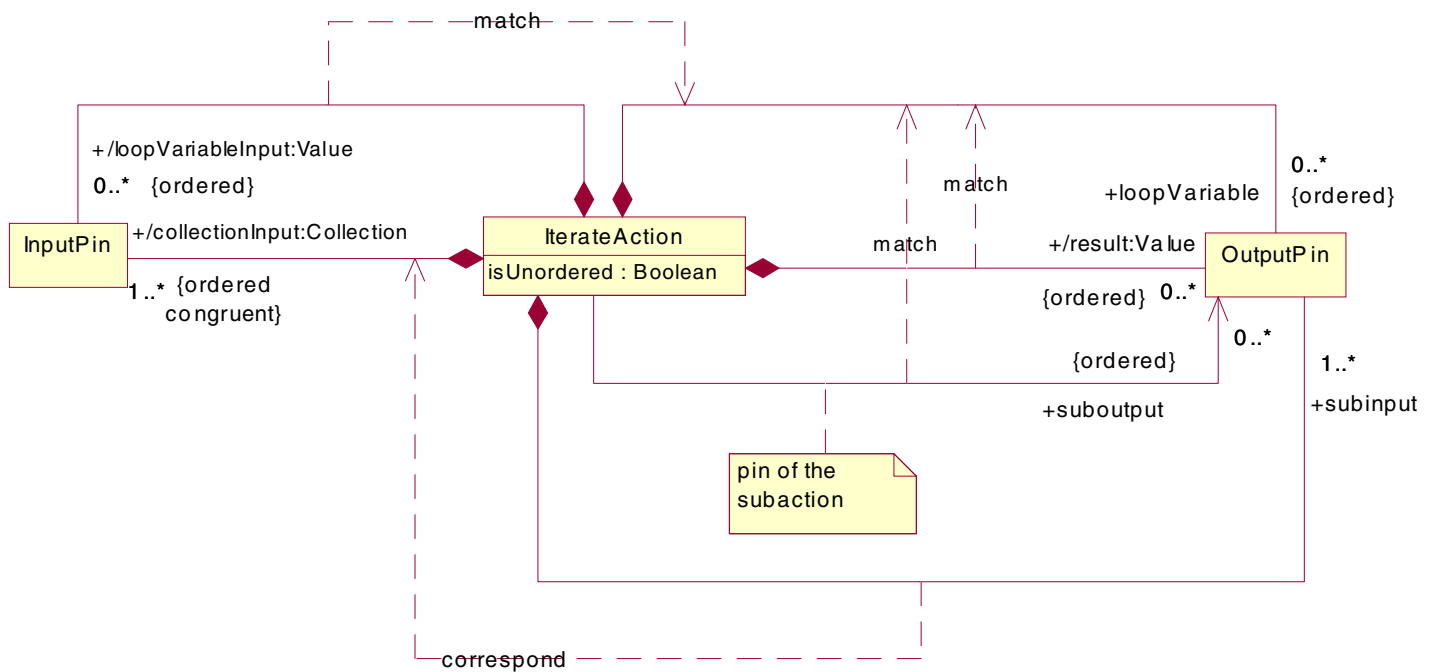


Figure 37. Iterate action

Attributes

isUnordered If true, indicates that the slices of values may be given to the successive executions of the subaction in any order. This should be set only if the final result is insensitive to execution order. If false, the subaction is executed on slices of the collections in order from first to last, in accord with the scan order of the particular kind of collection. If the collection is a set, then the iteration order is automatically unordered and this flag has no further effect.

Associations

subaction: Action An action that is executed repeatedly and sequentially, once for each slice of values in a list of input collections. During each execution of the action, one slice of values is made available to the execution, from the first to last position in the collections. Like a loop action, this action has a list of loop variables that make the results of one execution of the subaction available to the next sequential execution of the subaction. On the first execution of the subaction, the loop variables are set by the loopValueInput values that are input to the overall IterateAction. The iterate action designates a list of output pins from the subac-

tion as the updated values of the loop variables. During each execution of the subaction, the previous values of the loop variables are available within the subaction. After each execution, the designated output values in the subaction are copied to the loop variables. After the subaction has been executed once for each slice of the input collections, the final values of the loop variables are copied to the output pins of the iterate action.

loopVariable: OutputPin[0..*]

A (possibly empty) list of output pins (owned by the iterate action) whose values represent the accumulated result of executing the action so far. During the first execution of the subaction, each pin holds a copy of the corresponding loopVariableInput pin (among the input pins of the iterate action). During each subsequent execution of the subaction, each pin holds a copy of the value on the corresponding suboutput pin within the previous execution of the subaction.

subinput: OutputPin[1..*]

A nonempty list of output pins (owned by the iterate action) whose values represent one slice of the input collections during one execution of the subaction. During each execution of the subaction, a different slice of values appears on the pins. The pins are available to the subaction. The type of each element pin must match the type of element held by each collection on the corresponding input pin.

suboutput: OutputPin[0..*]

A (possibly empty) list of available output pins owned by the subaction. They represent updated values of the loop variables computed by the subaction. After each execution of the subaction, the values on these pins are copied to the loop variable pins for the next execution of the subaction or the output result of the overall iterate action. This list of pins must match in number and types the loopVariable pins.

Inputs

collectionInput: Collection[1..*] of ValueAn ordered list of input pins holding collections of argument values.

All the collections must have the same shape, but each argument may hold a collection of a different type of element. During each execution of the subaction, one value from the same position of each collection constitutes a slice of values available to that execution of the subaction. The subaction is executed once for each slice of values. During execution of the iterate action, all of the input collections must have the same size.

loopVariableInput: Value[0..*]

An ordered list of input pins holding values. The number and types of the pins must match the loop variable pins. These pins represent the initial values for the loop variables. During the first execution of the subaction, the loop variable pins hold copies of the values on the corresponding loop variable input pins.

Outputs

result: Collection[0..*] of ValueAn ordered list of output pins holding collections of result values. The number and type of each pin must match the corresponding loop variable input pin and loop variable pin. After the execution of the subaction on the final slice of values from the input collections, the result pins get copies of the values on the suboutput pins on the final execution of the subaction. If the input collections are empty, the result pins get copies of the values on the loop variable input pins.

MapAction

The map action applies a subaction concurrently and independently to each slice of elements from the input collection. If the internal action has output values, the output values of all the executions are assembled into a list of output collections of the same size and shape as the input collections.

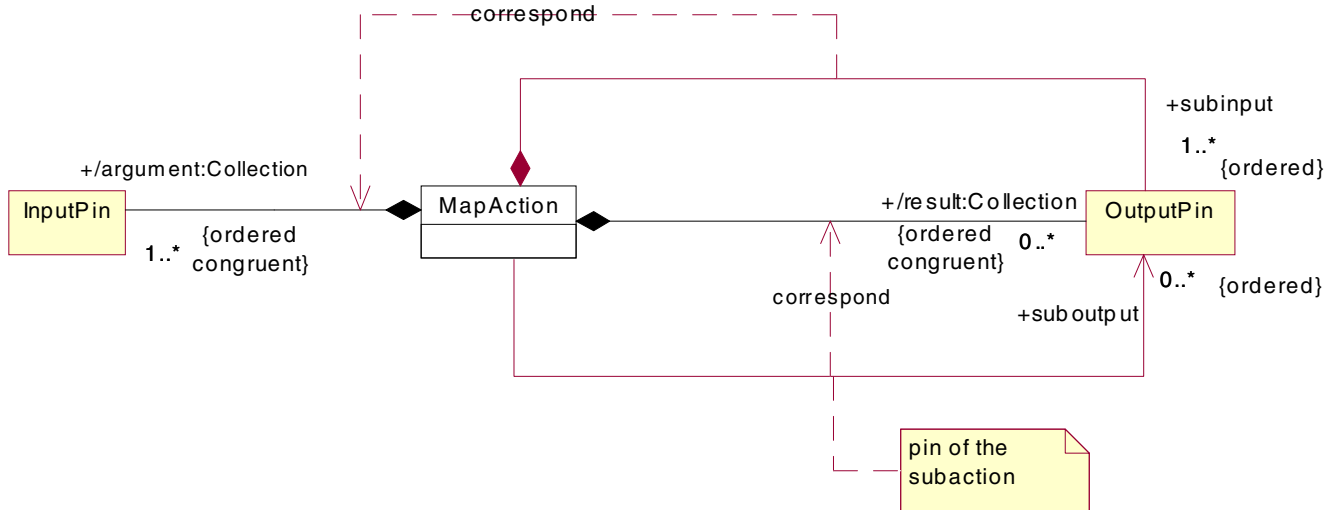


Figure 38. Map action

Description

The subaction is executed concurrently, once for each slice in the collection of argument values, all of which must have the same shape and size. During each concurrent execution of the subaction, the subinput pins hold the values on one slice of the argument collections. After the completion of execution of the subaction, the values on the suboutput pins form one slice of the result collections. The subaction may access available inputs from outside the map action. No output pin of the subaction is available outside the map action. The values on the suboutput pins from the individual executions are formed into collections, each containing one value from each execution. The result collections have the same shape and size as the argument collections and each result value occupies the corresponding position in its collection, but the types of elements may be different. When the executions of all the actions are complete, the collections of the result values are made available on the output pins of the map action. Result values are not required, but if they are absent, the subaction must produce its effects by writing memory values.

A map action has one or more input pins, each of which hold a collection type of the same shape, but each collection may contain a different type of element. The map action has zero or more output pins, each of which holds the same type of collection, but the types of the contents of the collections may differ from each other and the input collections. All the input and output pins must be the same type of collection. At run time, the size of the input collection on each pin must be the same, and the size of each output collection will be the same as the input collections. The subaction has access to a list of “subinput” output pins owned by the Map action, equal in number to the number of inputs of the Filter action. The map action also designates a list of “suboutput” output pins that are owned by the subaction. Each subinput pin and suboutput pin has a type that is the type of element held within the collection on the corresponding input or output pin on the filter action; that is, the input and output pins hold collections and the subinput and suboutput pins hold corresponding scalars. The subinput pins are typically connected to actions within the subaction; they may not be connected outside the filter action. The suboutput pins are typically connected indirectly to the subinput pins. The subaction may also access output pins within the wider containing scope. Output pins within the subaction may not be connected to pins outside the subaction.

The input pins and output pins of the map action are not explicitly connected to anything inside the map action; the data flow is implicit. The subaction is executed once for each slice of values in the input collections. During each execution, the values at a given position in each collection are made available to the subaction on the subinput pins. Each concurrent execution of the subaction gets a different slice of scalar values on the subinput pins. If the subaction accesses values from outside the filter action, such values will be the same for all the concurrent subaction executions during one execution of the map action. The execution of each subaction yields values on the suboutput pins within

the subaction. The values on the suboutput pins form a slice of the output collections at the same position within the collections as the input slice.

If executions of the subaction conflict (because they write shared objects) then the result is indeterminate and may be undefined.

Attributes

None

Associations

- subaction: Action An action to be executed once for each slice of values in a list of argument collections. During each execution of the subaction, one element value from each argument collection is available to the subaction. The subaction must produce one output value for each result collection; these are formed into output collections of the same size and shape as the input collections to the map action.
- subinput: OutputPin[1..*] A nonempty ordered list of output pins owned by the map action itself. Each output pin has a type that matches the type of element in the input collection at the same position. These output pins are accessible to the subaction. During each execution of the subaction, a value from each input collection is copied to the corresponding subinput pin. The subaction is executed once for each position in the input collections.
- suboutput: OutputPin[0..*] A (possibly empty) ordered list of output pins owned by the subaction. The number of suboutput pins matches the number of result pins of the map action, and each suboutput pin has a type that matches the type of element in the collection held by the map action result pin at the same position. The value from each suboutput pin is copied to the corresponding map action result pin at the same position in the collection as the input slice to the subaction.

Inputs

- argument: Collection[1..*] of ValueAn ordered list of collections of argument values. All the collections must have the same shape, but each argument may have a collection of a different type.

Outputs

- result: Collection[0..*] of ValueAn ordered list of collections of result values. Each collection must have the same shape as the input collections, but the number and element types of result pins can be different. After completion of execution of the subaction for each slice of input values, each result collection has one element value equal to the value of the corresponding suboutput pin for the same position in the collection.

ReduceAction

Applies an associative binary subaction, from two values of the same type yielding a value of the same type, repeatedly to adjacent pairs of slices in a list of collections, until the (intermediate working) collection is reduced to a single slice of values, which become the output values of the reduce action. The order in which the subaction is applied to pairs of values is indeterminate, and does not affect the ultimate result if the action is associative.

Description

The reduce action reduces a collection of values of a given type to a single value of the same type. It requires a binary associative operator from two inputs of the type producing an output of the same type. The operator need not be symmetric; an example is matrix multiplication. The operator is considered as applying between each pair of values, in all, one less than the number of values. Because of associativity, the operator can be applied to adjacent pairs of values in any order.

A reduce action has a bank of input pins. each of which holds a collection. The type of all collections must be the same, but they contain different types of elements. On any given execution, all the collections must be the same size. The reduce action has a bank of output pins, equal in number to the input pins. Each output pin holds an element of the same type as those contained by the corresponding input pin.

The reduce action contains a subaction. The reduce action owns two banks of “subinput” output pins that are available to the subaction. These are the left bank and the right bank. Each bank has identical structure to the structure of the output pins of the reduce action. The reduce action also designates a bank of “suboutput” output pins that are owned by the subaction itself. This bank also has structure identical to the output pins of the reduce action and to each of the subinput pins.

The reduce action works somewhat differently from the other collection actions, in that the subaction is executed one fewer time than the size of the input collections. The input may be thought of conceptually as a collection of tuples, where the tuples have the structure of the subinput pins, suboutput pins, and result pins of the reduce action. The subaction operates on adjacent pairs of tuples in the collection. (If the collection is unordered, then the values are presented to the subactions in an indeterminate order.) The subaction takes two slices of values (a left and a right tuple) and produces a new slice of values as suboutput. The suboutput slice conceptually replaces the two input slices in the collection of tuples. (The actual collections are not modified.) The subaction is applied repeatedly to pairs of slices until a single slice remains. Its value is placed on the result output pins of the overall reduce application. In other words, the reduce action serves to reduce a collection of values to a single value by repeated application of a binary function.

No output pins of the subaction may be connected outside the reduce action. The input pins and output pins of the iterate action are not explicitly connected to anything inside the reduce action; the data flow is implicit. If the subaction accesses values from outside the reduce action, such values will be the same for all the concurrent subaction executions during one execution of the reduce action.

The *isUnordered* attribute states that the subaction is symmetric and that the reduction can be applied to the slices in any order, even though the ordering of elements in each collection is still used to match corresponding elements into slices.

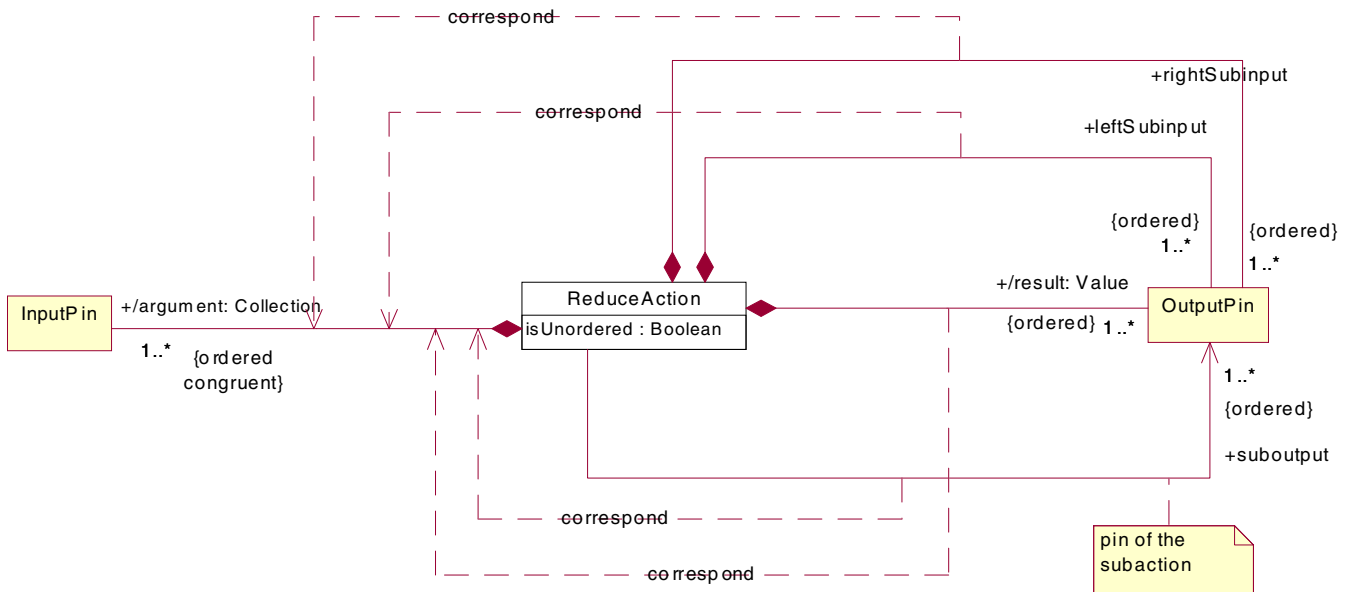


Figure 39. Reduce action

Attributes

isUnordered: Boolean If true, indicates that the slices of values may be given to the successive executions of the subaction in any order. This should be set only if the final result is insensitive to execution order. If false, the subaction is executed on slices of the collections in order from first to last, in accord with the scan order of the particular kind of collection. If the collection is a set, then the iteration order is automatically unordered and this flag has no further effect.

Associations

subaction: Action An action that is executed repeatedly and sequentially, once for each slice of values in a list of input collections. During each execution of the action, one slice of values is made available to the execution, from the first to last position in the collections. Like a loop action, this action has a list of loop variables that make the results of one execution of the subaction available to the next sequential execution of the subaction. On the first execution of the subaction, the loop variables are set by the loopValueInput values that are input to the overall IterateAction. The iterate action designates a list of output pins from the subaction as the updated values of the loop variables. During each execution of the subaction, the previous values of the loop variables are available within the subaction. After each execution, the designated output values in the subaction are copied to the loop variables. After the subaction has been executed once for each sliced of the input collections, the final values of the loop variables are copied to the output pins of the iterate action.

leftSubinput: OutputPin[1..*]

A nonempty list of output pins (owned by the reduce action) that represent the first of two tuples that are arguments to the subaction. On each execution of the subaction, these pins hold copies of the values in the first of two adjacent slices in the working collection of the reduce action. The number of pins and their types must match the output pins of the reduce action.

rightSubinput: OutputPin[1..*]

A nonempty list of output pins (owned by the reduce action) that represent the second of two tuples that are arguments to the subaction. On each execution of the subaction, these pins hold copies of the values in the second of two adjacent slices in the working collection of the reduce action. The number of pins and their types must match the output pins of the reduce action.

suboutput: OutputPin[1..*]

A nonempty list of available output pins owned by the subaction that represent the results of executing the subaction. After each execution of the subaction, the slice of values on these pins replaces the two adjacent slices in the working collection that served as subinputs to the subaction. The number of pins and their types must match the output pins of the reduce action.

Inputs

argument: Collection[1..*] of Value An ordered list of input pins holding collections of argument values. All the collections must have the same shape, but each argument may hold a collection of a different type of element. One value from the same position of each collection constitutes an ordered slice of values. The overall input to the reduce action is an ordered collection of slices that is the initial “working collection” of slice values. Each execution of the subaction takes two adjacent slices as the left subinput and right subinput to the subaction. After the execution of the subaction is complete, the value of the suboutput pins is a slice of values. This slice implicitly replaces the two adjacent slices used by the subaction in the working collection, which shrinks by one slice. When the working collection is reduced to a single slice, its value is copied to the result pins.

Outputs

result: Value[1..*]

An ordered list of output pins holding result values. The number of pins and type of each pin must match the number of input pins and the type of element in the collection held by each corresponding input pin. After the final execution of the subaction, the values of the suboutput pins are copied to the result pins of the reduce action and become available outside the result action.

10.3. Execution Model Actions

The run-time concepts for collection actions are shown in the following UML execution model.

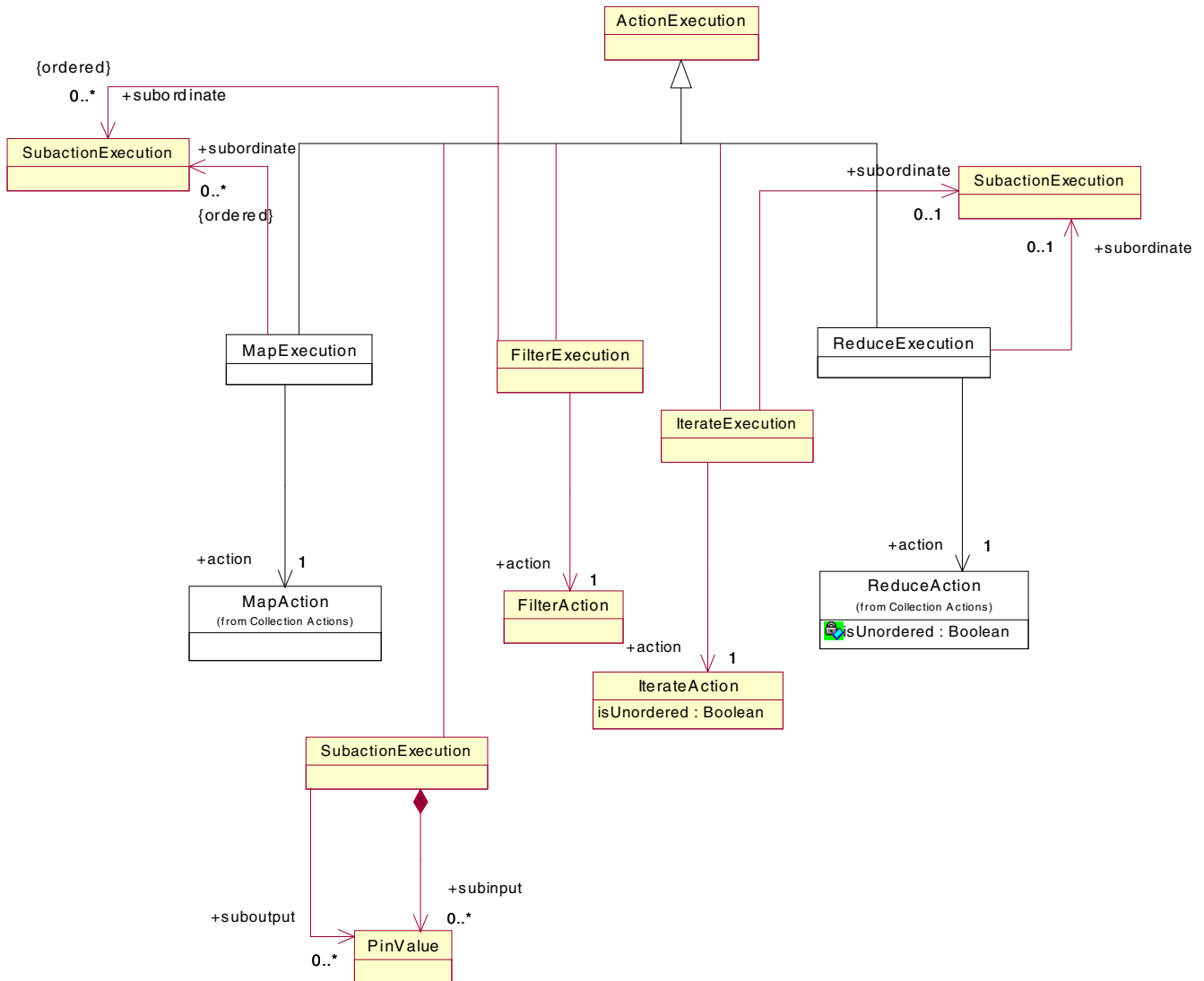


Figure 40. Collection action execution

FilterExecution

Lifecycle

Production 1: Create an execution for each element in the collection (use the first collection as a template, but they must all match). Copy a slice of the input collections as the input values to the execution.

Precondition: self.status = ready

Postcondition: self.status = executing and

let nparameters = self.action.inputPin->size and

let nresults = self.action.outputPin->size and

Sequence { 1 .. self.inputPinValue->at (1)->size } -> forAll (i:Integer |

let subi:ActionExecution.isNew();

```

    subi = self.subordinate->at(i);
    subi.status = ready;
    subi.context = OCLnestedContext (self.context); -- this must create a new nested context
    Sequence { 1 .. nparameters } -> forAll (j:Integer |
        let subinputj = subi.subinput->at(j) and
            subinputj.pin = self.action.subinput->at(j) and
            subinputj.value = self.inputPinValue->at(j)->at(i) } and
        subi.suboutput->at(1).pin = self.action.subtest
    )

```

Production 2: When all the subordinate executions are complete, gather the slices for true outputs into collections.

```

Precondition: self.status = ready and
    self.subordinate -> forAll (status = complete)
Postcondition: self.status = complete and
    let nresults = self.action.result->size and
        nelelements = self.subordinate->size and
        Sequence { 1 .. nresults } -> forAll (j: Integer |
            let resultj = Sequence { 1 .. nelelements } -> iterate ( i: Integer, result: Collection { } |
                if self.subordinate->at(i).suboutput->at(1).value then result->append (self.argument->at(j))
                    else result ) and
                self.result->at(j) = resultj

```

IterateExecution

Lifecycle

Production 1: Start the iteration

```

Precondition: self.status = ready
Postcondition: self.status = looping and initialize the loop variables

```

Production 2: Do a step

```

Precondition: self.status = looping and there are more elements from the input
Postcondition: self.status = executing and initialize the action inputs and start the action

```

Production 3: All elements exhausted

```

Precondition: self.status = looping and there are no more elements from the input
Postcondition: self.status = complete and copy out the results

```

Production 4: Subaction execution complete

```

Precondition: self.status = executing and subordinate is complete
Postcondition: self.status = looping and update the loop variables from the subaction outputs

```

MapExecution

Lifecycle

ready -> executing / create a procedure execution for each element in the input collection

executing [all subordinate executions complete] -> complete / form collections for each output

Production 1: Create an execution for each element in the collection (use the first collection as a template, but they must all match). Copy a slice of the input collections as the input values to the execution.

```

Precondition: self.status = ready
Postcondition: self.status = executing and
    let nparameters = self.action.argument->size and
        nresults = self.action.result->size and
        Sequence { 1 .. self.argument->at (1).value->size } -> forAll (i:Integer |
            let subi:ActionExecution and OCLisNew (subi) and
                subi = self.subordinate->at(i);

```

```

subi.status = ready;
subi.context = OCLnested (self.context); -- this must create a new nested context
Sequence { 1 .. nparameters } -> forAll (j:Integer |
  let subinputj = subi.subinput->at(j) and
  subinputj.pin = self.subinput->at(j) and
  subinputj.value = self.argument->at(j).value->at(i) )
)
Sequence { 1 .. nresults } -> forAll (j:Integer |
  let suboutputj = subi.suboutput->at(j) and
  suboutputj.pin = self.suboutput->at(j)
)
)
)

```

Production 2: When all the subordinate executions are complete, recollect the outputs into collections.

Precondition: self.status = ready and

self.subordinate -> forAll (status = complete)

Postcondition: self.status = complete and

let nresults = self.action.result->size and

Sequence { 1 .. nresults } -> forAll (j:Integer |

let resultj = self.result->at(j);

resultj.value.type = self.argument->at(1).type;

-- the outputs are all the same kinds of collections as the inputs

oclIsNew(resultj);-- the output is a new collection

resultj.value->at(i) = self.subordinate.suboutput->at(i).value->at(j) }-- it has a value for each subordinate

ReduceExecution

Lifecycle

Production 1: start the reduction

Precondition: self.status = ready

Postcondition: self.status = looping

Production 2: if there are at least two elements, reduce them

Precondition: self.status = looping and there are two or more elements so select two adjacent ones and set up the left and right inputs

Postcondition: self.status = executing and initialize the subaction on two adjacent elements

Production 3: a subaction has completed, replace the two elements and shrink the collection

Precondition: self.status = executing and subaction is complete

Postcondition: self.status = looping and replace the two elements with the result

Production 4: There is only one element left

Precondition: self.status = looping and the collection has only one element

self.status = complete and self.result is the element value

Chapter 11. Messaging Actions

These actions exchange messages among objects. The messages may be synchronous (calls or invitations) or asynchronous (signals). In both cases, the message is processed by a totally distinct context and all information is transmitted by parameters that are passed by value.

11.1. Procedures and Calls

The call action represents a traditional procedure call, in which input values to the procedure are transmitted as a list of argument values and output values are transmitted as a list of result values. All argument values and result values are transmitted “by value,” that is, they do not include mechanisms for “by name” parameters or addresses of local variables. The values can include object handles or references.

A procedure represents a distinct, closed context. No variables are shared among caller and called procedure. A procedure also does not have “global variables” or “static variables” that hold state. Any such state must be implemented using objects. Because it has no state, many executions of the same procedure are conceivable on the same object, each with its own execution context. This does not mean that all procedures must support such simultaneous access, and many will not, but simply that the structure of procedures does not prevent it.

The target object can be different from the calling object and they can have different classes. They may live on different nodes in different locations. We do not distinguish “local calls” from remote procedure calls; semantically, there is no difference. There is an implicit chain of call executions during execution, but no need to assume that they are organized into a stack or owned by an explicit task or process. Such things are implementation choices that can be implemented in different ways.

While the execution of the called procedure is progressing, the calling action execution is “blocked.” This does not require any special mechanisms; the call action execution is simply in the “executing” state until the called procedure execution, which it created, terminates and returns. The called procedure execution has, as part of its state, sufficient information to identify the calling action execution. There is no special “return” action available to a procedure. A procedure execution returns when it completes the execution of its action, and the output values of the procedure are used as the return values.

11.2. Sending Signals

Sending a signal requires a list of parameter values that are marshalled into an argument list of values. The send action does not block after the values have been sent on their way to the target active object. It is complete as soon as the send information has been transmitted, and subsequent actions may execute. The transmission of a send packet may require some finite time, so the sender cannot assume that the message has been received immediately.

Unlike the call action, the send action must represent the situation of “information in transit”. This is modeled by the state of a `SendPacket`, which represents a message that has not been received yet. Such a construct is not needed for the call action (although it could have been added), because nothing can happen to the caller while a call is “in transit,” therefore there is nothing to model.

Signals must be sent to active objects. Only active objects can receive signals. The mechanism by which active objects process received signal is discussed in Chapter 3, State Machine Execution.

By definition, a passive object is one that may not receive signals. Both passive and active objects can be the target of calls; in both cases, the execution of a call creates a new procedure execution and does not interact with an existing state machine or running action execution. A call creates, from the point of view of the receiver, a new “thread of control” that executes alongside existing “threads of control” on the same object, but without sharing control or variables. Different executions interact only indirectly, through the attributes of the target object, which are shared by all executions on it.

11.3. Invitations

An Invitation is a “synchronous signal”. It is sent by an Invite action within an execution. Both the Invite action and the Send action transmit a signal to a target active object, and in both cases a list of argument values may be part of the transmission. Unlike a Send action, an Invite action blocks the inviter until the signal is received by the target active object, triggers a transition, and the procedure attached to the transition completes execution. If a state machine has a transition triggered by an invitation, the transition must have exactly one attached procedure, whose input and output parameters match the inputs and outputs of the invitation. When the transition is triggered by an Invitation, the procedure is executed. When the procedure execution is complete, its output values are passed back to the action that sent the Invitation, and they become the outputs of the action.

An invitation allows an action to send a signal to a state machine, trigger a transition, and get return values back from the procedure attached to the transition.

The Invitation is the representation of the “call event”. It is a kind of “synchronous send”. The caller must explicitly send an Invitation when one is needed. However, if it is desired to hide the implementation from the caller, then the following pattern will work: The caller calls an ordinary operation on the target class. The method for this operation simply sends an Invitation to the self object, passing along the arguments. When the Invitation returns, its result values are returned to the caller of the operation.

11.4. Messaging Classes

CallAction

Call an operation. Creates a subordinate procedure execution, copies the arguments to it, and waits until it completes, at which time the results are copied back as the results of the call action execution.

Attributes

None

Associations

operation: Operation The operation to be called. This must be defined in the class of the target object.

Inputs

target: Object The target object. The operation will be applied to the class of this object to obtain a method procedure to execute the operation.

argument: Value[*] The remaining arguments of the operation. Their number and types must match the operation.

Note. It must be possible to determine the target of a call at run-time (e.g., if a remote procedure call is sent to another object).

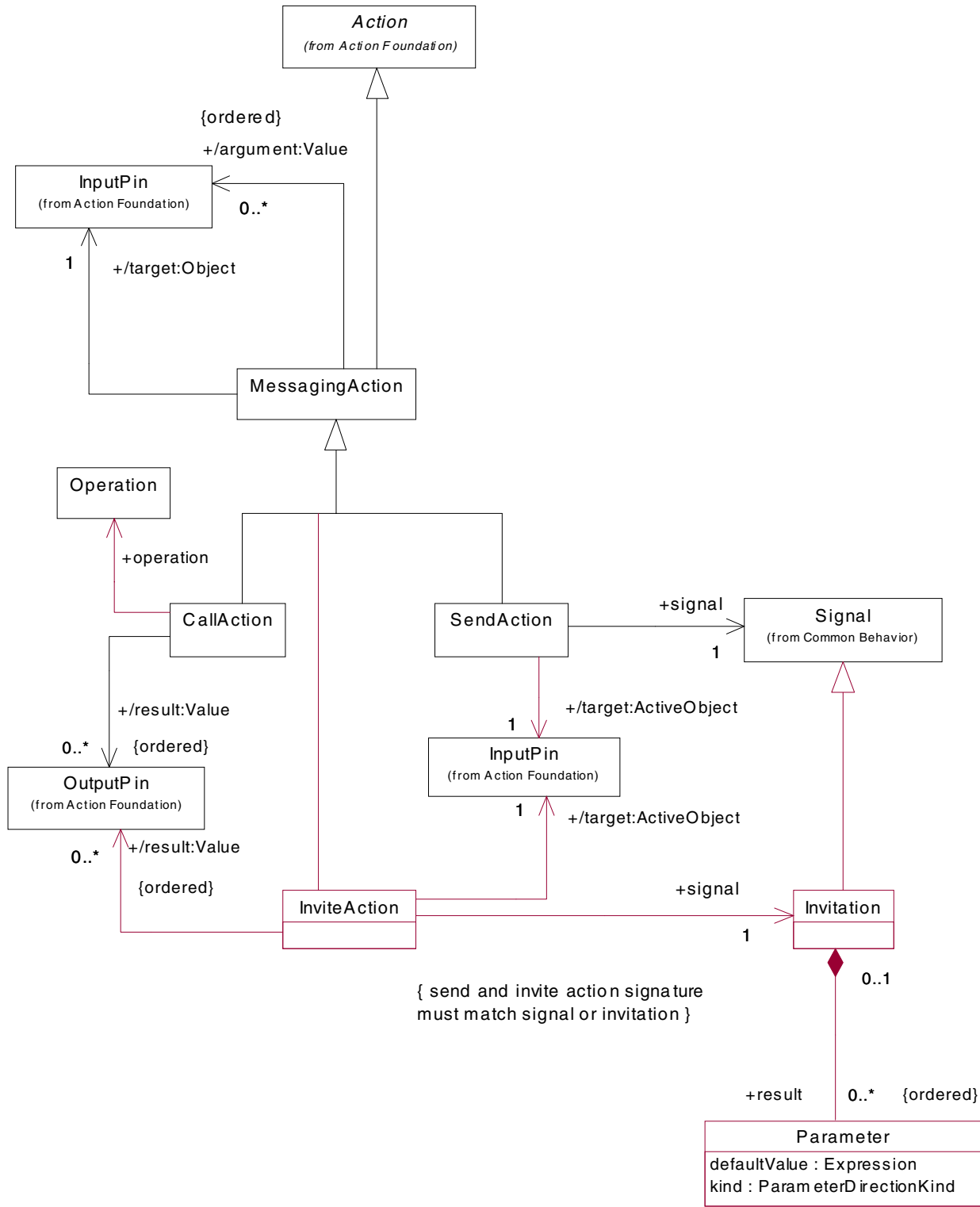
Outputs

result: Value[*] The results produced by executing the operation on the given arguments. The number and types of the results must match the operation.

Description: The operation is called on the target object with the given list of arguments. When the execution of the operation is complete, its result values become the result values of the call action.

Well-formedness rules

```
context Operation::in_parameter
  post: result = parameter->select (p:Parameter | p.kind = #in or p.kind = #inout)
context Operation::out_parameter
  post: result = parameter->select (p:Parameter | p.kind = #result or p.kind = #out or p.kind = #inout)
```



result parameters have kind=return

Figure 41. Messaging classes metamodel

- [1] The operation must be defined for the type of the target pin.
`self.target.type.operations->includes (operation)-- Note: I assume this includes inherited operations???`
- [2] The number, types, and multiplicities of the input argument and output result pins must be compatible with the number, types, and multiplicities of the parameters of the operation.
`self.input_argument()->size() = self.operation.in_parameter->size()
Sequence { 1..self.input_argument()->size()} -> forAll (i:Integer |
 let argumenti = self.input_argument (i);
 let inparameteri = self.operation.in_parameter->at(i);
 argumenti.type.isCompatibleWith (parameteri.type);
 argumenti.multiplicity.isCompatibleWith (parameteri.multiplicity))
self.output_result()->size() = self.operation.out_parameter->size()
Sequence { 1..self.output_result()->size()} -> forAll (i:Integer |
 let resulti = self.output_result (i);
 let outparameteri = self.operation.out_parameter->at(i);
 outparameteri.type.isCompatibleWith (resulti.type);
 outparameteri.multiplicity.isCompatibleWith (resulti.multiplicity))`

Note: Classifier::isCompatibleWith (Classifier):Boolean needs to be defined if it doesn't already exist. The first must be a descendent of the second (with appropriate handling of multiple classification).

InviteAction

An invitation instance is created and initialized with the given values and sent to the target active object (there must be exactly one target object). The sender is blocked until the target object handles the invitation and gives a reply, which includes the return values specified by the invitation. The invitation is treated as an event by the active object and may trigger a transition. Any transition triggered by the invitation must have exactly one procedure attached to it (not to an entry or exit). The procedure must have a signature that matches the signature of the invitation. The argument values of the invitation are made available to the procedure execution. When the procedure execution completes, its result values are copied into the a reply instance, which is returned to the caller. When the reply is received, the sender places the return values on the output pins of the invite action and the action is complete.

Attributes

- | | |
|--------------------|--|
| target: Object[*] | (inherited from SendAction) Exactly one target object (restricted from many in the Send-Action) . An invitation instance will be sent to the object. |
| argument: Value[*] | (inherited from SendAction) The attribute values of the invitation instance. Their number and types must match the attributes defined for the invitation.. |

Associations

- | | |
|--------------------|---|
| signal: Invitation | (inherited from SendAction) The kind of signal to be sent. This attribute is inherited from SendAction, but in the case of an InviteAction is further restricted to be an Invitation (not just any Signal). |
|--------------------|---|

Outputs

- | | |
|------------------|--|
| result: Value[*] | (inherited from CallAction) The results produced by executing a procedure on the given arguments. The number and types of the results must match the procedure. The procedure is attached to a transition triggered by the invitation. |
|------------------|--|

Well-formedness rules

- [1] The invitation must be defined for the type of the target pin.
`self.target.type.receptions->includes (self.signal)-- Note: receptions is similar to "operations", must be added to OCL to select out a given kind of feature`

- [2] The number, types, and multiplicities of the input argument pins must be compatible with the number, types, and multiplicities of the attributes of the invitation. (Assumes the attributes of the invitation are ordered.)

```
self.input_argument()->size() = self.signal.attribute()->size();
Sequence { 1..self.input_argument()->size() } -> forAll (i:Integer |
  let argumenti = self.input_argument (i);
  let attributei = self.signal.attribute()->at(i);
  argumenti.type.isCompatibleWith (attributei.type);
  argumenti.multiplicity.isCompatibleWith (attributei.multiplicity))
```

- [3] The number, types, and multiplicities of the output result pins must be compatible with the number, types, and multiplicities of the result parameters of the invitation.

```
self.output_result()->size() = self.invitation.result->size()
Sequence { 1..self.result()->size() } -> forAll (i:Integer |
  let resulti = self.output_result (i);
  let outparameteri = self.invitation.result->at(i);
  outparameteri.type.isCompatibleWith (resulti.type);
  outparameteri.multiplicity.isCompatibleWith (resulti.multiplicity))
```

Postcondition:

InvitePacket

A unit of information transmitted from an invite action to a target active object. It has a target object reference and it contains a signal instance. When it is received by the target object, it causes a signal event to be processed by the active object. Handling of events by objects is described separately under state machines.

Attributes

status: enum {ready, triggered, ignored, completed}-- whether the packet has been processed

Associations

target: ActiveObject -- the object to receive a signal event

signalInstance: SignalInstance-- the instance of the signal to be delivered

Lifecycle

ready -> triggered / start the execution of the triggered procedure (handled by the state machine chapter)

triggered -> complete / copy the results from the triggered procedure to the packet (handled by the state machine)

ready -> ignored (handled by the state machine if it fails to trigger a transition)

Production 1: Signal processing is independent of the sender. Eventually the signal packet is handled by the target object (usually by its state machine). The handling process is described in the state machine chapter.

Comment: Some time may elapse between sending and receipt, but that is true of all actions.

SendAction

A signal packet is created and initialized with the given values and sent to each object in the target set. A separate copy of the signal packet is sent to each object. The sender may proceed immediately. There are no guarantees about transmission time, and it may vary for objects in the target set. Each signal packet remains active until it is processed by its target object, usually by triggering or failing to trigger a transition in its state machine..

Attributes

None

Associations

signal: Signal The kind of signal to be sent.

Inputs

target: Objec[*]t The set of target objects. A signal instance will be sent to each object in the set.

argument: Value[*] The attribute values of the signal instance. Their number and types must match the attributes defined for the signal..

Note. It must be possible to determine the target of a call at run-time (e.g., if a remote procedure call is sent to another object).

Outputs

none

Well-formedness rules

```
context Classifier::attribute (): Attribute[*]
post: result = self.feature->select (oclIsKindOf (Attribute))
```

[1] The number, types, and multiplicities of the input argument pins must be compatible with the number, types, and multiplicities of the attributes of the signal. (Assumes the attributes of the signal are ordered.)

```
self.input_argument()->size() = self.signal.attribute()->size();
Sequence { 1..self.input_argument()->size() } -> forAll (i:Integer |
  let argumenti = self.input_argument (i);
  let attributei = self.signal.attribute()->at(i);
  argumenti.type.isCompatibleWith (attributei.type);
  argumenti.multiplicity.isCompatibleWith (attributei.multiplicity))
```

SendPacket

A unit of information transmitted from a sender action to a target active object. It has a target object reference and it contains a signal instance. When it is received by the target object, it causes a signal event to be processed by the active object. Handling of events by objects is described separately under state machines.

Attributes

status: enum {ready, completed}-- whether the packet has been processed

Associations

target: ActiveObject the object to receive a signal event

signalInstance: SignalInstance the instance of the signal to be delivered

Production 1: Signal processing is independent of the sender. Eventually the signal packet is handled by the target object (usually by its state machine). The handling process is described in the state machine chapter.

Signal

A class of information communicated asynchronously as a unit from an object to an active object. A signal has a set of attributes, which can equivalently be interpreted as a list of parameters. The receipt of a signal by an active object is an event that is added to its event queue. The communication of signals among objects may take time.

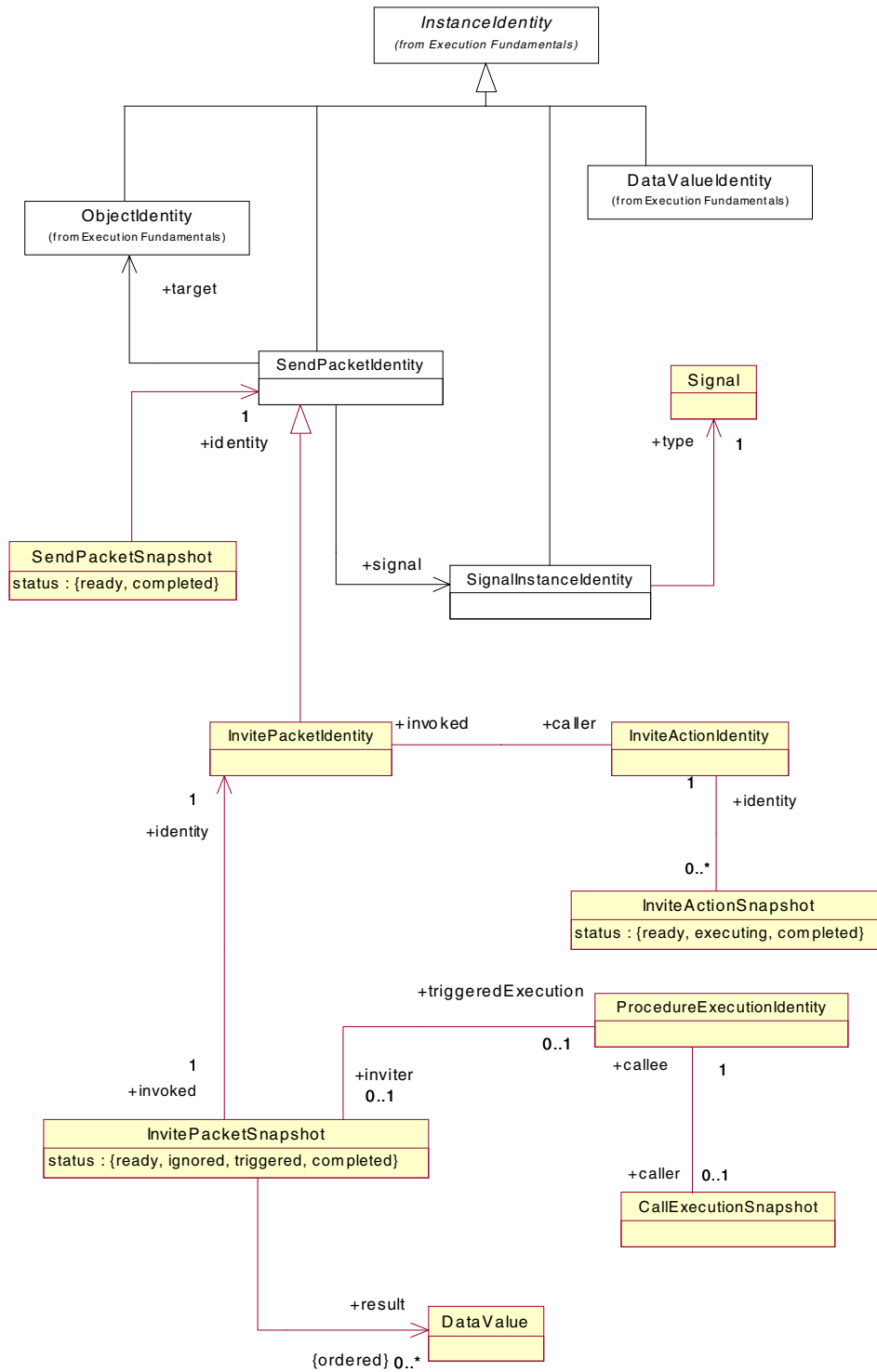


Figure 42. Messaging classes execution metamodel

11.5. Execution Model Classes

CallActionExecution

The execution of a call action.

Lifecycle

ready -> executing / create the subordinate procedure execution and copy over the arguments
 executing [subordinate procedure execution complete] -> complete / copy back the results

Semantics

Production 1: Call action generates a subordinate procedure execution and action execution and blocks itself

Precondition: self.status = ready

Assertions:

- [1] The operation must be defined for the actual class of the target object
 self.input_target ().class.allSupertypes.operations->includes (self.operation)
- [2] The number and types of actual arguments must match the number and types of the declared parameters of the operation. Normal polymorphism rules apply.

```
self.action.operation.in_parameter->size() = self.action.input_argument()->size()
Sequence { 1 .. self.action.input_argument()->size()->forall (i:Integer |
  let argumenti = self.input_argument (i);
  let inparameteri = self.action.operation.in_parameter->at (i);
  argumenti.class.isCompatibleWith (inparameteri.type) -- the type of the actual argument must match
  parameteri.multiplicity->inRange (argumenti->size()) -- cardinality must match multiplicity
}
```

Postcondition: self.status = executing -- put call in executing state while subordinate execution proceeds

procExec: ProcedureExecution.isNew-- create a subordinate procedure execution

self.invoked = procExec-- the new procedure execution is a dependent

procExec.caller = self

procExec.status = ready

procExec.host = self.input_target ()

procExec.result.isEmpty ()

procExec.actionExecution.isEmpty ()

procExec.variableExecution.isEmpty ()

Sequence { 1 .. nparameters }->forall (i:Integer |

procExec.argument->at(i).value = self.input_in_parameter(i)--copy the input parameters

procExec.procedure = self.input_target ().methodLookup (self.action.operation)-- look up the method in the classes of the object

context Object::method-lookup (operation: Operation): ProcedureAction

-- Selects a ProcedureAction based on Operation.

-- If no unique ProcedureAction can be found, the model is ill-formed.

Production 2: Subordinate procedure execution is complete, copy back the results and finish the call action execution

Precondition: self.status = executing

self.invoked.status = completed

Assertions:

- [1] The types and cardinalities of the result values must match the declaration of the operation.

```
self.action.operation.out_parameter->size() = self.action.output_result()->size()
```

```
Sequence { 1 .. self.action.output_result()->size() } ->forall (i:Integer |
```

```
let resulti = self.output_result (i);
```

```
let outparameteri = self.action.operation.out_parameter->at (i);
```

```
outparameteri.class.isCompatibleWith (resulti.type) -- contrapositive match
```

```

        outparameteri.multiplicity->inRange (resulti->size()) -- cardinality must match multiplicity
    }
}
Postcondition: self.status = completed
Sequence { 1 .. self.action.operation.out_parameter->size()->forall (i:Integer |
    self.output_result (i) = procExec.result->at(i).value)--copy the output results

```

InviteActionExecution

The execution of an invite action. This takes parts from both call and send.

Lifecycle

ready -> executing / create the invite packet and copy over the arguments

executing [invite packet complete] -> complete / copy back the results

Lifecycle

Production 1: Invite action generates an invite packet and blocks itself.

Precondition: self.status = ready

Assertions:

- [1] A reception for the invitation must be defined for the actual class of the target object.
self.input_target ().class.allSupertypes.receptions->includes (self.invitation)
- [2] The number and types of actual arguments must match the number and types of the declared attributes of the invitation. Normal polymorphism rules apply.
self.action.signal.attribute()->size() = self.input_argument()->size()
Sequence { 1 .. self.action.input_argument()->size()->forall (i:Integer |
 let argumenti = self.input_argument (i);
 let attributei = self.action.signal.attribute->at (i);
 argumenti.class.isCompatibleWith (attributei.type) -- the type of the actual argument must match
 attributei.multiplicity->inRange (argumenti->size()) -- cardinality must match multiplicity
 }

Postcondition: self.status = executing -- put invite in executing state while subordinate execution proceeds
invitePacket: InvitePacket.isNew-- create a subordinate invite packet
invitePacket.caller = self
invitePacket.status = ready
Sequence { 1 .. nparameters }->forall (i:Integer |
 invitePacket.argument->at(i).value = self.input_in_parameter(i))--copy the input parameters
invitePacket.target = self.target
invitePacket.signal = self.signal
self.invoked = invitePacket

Production 2: Subordinate invite packet is complete, copy back the results and finish the invite action execution

Precondition: self.status = executing
self.invoked.status = completed

Assertions:

- [1] The types and cardinalities of the result values must match the declaration of the operation.
self.action.signal.result->size() = self.action.output_result()->size()
Sequence { 1 .. self.action.output_result()->size() } ->forall (i:Integer |
 let resulti = self.output_result (i);
 let outparameteri = self.action.signal.result->at (i);
 outparameteri.class.isCompatibleWith (resulti.type) -- contrapositive match
 outparameteri.multiplicity->inRange (resulti->size()) -- cardinality must match multiplicity
 }

Postcondition: self.status = completed

```
Sequence { 1 .. self.action.signal.result->size()->forall (i:Integer |
  self.output_result (i) = self.invoked.result->at(i).value)--copy the output results
```

Production 3: Subordinate packet has been ignored, cause an exception

Precondition: self.status = executing and

self.invoked.status = ignored

self.status = exception-- this will need to be worked out when exceptions are added

SendActionExecution

The execution of a send action. There are two independent postconditions..

Lifecycle

ready -> complete / create the send packet copy over the arguments

Semantics

Production 1: Send action generates a send packet and sends copies of it to each target object. The postcondition creates SendPackets in the ready status; these objects then trigger productions under the SendPacket class.

Precondition: self.status = ready

Assertions:

[1] The number and types of actual arguments must match the number and types of the declared attributes of the signal. Normal polymorphism rules apply.

```
self.action.signal.attribute()->size() = self.input_argument()->size()
```

```
Sequence { 1 .. self.action.input_argument()->size()->forall (i:Integer |
```

```
  let argumenti = self.input_argument (i);
```

```
  let attributei = self.action.signal.attribute->at (i);
```

```
  argumenti.class.isCompatibleWith (attributei.type) -- the type of the actual argument must match
```

```
  attributei.multiplicity->inRange (argumenti->size()) -- cardinality must match multiplicity
```

```
  }
```

Postcondition: self.status = complete -- sender may proceed once send packets are built

```
self.target->forall (destination: Object |
```

```
  let sendPacket: SendPacket.isNew;-- create a signal instance
```

```
  sendPacket.status = ready;
```

```
  sendPacket.target = destination;
```

```
  Sequence { 1 .. self.action.input_argument()->size()->forall (i:Integer |
```

```
    sendPacket.signalInstance->at(i).value = self.input_in_parameter(i)--copy the input parameters
```

```
  )
```

Part IV. Appendices

There are two appendices. The first describes changes to UML suggested by this submission. The second provides some examples of mapping from languages into the action semantics.

Appendix A. Updates to UML

This appendix gives updates to UML specified by the submission. There are various kinds of update:

- Changes, additions, or deletions to the semantics as expressed in the abstract syntax, textual descriptions, or well-formedness rules. It includes clarification of areas where UML is imprecisely specified. Since imprecision affords modelers leeway that they would no longer have under this submission, clarification is considered a change in UML semantics.
- Changes in the way a modeler does things. This is not a change in semantics, but requires a change in user models for them to mean the same thing under this submission that they do in UML currently.
- Areas of the UML that the action semantics does not address. In these areas, the semantics of action may be defined by the existing UML or not. In any case, users will want to know what applications would not have action semantics defined under this submission.

The updates refer to UML 1.3, but will refer to UML 1.4 in the final submission. Section headings follow the package structure of the UML abstract syntax, and the metamodel element being updated under those. Only semantics is covered, assuming that notation will be changed as necessary.

Undefined Semantics

There are two kinds of undefined semantics in this submission: those that *remove* semantics from UML, and those that *leave* the semantics as defined by UML. The internal specification of each action may remove semantics from UML. For example, the semantics of actions on features with classifier targetScope is undefined in this submission, and would remain undefined after it is integrated with UML. However, the way actions are used leave the semantics to UML. For example, the semantics of using actions in collaborations, or in state machines used as methods, is undefined in this submission, and the semantics are inherited from UML.

Execution Semantics in UML

A general topic for integrating this submission in UML is that UML currently has no abstract syntax for its execution semantics. All UML classes are instantiated into the user model, that is, they are metaclasses, and are interchanged between modelers. UML metaclasses are not instantiated at runtime. The execution model of this submission, on the other hand, is not instantiated into user models and consequently requires a new category of UML model that is not interchanged. The purpose of the execution model is to formally express behavioral semantics previously specified only in text. It is assumed here that the UML would have a new category of models to which XMI is not applied.

A.1. Foundation

Datatypes

Expressions

Since actions are specified with expressions in UML, this area is fundamental to integrating the submission with UML. There are a number of problems to be addressed:

- Expressions currently are represented in UML models as strings (the Expression metaclass has a body attribute of type String). Users will want to enter strings as the surface syntax, regardless of metamodels introduced for action semantics. Changing this arrangement would also pose an upward migration problem for existing models. Users will not have a standard way to translate their expression strings to the action semantics of this submission, since there are no normalized language bindings proposed. In addition, even if it were decided that an action semantics were to be used for expressions, it would not be applicable to all cases,

for example MappingExpression in the Abstraction dependency, which models mappings of one layer to another.

- OCL is a good language for boolean expressions, such as guards. It is not the intent of this submission to prevent models from having OCL expressions. However, there is an issue with how OCL gets access to data in its scope. It is necessary to resolve this and perhaps other issues with OCL to satisfy the requirements for precision to which this submission is responding.
- The various datatypes that are the result of expressions are not defined in UML. For example, there is no subtype of Datatype called Boolean, in fact, there no subtypes of Datatype at all. This means users will all define their own Boolean, Integer, etc, breaking interchangeability.

A partial solution proposed here is to introduce a association from Expression to Procedure and copy Expression's attributes onto Procedure, as shown in Figure 43. Procedures are the way control and dataflow behaviors are modeled in this submission. The proposed metamodel supports expressions used in cases where procedures may not be applicable, such as MappingExpression, while still supporting procedures to specify an expression where it makes sense, and giving modelers the option of writing procedures in a surface syntax. The attributes of Expression are copied onto Procedure because some procedures are not expressions, namely those with side-effects, so cannot be modeled with expressions.

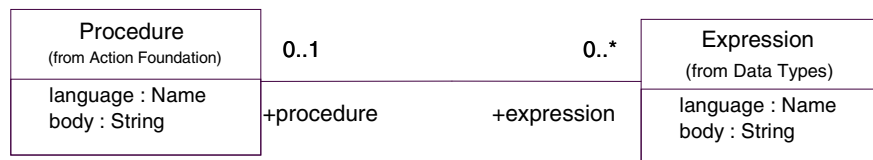


Figure 43. Procedure and Expression

All uses of the following expression types are affected by the integration proposed in this appendix:

- ProcedureExpression
- ActionExpression
- IterationExpression
- ObjectSetExpression

Some uses of these expression types are affected:

- BooleanExpression (in state machine transition guards)

Details are given in the rest of the appendix.

Core

Feature

The body attribute is removed from Method and replaced with an association between Procedure and Method, as shown in Figure 44.

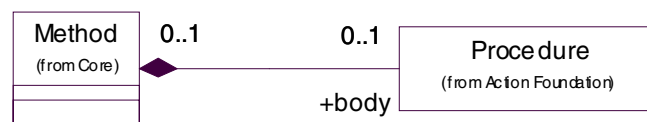


Figure 44. Procedure

Inputs and outputs of procedures cannot be marked with the parameter kinds `inout` and `return` as in UML, but the parameters of operations are mapped onto inputs and outputs in such a way that `inout` and `return` are supported. See the Class `CallActionExecution` in Chapter 11, “Messaging Actions” for more detail.

UML gives semantics to features in an area that is not covered by actions in this submission:

- Attributes with classifier `targetScope`.

UML is ambiguous in an area of features that is not defined in this submission either:

- The semantics of attributes with classifier `ownerScope` declared on classifiers that have children is not completely defined by the actions. It is not defined whether this means a single value for the classifier and all its descendants, or a value for the classifier and each descendant.

UML has an informally expressed well-formedness rule that is loosened in this submission:

- The current UML semantics for method inheritance is that a model with multiply-inherited methods is ill-formed (UML 1.3 Semantics document, section 2.5.4, Inheritance heading, third paragraph, second-to-last sentence, page 2-61, presumably this means multiple methods for the same operation). The restriction makes it impossible for profiles to specialize method lookup to a particular language semantics. This submission removes the restriction, leaving only the constraint that a single method be selected. Profiles of the action semantics can specify their own method lookup operation used by call actions. See the Class `CallActionExecution` in Chapter 11, “Messaging Actions” for more detail.

Association

A number of areas in associations have been defined in this submission where the UML is ambiguous:

- Links are not mutable once they are created, except that they can be destroyed and reordered. Qualifier values, end objects, and link classifier, if any, cannot be changed once a link is created.
- For associations of arity higher than two, the semantics for ordering, changeability, and visibility of association ends is defined by analogy to n-ary multiplicity: the constraint on an end is applied to the objects on the other association ends, holding those objects and qualifiers on all ends constant.
- The multiplicity of qualifier attributes is always 1..1.
- Navigability applies only to reading, but not when reading link objects.
- Visibility applies to both reading and writing, but reading is only restricted on the end being read, whereas writing is restricted on all ends. Visibility does not apply to reading link objects.

UML is ambiguous on the following semantics of associations that are not defined in this submission either:

- As so cation ends with classifier `ownerScope` declared on classifiers that have children is not completely covered by the actions. It is not defined whether this means a single value for the classifier and all its descendants, or a value for the classifier and each descendant.

UML gives semantics for the following aspect of associations that is not covered by actions defined in this submission:

- Associations that have any ends with classifier `targetScope`.

It is noted by this submission that associations with arity greater than two with ends of both classifier and instance `targetScope` are ill-formed. For example, suppose a ternary association has one end with classifier `targetScope` and the others with instance `targetScope`. Those with instance `targetScope` are incorrectly marked as such, because the ends opposite them have both class and instance values. This is an additional well-formedness rule for associations in UML:

```
self.connections→size() = 2 or self.connections→forall(end1, end2 | end1.targetScope = end2.targetScope)
```

UML does not give semantics for the following aspect of associations that is covered by actions defined in this submission:

- Violating minimum multiplicity is given a semantics equivalent to the lower multiplicity being zero.

See Chapter 8, “Read and Write Actions” for more information.

Classifier

UML 1.3 defines an active object as one that has its own thread of control and runs concurrently with other active objects. This submission changes the definition to an object that runs concurrently with other active objects, which allows it to accept asynchronous signals and otherwise respond to the occurrence of incoming events.

A.2. Behavioral Elements

Common Behavior

Action

The Actions metamodel in UML is replaced by the metamodel defined in this submission. Action and ActionSequence are replaced by Procedure everywhere they are used in UML. The Gamma pattern in UML between Action and ActionSequence is pushed down one level in this submission. Procedure has only an association to Action, it is not a kind of Action or vice-versa. The Gamma pattern is used between Action and GroupAction. See Figure 46.

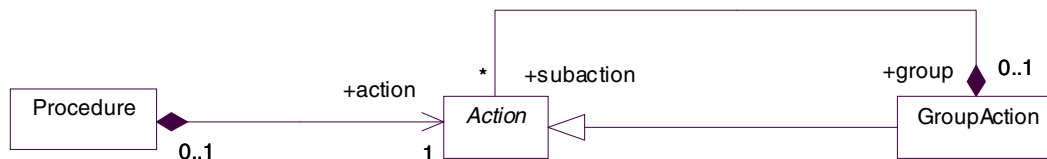


Figure 45. Procedures and Actions

The attributes of Action are removed: recurrence, target, isAsynchronous, and script. Their functionality is still mostly available in the action semantics. Recurrence is done with the collection actions, target is handled in messaging actions, isAsynchronous is handled by messaging actions or by constructing the appropriate flows with the action foundation, and script is also handled by defining flows with the action foundation. One consequence is that there is no semantics defined for asynchronous calls, whereas UML explicitly defines this for CallAction. Asynchronous calls in this submission are modeled by parallel control flow that to a call, which in turn has no outgoing flows.

One exception to the replacements above is the Argument metaclass currently in UML. The relation of Argument to the action semantics is not determined yet. This submission supports the functionality of Argument only for state machines and methods. However, Argument is also used by collaborations currently in UML, which is an issue to be resolved.

The class hierarchy of actions is shown in Figure 46 and Figure 47.

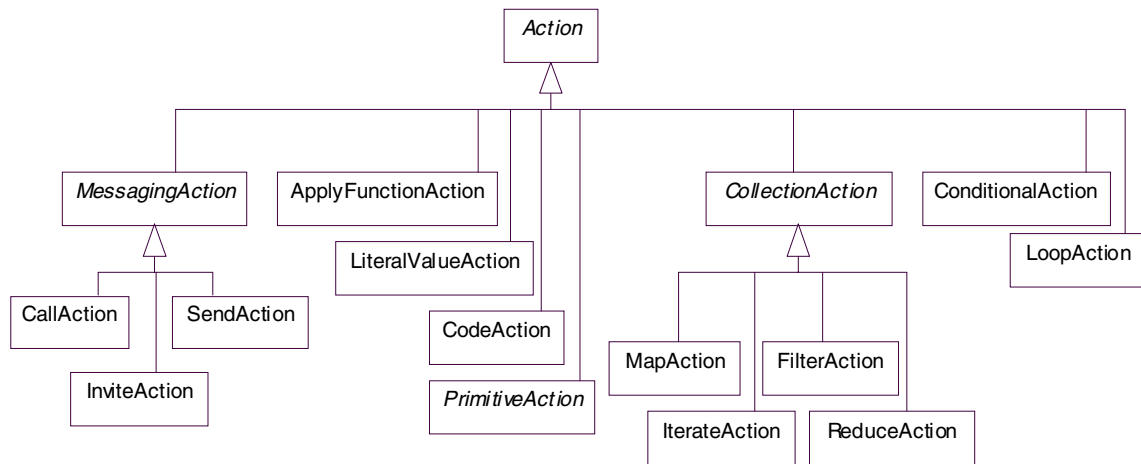


Figure 46. Messaging, Computation, Collection, and Composite Actions

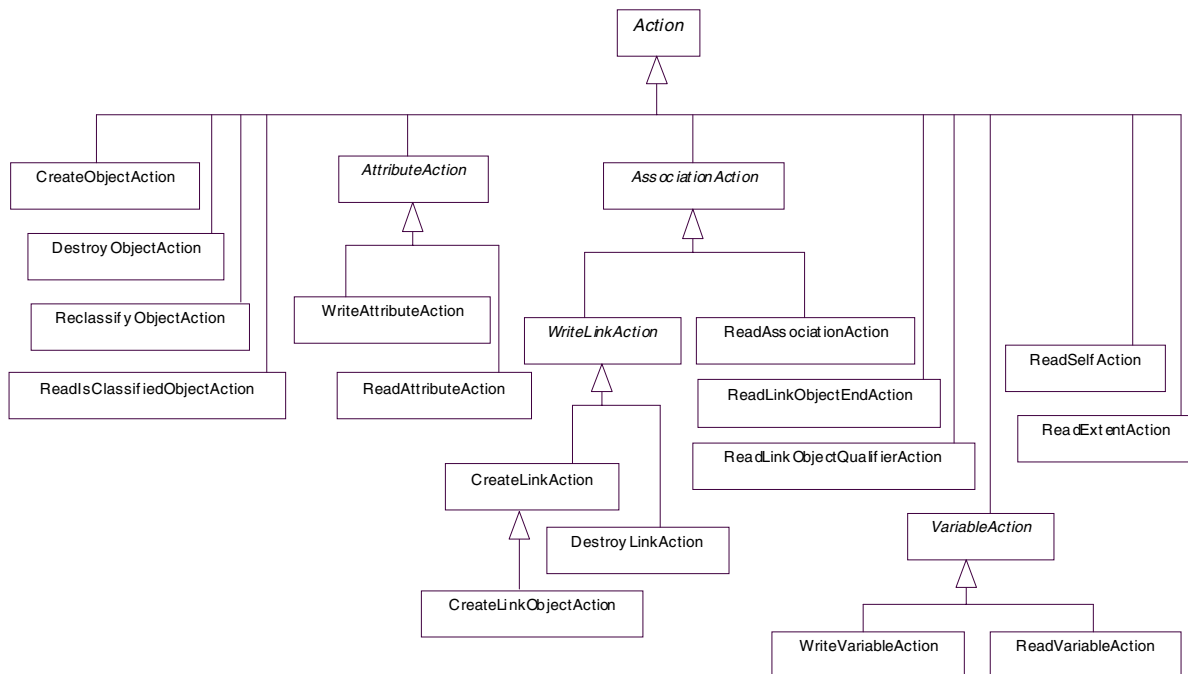


Figure 47. Read and Write Actions

Signal

Invitations are new kind of signal proposed by this submission. They are synchronous signals that support returning values from actions on transitions triggered by the invitation. Invitations are sent using the InviteAction shown in Figure 46. A corresponding new type of event is also proposed, namely, InviteEvent, which is associated with Invitation. Existing models of protocol state machines could be converted under this submission to send invitations from the

operations that are currently monitored by call events. An execution model is introduced that models the time it takes for a signal to arrive at its target. See Chapter 11, “Messaging Actions” for details.

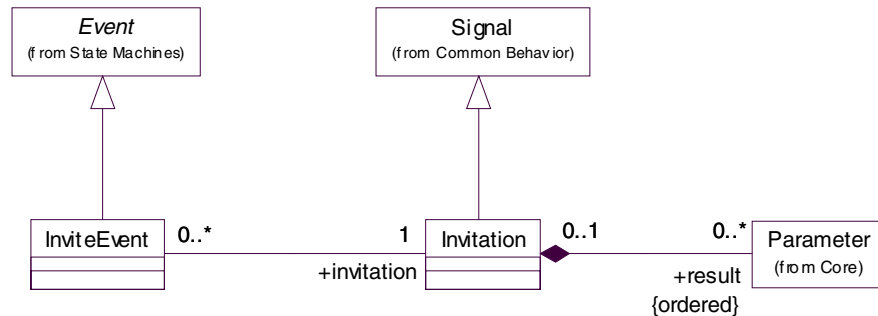


Figure 48. Invitations

State Machines

Undefined areas

This submission does not formalize all the semantics of state machines, only the parts necessary to define the proposed actions. It relies on the existing UML for the rest of state machine semantics, per the approach explained in Subsection “Undefined areas” above. For example, the semantics of deferred events, completion events, transition selection, dispatch priorities, state machines as methods, are not formalized. All of these are supported in UML, and this submission relies on the semantics defined there to cover actions used in these cases.

Event

The term “event”, when it refers to triggers on transitions, is replaced with “event specification”. The term “event instance”, when it refers the entity added to the queue of a state machine at runtime, is replaced with “event occurrence”. Events are not treated as classes by this submission, and so do not have instances.

Entry and Exit Actions

This submission adds a well-formedness rule to state machines. Procedures used in state machines are required to have compatible signatures with all the events that might trigger them, except for entry and exit procedures with no inputs. For example, the state machine in Figure 49 would be ill-formed under this submission. If the entry procedure for state T had no arguments, then it would be well-formed. In that case, the entry procedure would be executed with no arguments regardless of which event triggered it. UML, by contrast, currently states that “Any parameter values

associated with the current event are available to all actions directly caused by that event". The UML places no restriction on the relation of events to actions triggered by them.

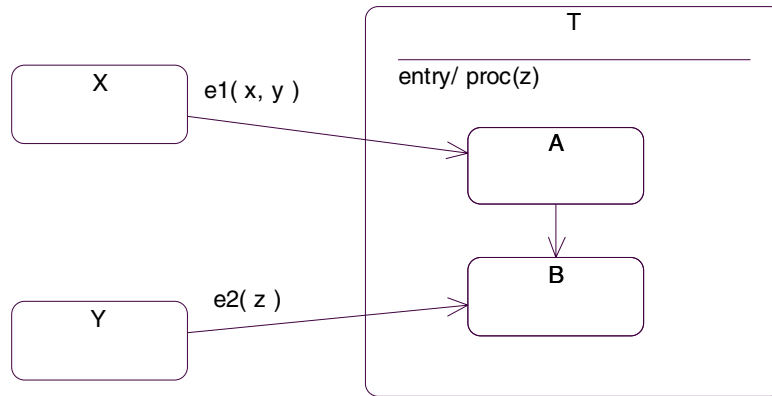


Figure 49. Ill-formed state machine

To support the above well-formedness rule, signals may be associated with the entry and exit procedures, as shown by the metamodel in Figure 50. If a signal is present on a procedure, then all events that might trigger the procedure must be events for that signal or one of its descendants. The procedure must have arguments compatible with the signal. If the signal is not present, then the procedure may not have any arguments. This model assumes that change events and time events have no arguments...

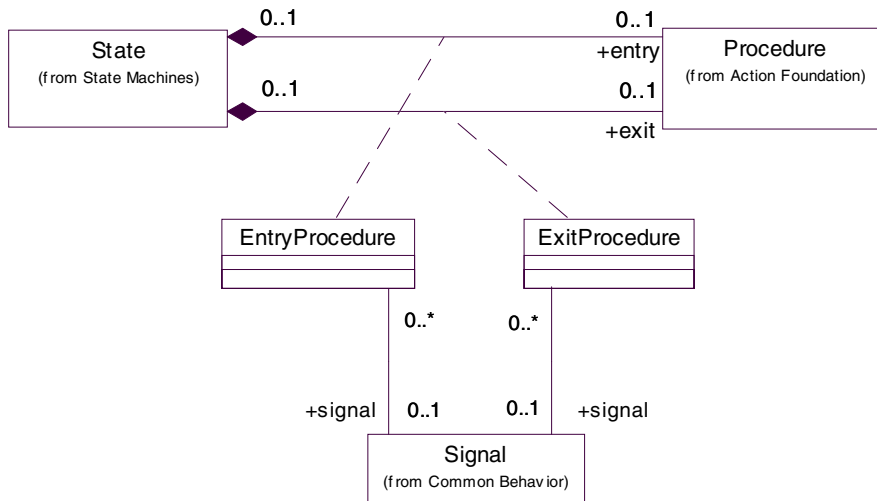


Figure 50. Signals on entry and exit procedures

Exit procedures may not have outputs. Internal transitions always have procedures.

Appendix B. Action Language Examples

This Appendix contains examples mapping from fragments of specifications in existing action languages to UML Action Semantics models. The intent of this is to demonstrate by example that the Action Semantics metamodel contains concepts and has a structure that supports the concepts and structure of real action languages currently in use.

In addition, such examples serve to aid the reader in further understanding the metamodels. By providing concrete examples in the familiar form of a textual language the reader can more readily see what some of the concepts actually mean. The reader should be aware, however, that the mappings are not definitive. In any discrepancy between the examples and the Action Semantics specification, the latter is the definitive reference.

B.1. The Action Languages

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

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

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

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

B.2. Presentation of the Examples

This appendix presents a series of examples, usually of one source statement in size. The examples consist of

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

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

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

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

The first series of examples tries to give examples that illustrate many of the individual metamodel elements of the Action Semantics: control structures (see Section B.3, "Control Structures"), object manipulation (see Section B.4, "Object Manipulation"), and messaging actions (see Section B.5, "Messaging Actions").

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

Unlike ASL and AL which have implicit variable declarations and typing, Kabira AS requires explicit declaration. Such declarations are not shown in the language examples here. It should be noted, however, that in the Action Semantics local variables have an association with GroupAction providing for scoping within an action language. The examples do not show this association.

The final section (see Section B.6, "A Complete Example"), provides a series of examples that together form a complete operation specification to illustrate how the individual model fragments are combined to give the meaning of a complete Procedure (remember, that model element Procedure replaces Action in the UML metamodel, see Chapter A.2, "Behavioral Elements"). These examples rely on the action language of SDL. They are based on the language mapping described in OMG document <http://cgi.omg.org/cgi-bin/doc?ad/00-08-01> (which can be found on the OMG web server).

To keep the examples manageable, we have simplified the resultant UML models in the manner that could have been performed by an intelligent compiler. We partially evaluated the resultant models where necessary information could be inferred based on the given input. For example, if it can be statically determined which test in the clauses of a ConditionalAction will uniquely evaluate to true, then the ConditionalAction can be replaced by the body of that clause.

Further, as described in Section 6.1, "Action Specification", instances of subclasses of the abstract metaclass Action are never connected directly to another action. They either connect via an associated ControlFlow, or via an associated OutputPin/DataFlow/InputPin chain. These intermediate classes are suppressed. We shall show an associated ControlFlow as a derived association between two actions with the role name taken from the association end connected to the action. We shall show an associated OutputPin/DataFlow/InputPin chain as a derived association between two actions with the role names taken from the association ends connected to the InputPin and OutputPin, respectively. We elide the special notation indicating a derived model element.

B.3. Control Structures

These examples are of the traditional control structures and sequential logic present in the action languages. These map on to the actions found in Chapter 7, “Composite Actions”.

If-then-else Logic

This example shows a simple if-then-else involving a logical comparison. The semantics of the action language construct is that if and only if the value of `factor` is true then some `action 1` will be executed. In all other cases, some `action 2` will be executed.

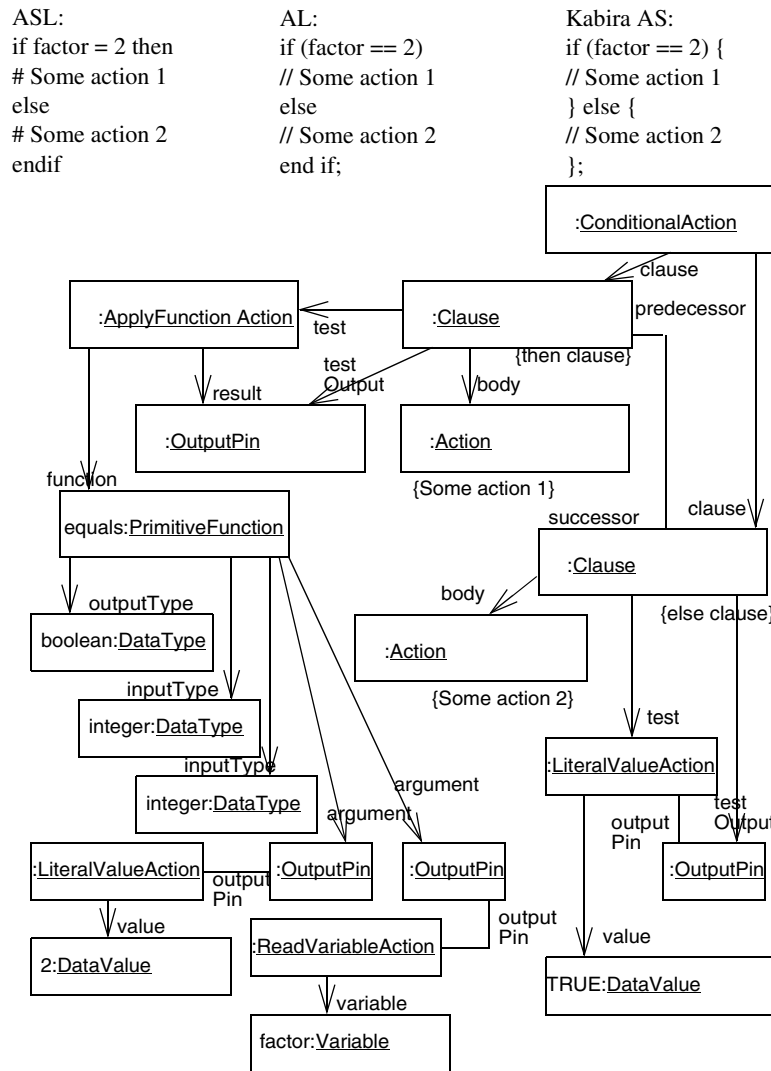


Figure 51. If-then-else Logic

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

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

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

Multi-way Decision

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

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

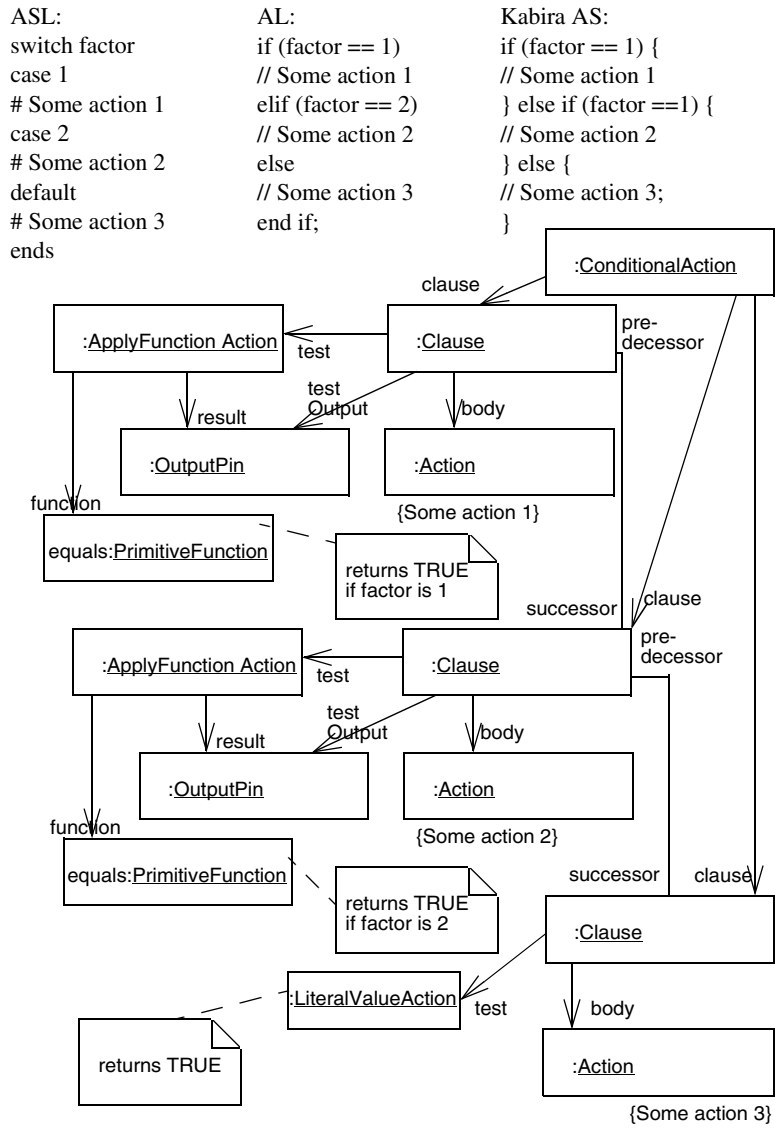


Figure 52. Multi-way Decision

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

In the example, the details of the test applied to each clause have been omitted to make the example clearer. However, the structure of these can be found by examining the details of the ApplyFunctionAction attached to the “then” clause of the if-then-else example in the previous section. Note also the predecessor-successor control flows between the clauses that provide the correct semantics for both the ASL switch and the explicit if-then-else construct.

B.4. Object Manipulation

These examples are concerned with the basic creation and manipulation of Objects. The actions used by these language constructs are for the most part those described in the chapter on Read and Write Actions. Being local variable oriented languages, all of these operation use local variables of type Object Reference (or sometimes collections of Object References).

Simple Object Creation

This shows an example of creating an Object of the Class `customer`. The reference to the newly created object is then assigned to the local variable `new_customer`.

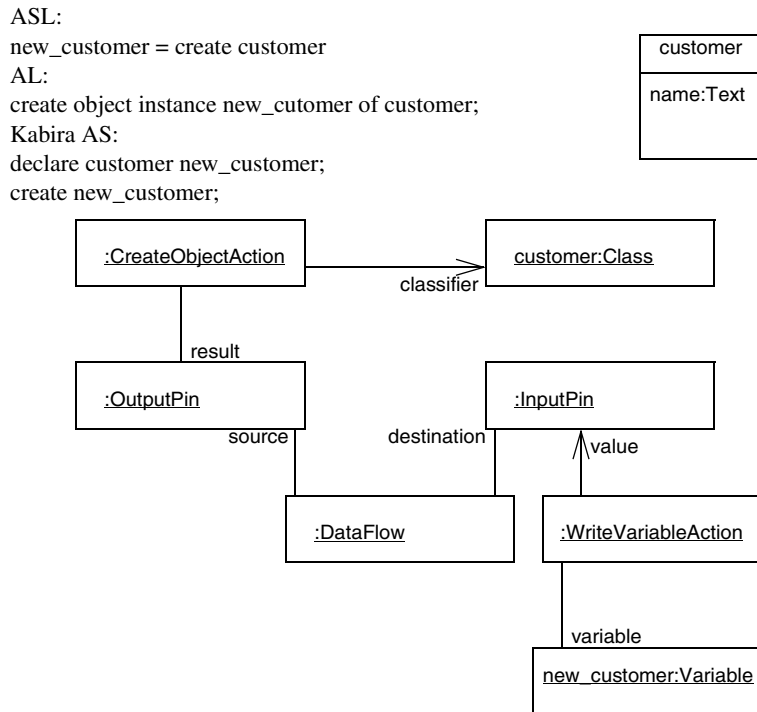


Figure 53. Simple Object Creation

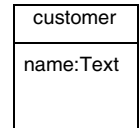
Note: AL actually uses a second form of the class name (the “key letter” captured as a UML tag) to identify the class. This is a minor syntactic detail and, in order to avoid confusion, the examples in this Appendix use the Class name instead.

Object Creation with Attribute Assignment

This shows an example of creating an Object of the Class customer with assignment of the value of the attribute name from the local variable new_name.

ASL:
 new_customer = create customer with name = new_name

AL:
 create object instance new_cutomer of customer;
 new_customer.name = new_name;



Kabira AS:
 declare customer new_customer;
 create new_customer values (name:new_name);

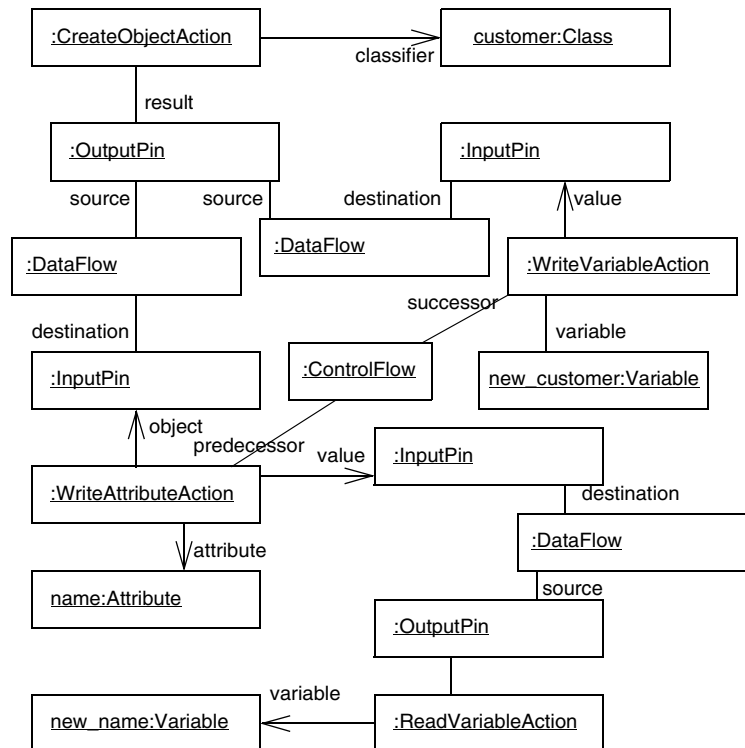


Figure 54. Object Creation with Attribute Assignment

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

Object Destruction

This shows an example of destruction of an object. In the examples, the object is identified by a reference my_customer.

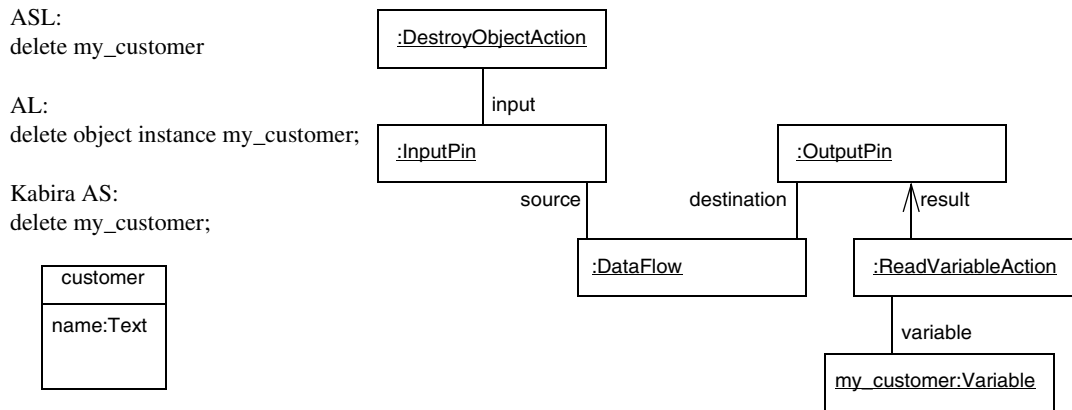


Figure 55. Object Destruction

Writing of Attributes: Single Attribute, Single Object

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

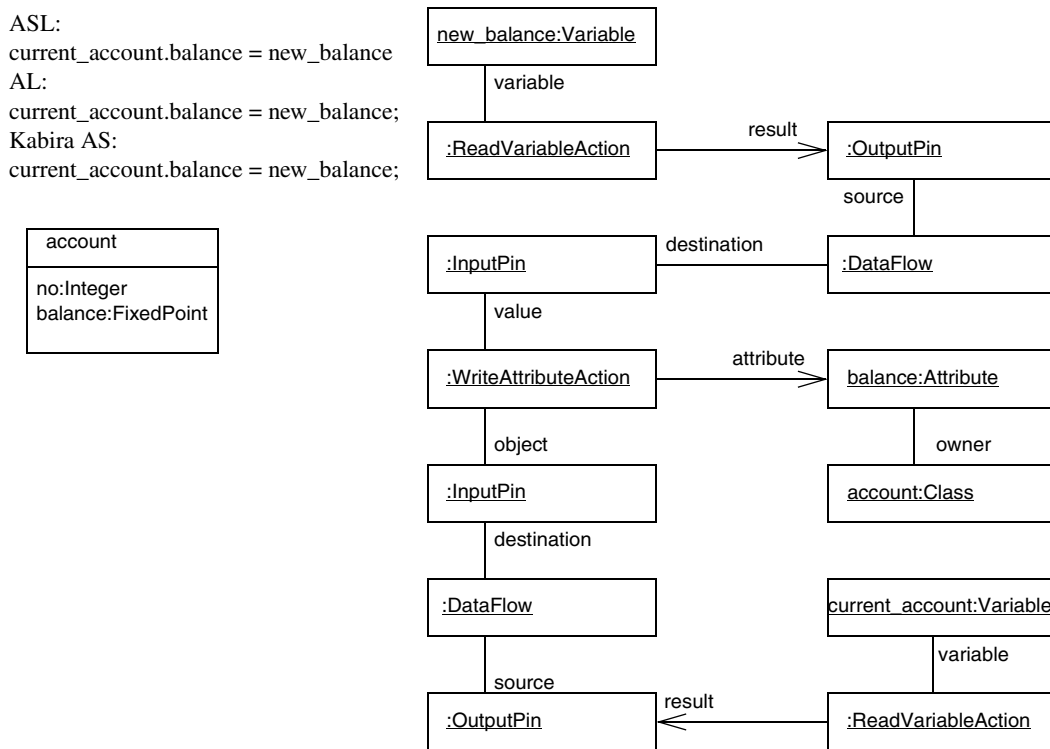


Figure 56. Writing of Attributes (Single Attribute, Single Object)

Writing of Attributes: Multiple Attributes, Single Object

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

this must be achieved through multiple statements, each operating from the same object instance. This example shows ASL only.

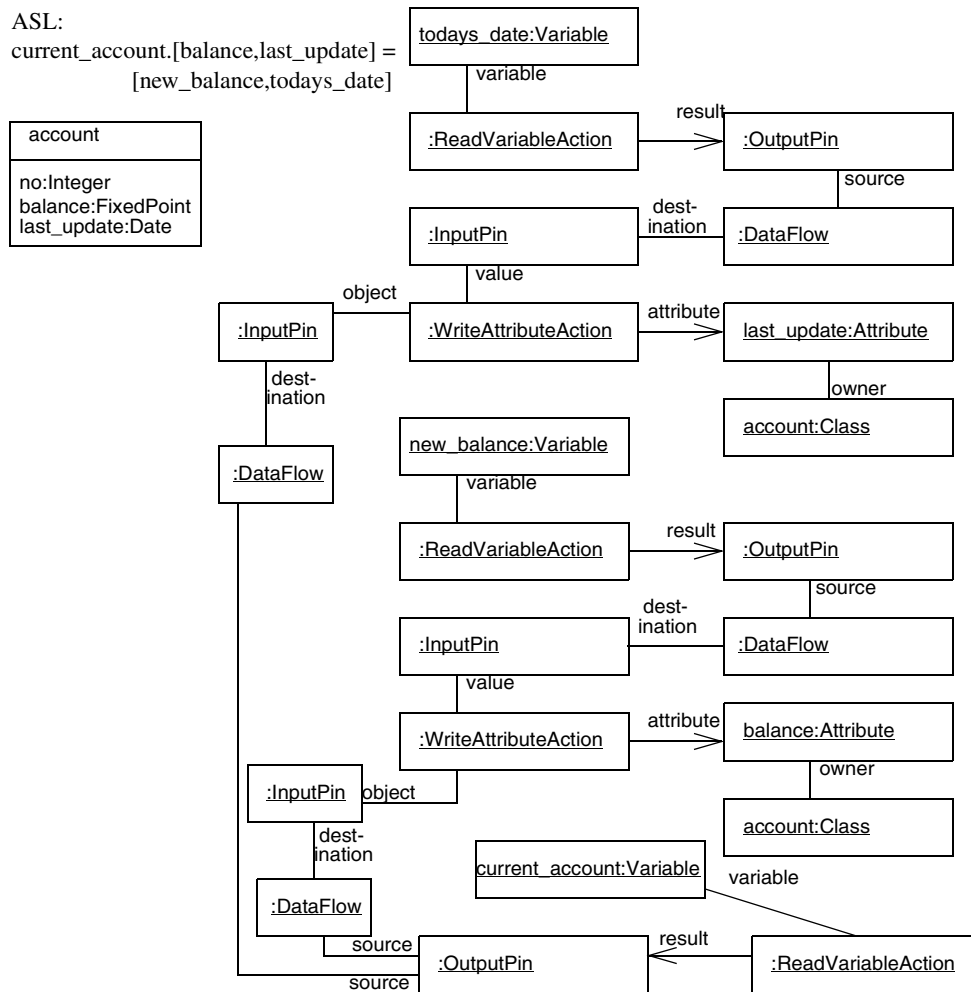


Figure 57. Writing of Attributes (Multiple Attributes, Single Object)

Writing of Attributes: Single Attribute, Multiple Object

In this example, a value is written to an attribute of a multiple object instances. The value comes from a local variable (todays_date), and the objects are identified by a object reference collection local variable ({reviewed_accts}). In ASL this can be achieved through a single language statement, but in AL and Kabira AS, this must be achieved through an explicit loop. The example shows ASL only and explicit loops are shown in other examples

NOTE: In this last example, the WriteAttributeAction gets its target object from each element of the MapAction input collection in turn. I am not sure how to show this on the diagram. Any thoughts?

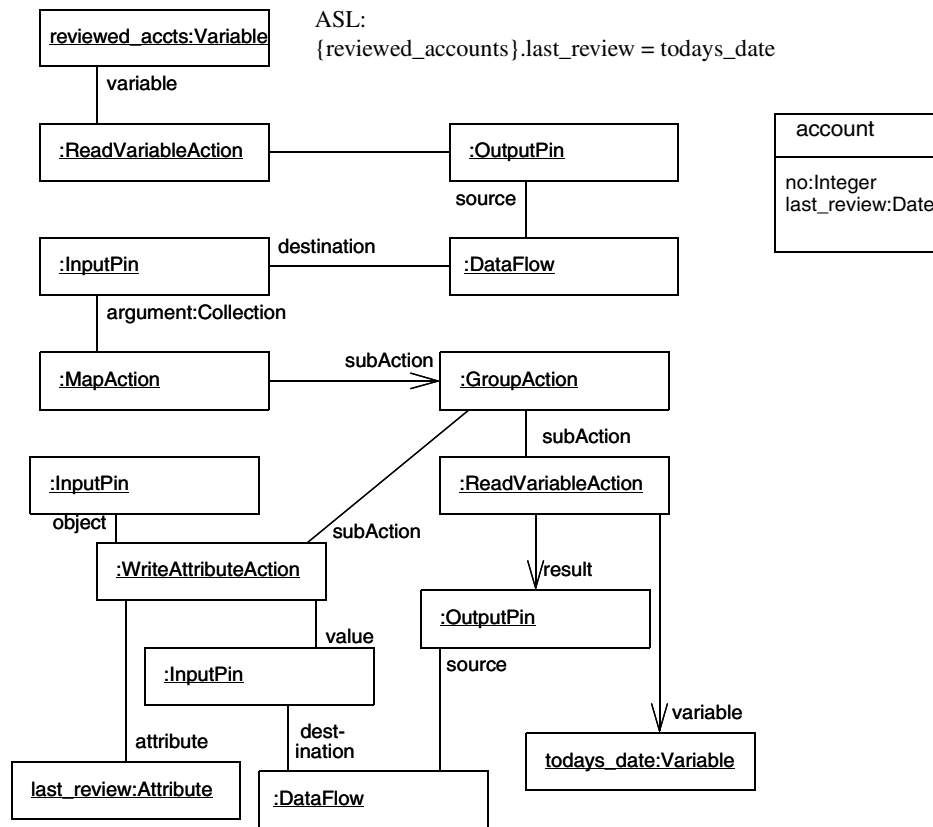


Figure 58. Writing of Attributes (Single Attribute, Multiple Objects)

Obtaining a Selection of Objects

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

In the first example, a selection is made from the `account` class according to a simple logical condition. The condition is such that this results in a single object instance being selected and the reference assigned to a local variable (`my_account`).

The top part of the instance diagram is concerned with the reading of the extent of the `account` class and flowing the resulting collection into a filter action. The filter action produces an output which is written to the `my_account` local variable. This is shown on the right hand middle of the diagram. The remainder of the diagram is concerned with the subAction of the FilterAction. This is an ApplyFunctionAction which invokes the logical comparison of the two items in the logical expression. One of the items is a constant (supplied by the LiteralValueAction) and the other is obtained by reading the value of the attribute of the instance being tested.

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

ASL:
 my_account = find-only account where no = 42
 AL:
 select one my_account from instances of
 account where selected.no = 42;
 Kabira AS:
 select my_account from account
 where(my_account.no = 42);

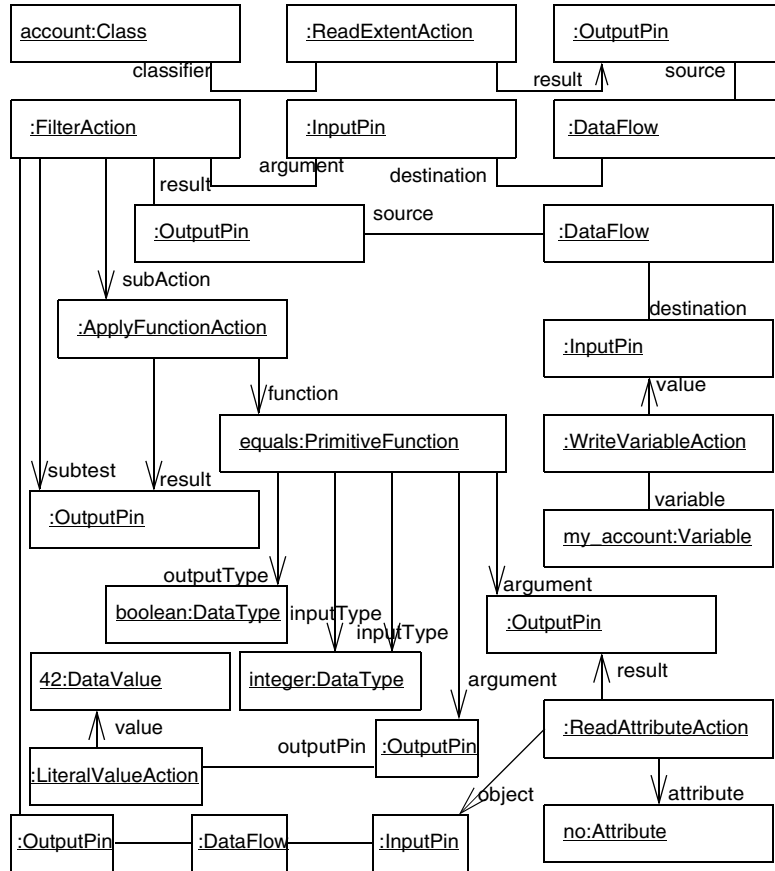
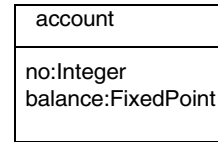


Figure 59. Single Object Selection

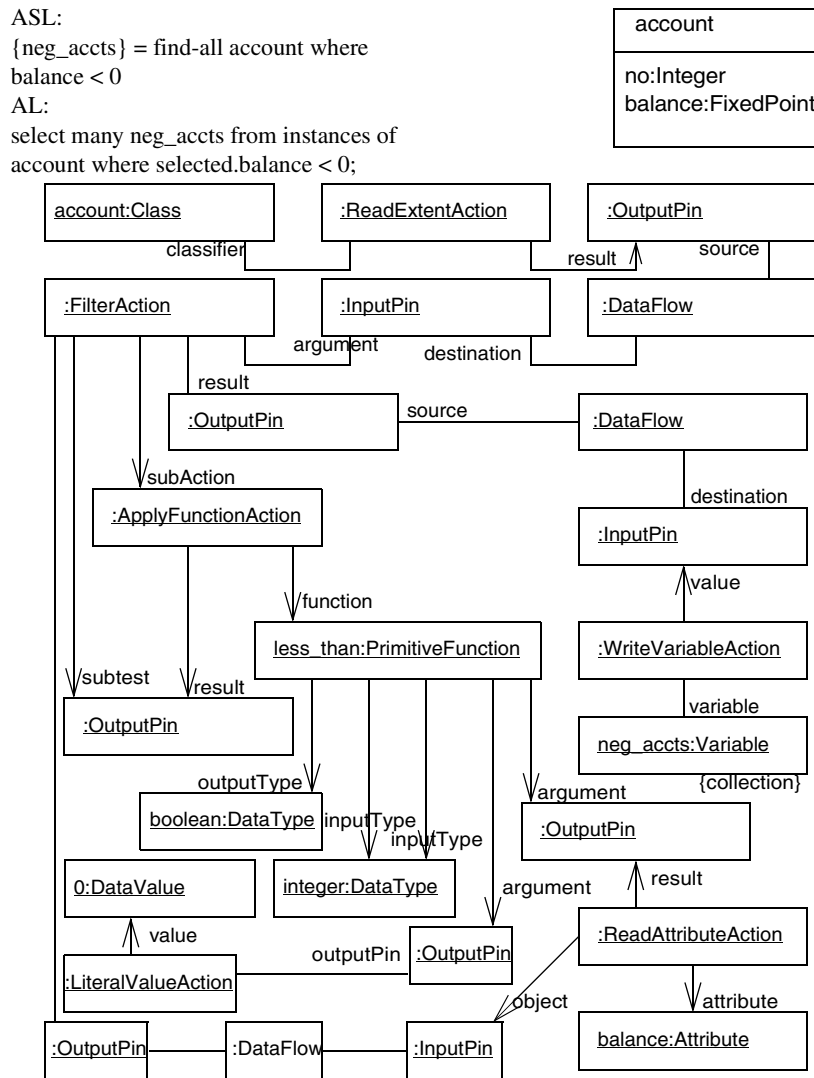


Figure 60. Obtaining a Collection of Objects

Creating a Link

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

All three of the languages make use of Role names where they are required to disambiguate an operation. For example, in a reflexive association it is necessary to know in which direction the link is intended. Role names will clarify this.

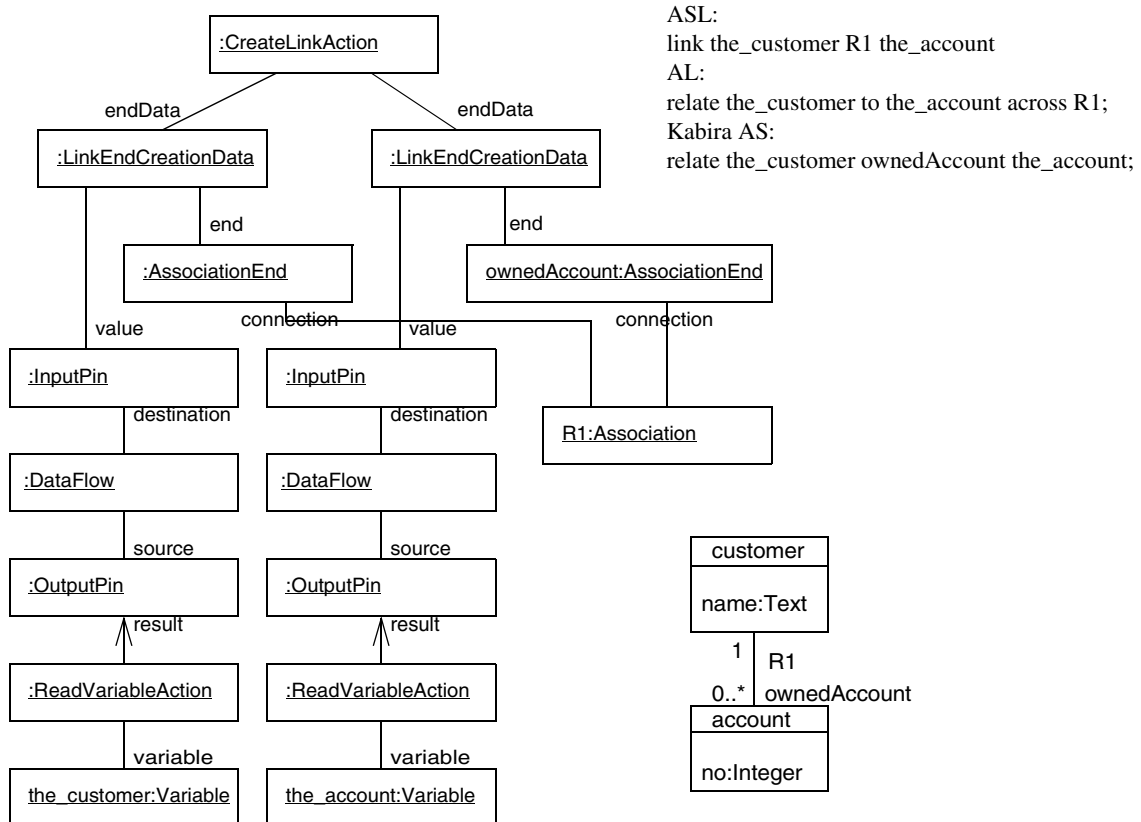
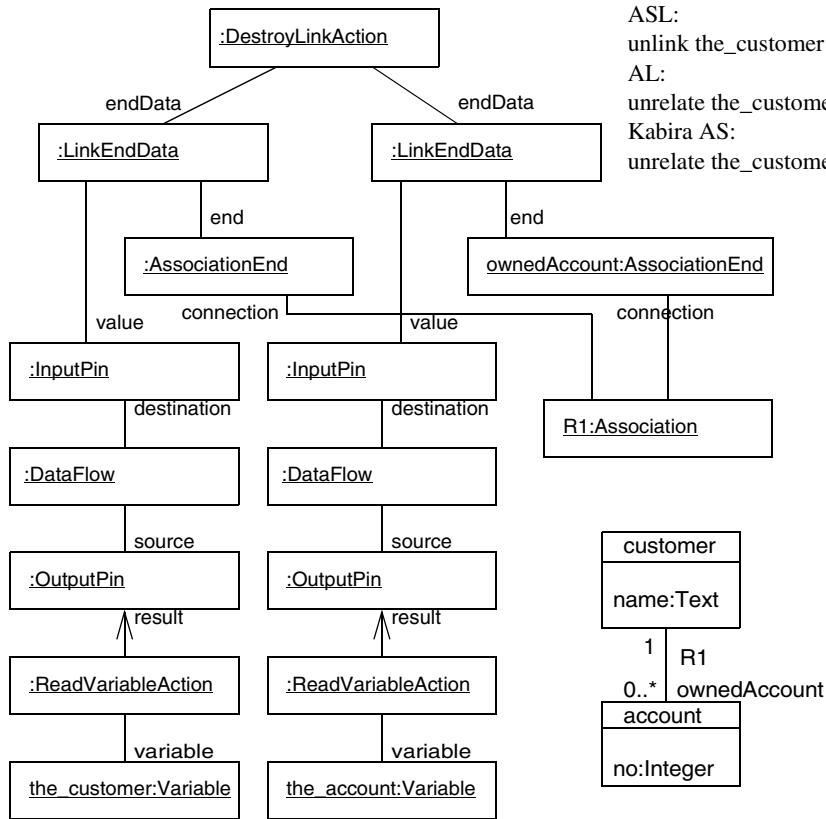


Figure 61. Creating a Link

Destroying a Link

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

All three of the languages make use of Role names where they are required to disambiguate an operation. For example, in a reflexive association it is necessary to know in which direction the link is intended. Role names will clarify this.



ASL:
 unlink the_customer R1 the_account
 AL:
 unrelate the_customer from the_account across R1;
 Kabira AS:
 unrelate the_customer ownedAccount the_account;

Figure 62. Destroying a Link

Navigating an Association to a Single Object

Navigation of an association in these action languages is carried out for the purposes of obtaining the object references of linked objects. The references thus obtained can then be used to perform some manipulation on the linked objects, for example calling an operation provided by the object.

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

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

```

ASL:
owner = the_account -> R1
AL:
select one owner related by
the_account -> customer[R1];
Kabira AS:
owner = the_account -> customer[owner];
    
```

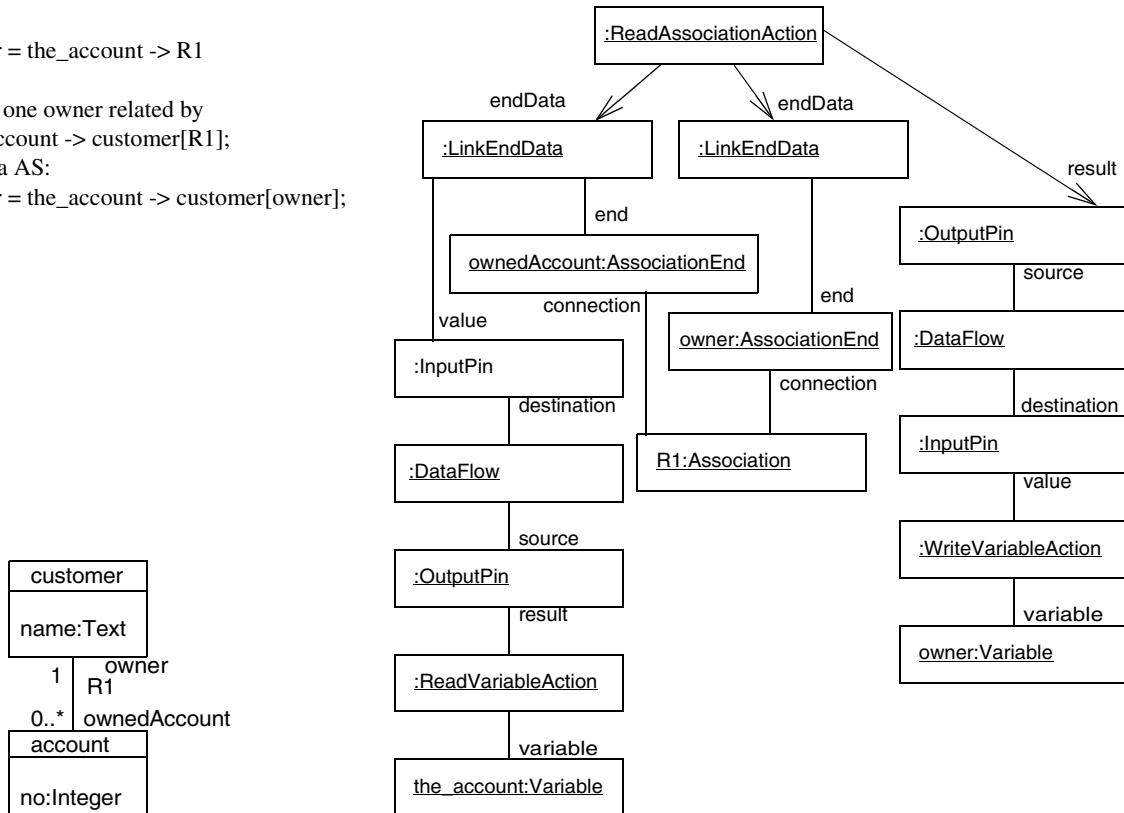


Figure 63. Navigating an Association

Navigating an Association to Multiple Objects

Navigation of an association in these action languages is carried out for the purposes of obtaining the object references of linked objects. The references thus obtained can then be used to perform some manipulation on the linked objects, for example calling an operation provided by the object.

In this example, a navigation is carried out starting from an instance of `customer` (specified by the object reference `the_customer`) in order to obtain references to all the accounts owned by customer. Due to the multiplicity of the association in the example, the result will be a collection.

In ASL and AL, local variables can be collections and so the example and mapping is very similar to those in the previous example (of navigation to a single instance). In Kabira AS however, local variables cannot be collections and to the result of the navigation must be used directly in an explicit loop. This is shown in the second mapping, where use of the loop is to execute some unspecified action for each object resulting from the navigation. On each iteration of the loop the local variable `the_account` references the next object related through the association being navigated. The body of the loop carries a comment indicating that there would be some action (or sequence of actions) operating on that object.

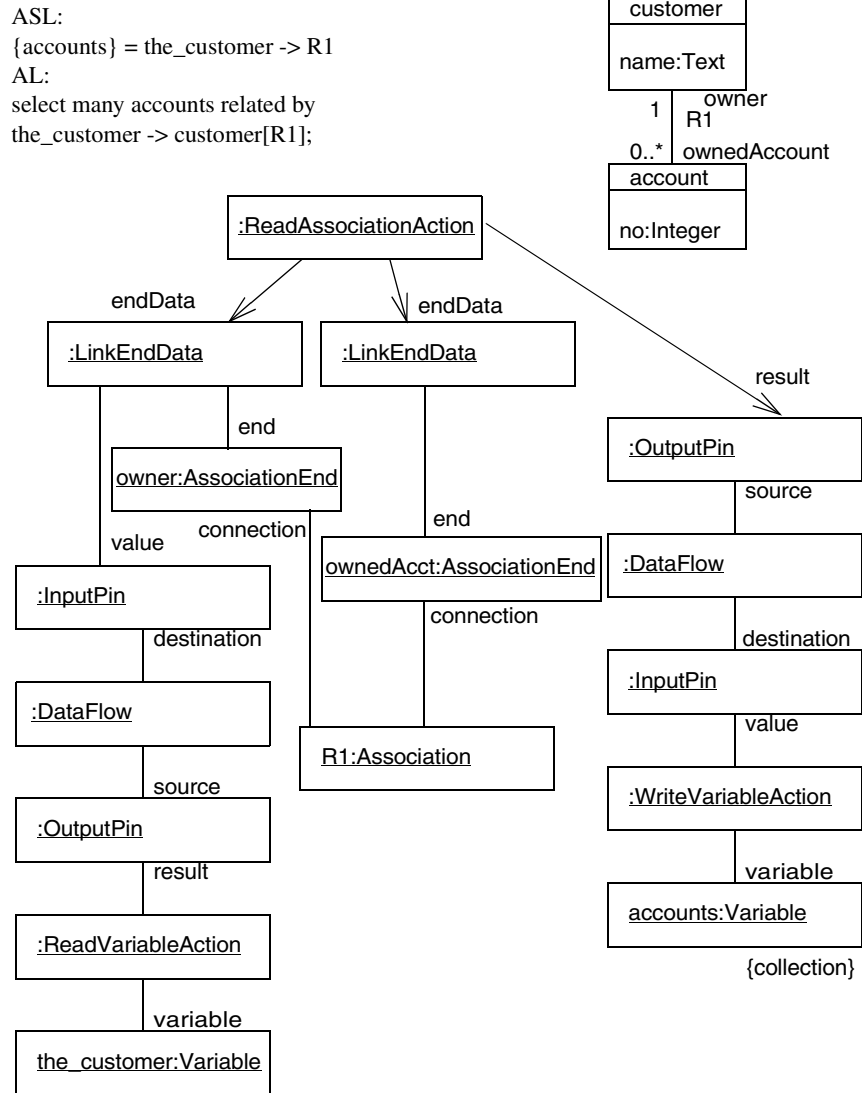


Figure 64. Obtaining a Collection by Navigation

```

Kabira AS:
for the_account in the_customer ->
account[ownedAccount]
{
// some action on the_account
}
    
```

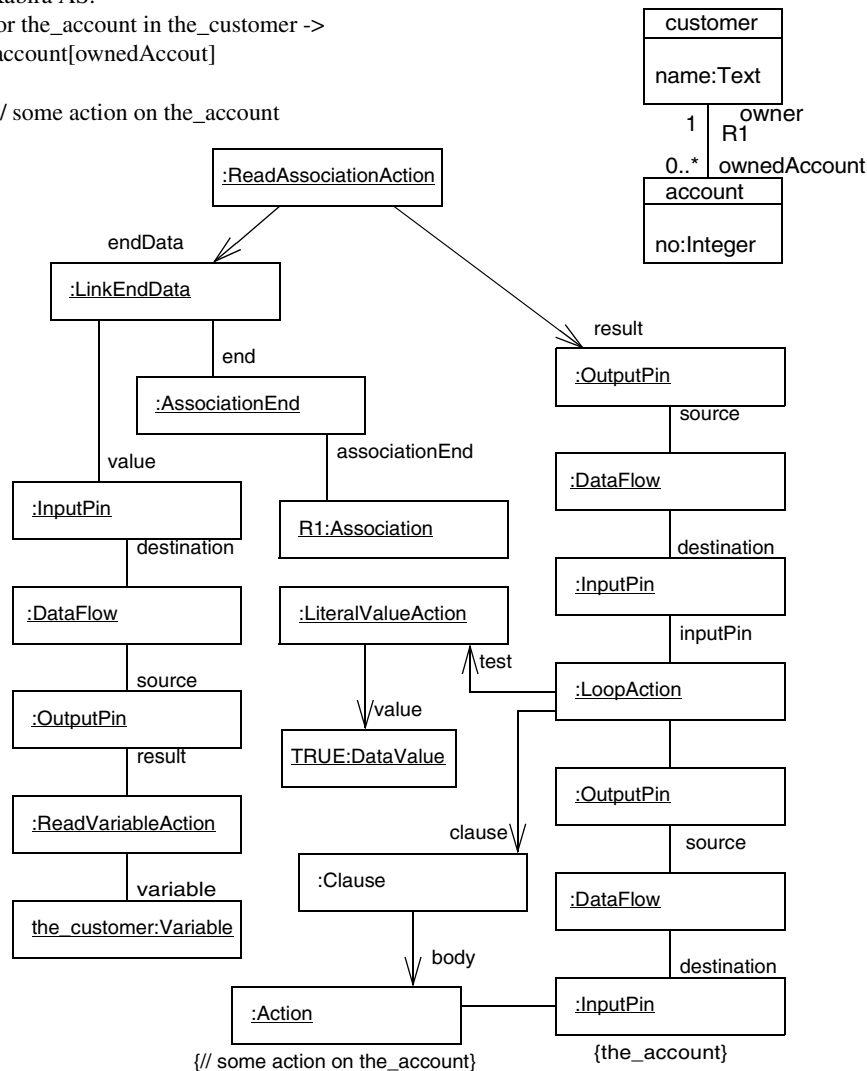


Figure 65. Loop over a Navigation

B.5. Messaging Actions

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

Invocation of an Instance Operation with no Parameters

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

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

ASL:
 []= delete[] on the_customer
 AL:
 the_customer.delete();
 Kabira AS:
 the_customer.delete();

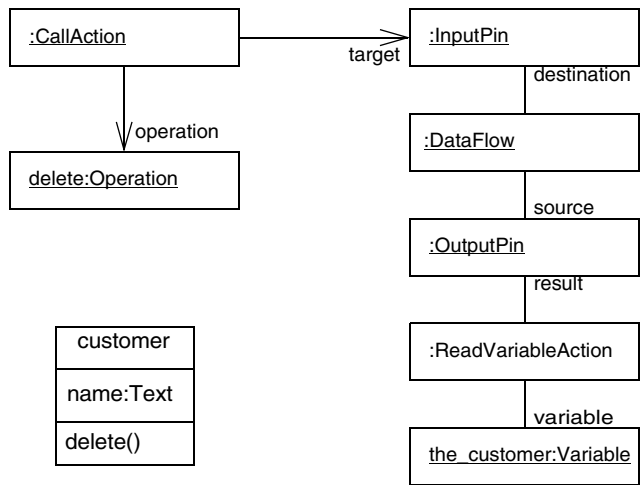


Figure 66. Simple Operation Invocation

ASL:
 [new_balance]= update_balance[amount] on the_account
 AL:
 the_customer.update_balance(amount);
 Kabira AS:
 new_balance = the_account.update_balance(amount);

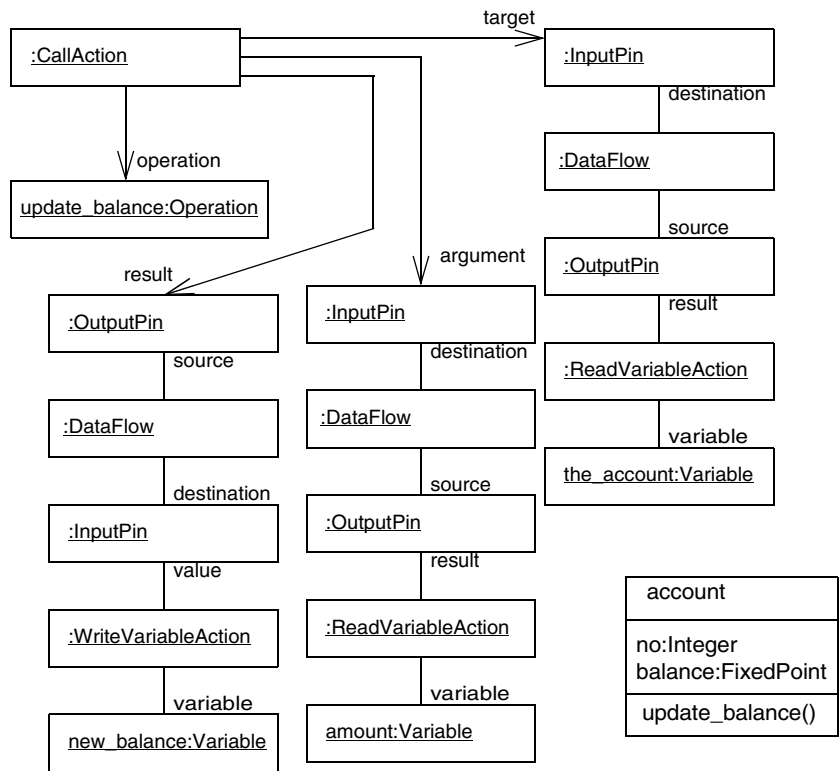


Figure 67. Operation with Parameters

Invocation of an Instance Operation with Parameters

This shows the invocation of an operation that takes an input parameter (amount) and updates the balance of an account. The resulting balance is returned as an output parameter.

Sending of a Signal Event with no Parameters

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

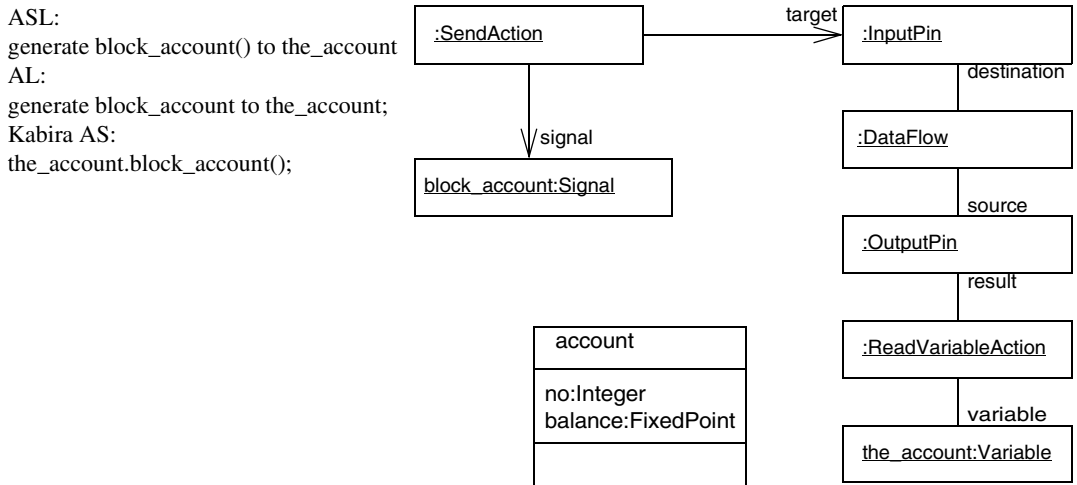


Figure 68. Sending of Signal (no parameter)

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

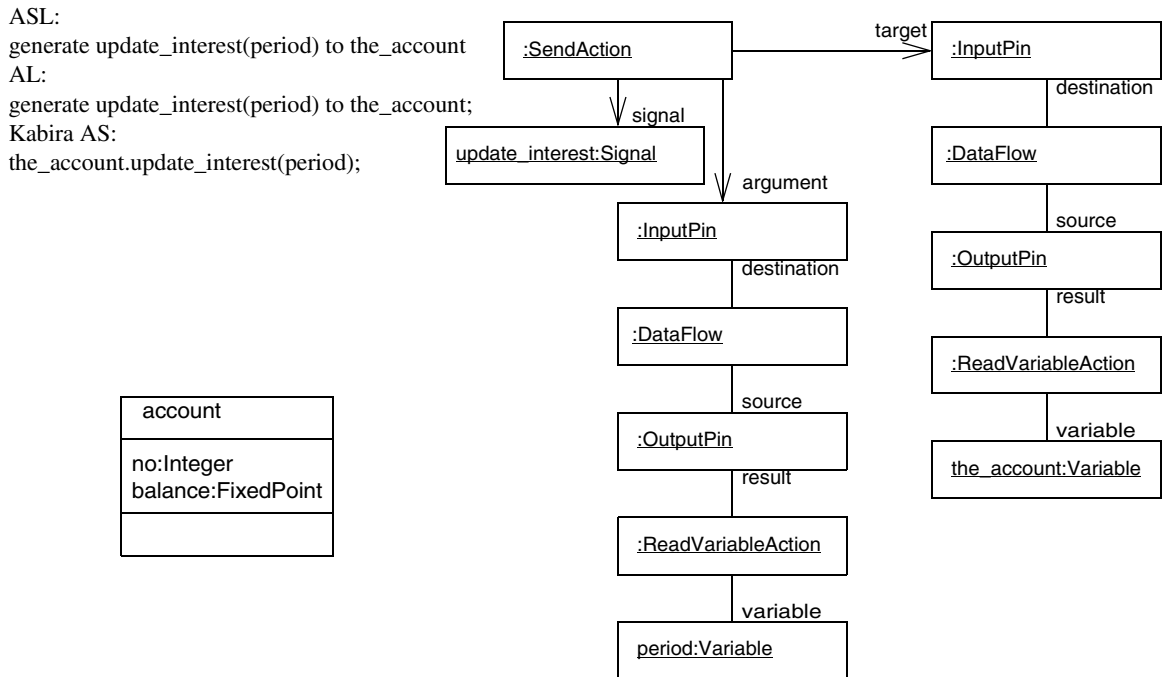


Figure 69. Sending of Signal with Parameters

Sending of a Signal Event with Parameters

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

B.6. A Complete Example

In the following, we show an example of a method associated with an operation, covering the usage of local variables, simple expressions, assignments, object creation, procedure calls, and the sending of signals. As is often the case with examples demonstrating language aspects, they appear somewhat contrived.

The examples depend on a context that is outside the scope of the action semantics, but which nevertheless is impossible to ignore. Here, it is described using the UML style notation of SDL.

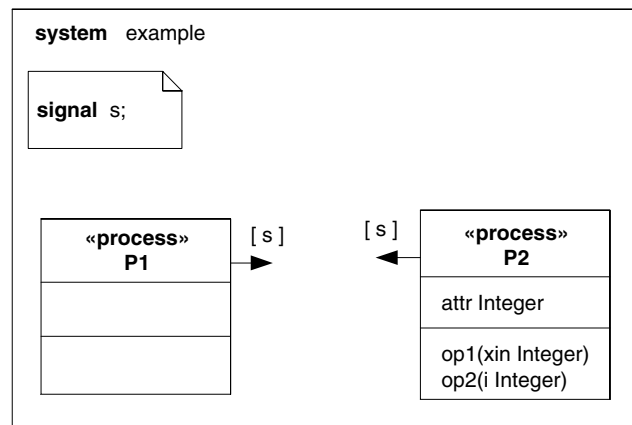


Figure 70. The Context of the Examples

This example relies on two classes that are stereotyped as «process». These represent a special kind of active class that in SDL is referred to as process agent, or simply agent. Agents encapsulate not only their behavior (provided through a finite state machine), but also their internal structure (which may be composed of other agents connected through communication channels, and may be further decomposed in a hierarchical manner). Agents provide a scope for contained entities such as variables, data types, procedures, signals, and timers.

Further, agents support interfaces which allow more flexibility than their UML namesakes. As shown in Figure 70, SDL interfaces are represented by the arrows protruding from the class symbols. While UML interfaces only describe the services that are provided by an active object, in SDL a distinction is made between such provided interfaces, and required interfaces that describe the services that the agent expects from its environment. SDL interfaces are associated with connection points (called gates) that allows the description of bi-directional interactions with an agent, and also act as the only available entry or exit points to the internal structure of the agent. Taken together, these constructs provide an encapsulation boundary for agents that affords a high degree of reusability. In the example, agent P1 offers a provided interface (it is capable of receiving the signal s), while agent P2 displays a required interface (it assumes that another agent is able to receive the signal s). For a more detailed description of the semantics of these elements and their mapping to UML, please refer to the SDL UML profile described in the ITU-T Recommendation Z.109.

We shall focus on the operation `op1` (referred to as a procedure in SDL) defined in agent P2 which we consider specified as shown in Figure 71. We will first break this example into its constituent statements and cover these one by one. Finally, we will assemble the resultant model fragments into a single model.

```

procedure op1 (xin Integer) -> Integer {
  dcl var Integer;
  dcl p P1;
  var := 3;
  call Op2(7);
  var := var + attr;
  p := create P1; /* create instance p of process P1 */
  output s to p; /* send signal s to process instance p */
  if (xin > var) var := 0;
  return var;
}
    
```

Figure 71. Example SDL Specification

Assignment

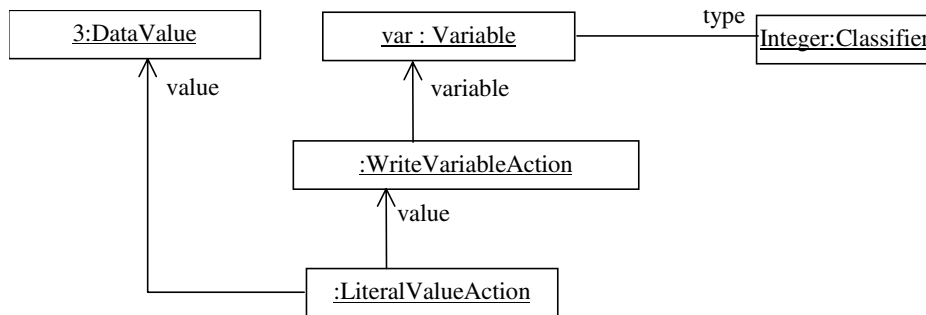


Figure 72. Mapping of Assignment Statement `var := 3;`

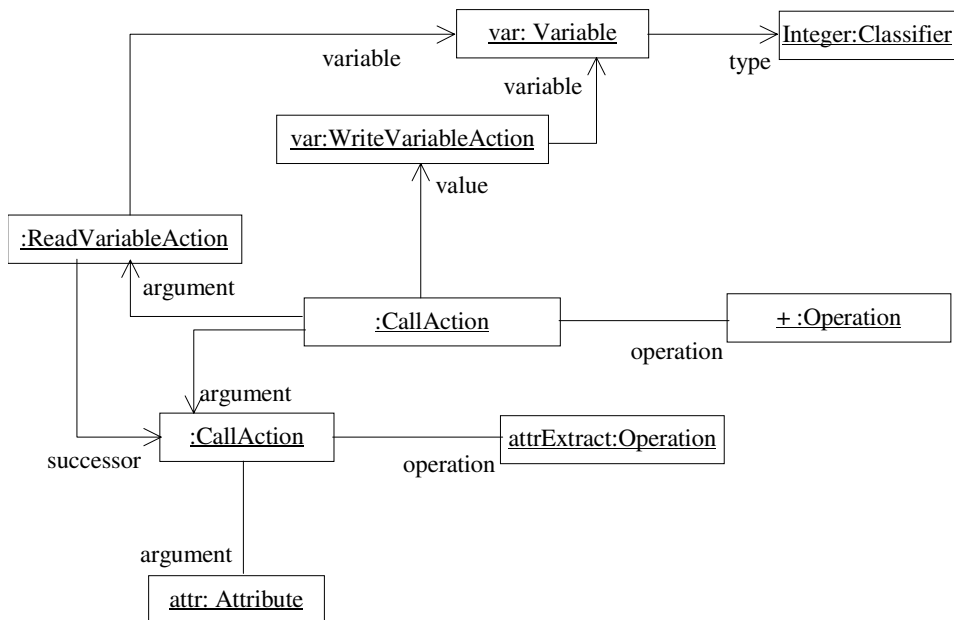


Figure 73. Mapping of Assignment Statement `var := var + attr;`

An assignment to a variable is represented by a `WriteVariableAction`. Variables in SDL can hold two types of data: objects and inlined data objects (which are referred to as values in SDL and subsume the UML `DataValue`). The detailed semantics of an assignment statement depends on the nature of the variable and the expression assigned to the variable.

In the example, the variable holds a value. First, the expression is interpreted, and its result is copied onto the variable by interpreting the `copy` method defined implicitly by the definition that introduced the type of the variable, given expression as the actual parameter. If the expression is `Null`, the predefined exception `InvalidReference` is raised. As described above, the UML model expressing this semantics can be further simplified: The expression is known to be a literal data value (the integer 3), and the test for the expression having no value can be omitted also.

In situations when the variable holds an object, or when an assignment attempt is performed (where the type correctness of the assignment has to be determined at run-time due to polymorphic narrowing), the resultant UML model would differ.

Procedure Call

The interpretation of a call statement beings by interpreting the actual parameter expressions in the order given and then transfers the interpretation to the operation referenced, and the method associated with this operation is interpreted. The interpretation of the action containing a call statement continues when the interpretation of the called operation is finished and the result of the called operation is returned.

For a dynamically dispatched operation, the SDL UML profile specializes the mechanism by which the associated method to be interpreted is determined (see Chapter 11.5, “Execution Model Classes”).

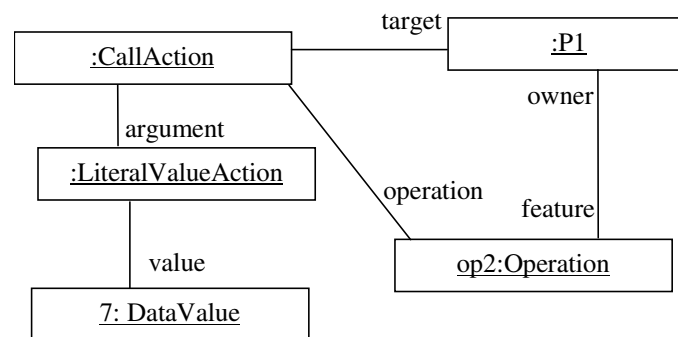


Figure 74. Mapping of Call Statement `call Op2(7);`

Create request

The interpretation of a create request statement causes the creation of an agent instance either inside the agent that performs the create or in the agent that contains the agent that performs the create. The `parent` attribute of the created agent has the value returned by the `self` attribute of the creating agent. The `self` attribute of the created agent and the `offspring` attribute of the creating agent receive the identify of the created agent. When an agent instance is created, the actual parameter expressions are interpreted in the order given, and assigned to the corresponding formal parameters. If an attempt is made to create more agent instances than specified by the maximum number of instances in the agent definition, then no new instance is created, the `offspring` attribute of the creating agent has the result `Null`, and interpretation continues.

As there are no upper bounds given on the number of instances of the agent `P1` that can be created, the latter test can again be compiled away and is omitted.

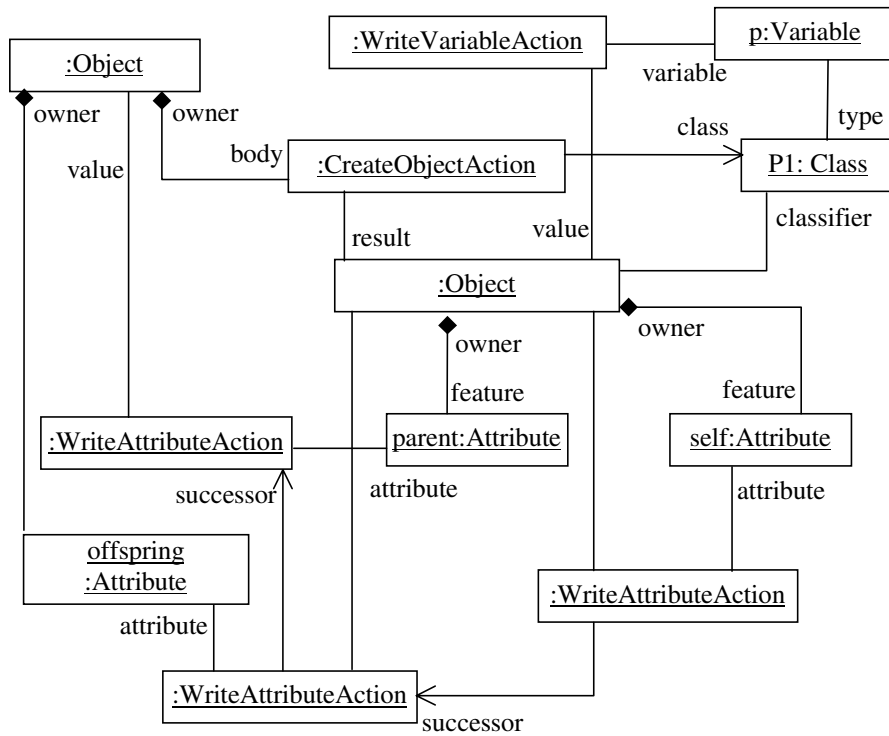


Figure 75. Mapping of Create Statement `p := create P1;`

Output

The receiver (i.e., the target object) of an output statement is determined by interpreting the destination expression and then a compatibility check for the dynamic type of the destination expression is performed for the signal instance. If the destination expression evaluates to Null, the signal instance will be discarded.

If both the sender and the receiver are contained in the same process instance, then the data items conveyed by the signal instance are the results of the actual parameters of the signal in the output statement. If the sender and the receiver are contained in different process instances, then the data values conveyed by the signal instance are newly created replicates of the results of the actual parameters of the output and share no references with the results of the actual parameters of the output. When there are cycles of references in the result of the actual parameters, the conveyed data items will also contain these cycles. Each conveyed data item will be equal to the corresponding actual parameter of the output. The identity of the originating agent is conveyed by the signal instance in the `sender` attribute.

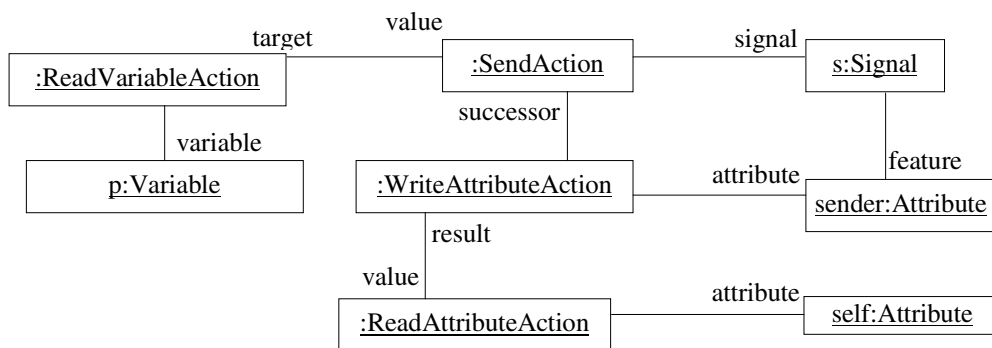


Figure 76. Mapping of Output Statement `output s to p;`

Partial evaluation again helped simplify the resultant UML model, as there is no data communicated with the signal, and we will be assured that the receiver process exists (it had been created in the preceding statement).

Conditional

In a conditional statement, first the condition expression is interpreted. If it evaluates to the Boolean value true, the interpretation is transferred to the consequent statement. Otherwise, interpretation is transferred to the alternative statement. If there is no alternative statement, interpretation continues following the conditional statement.

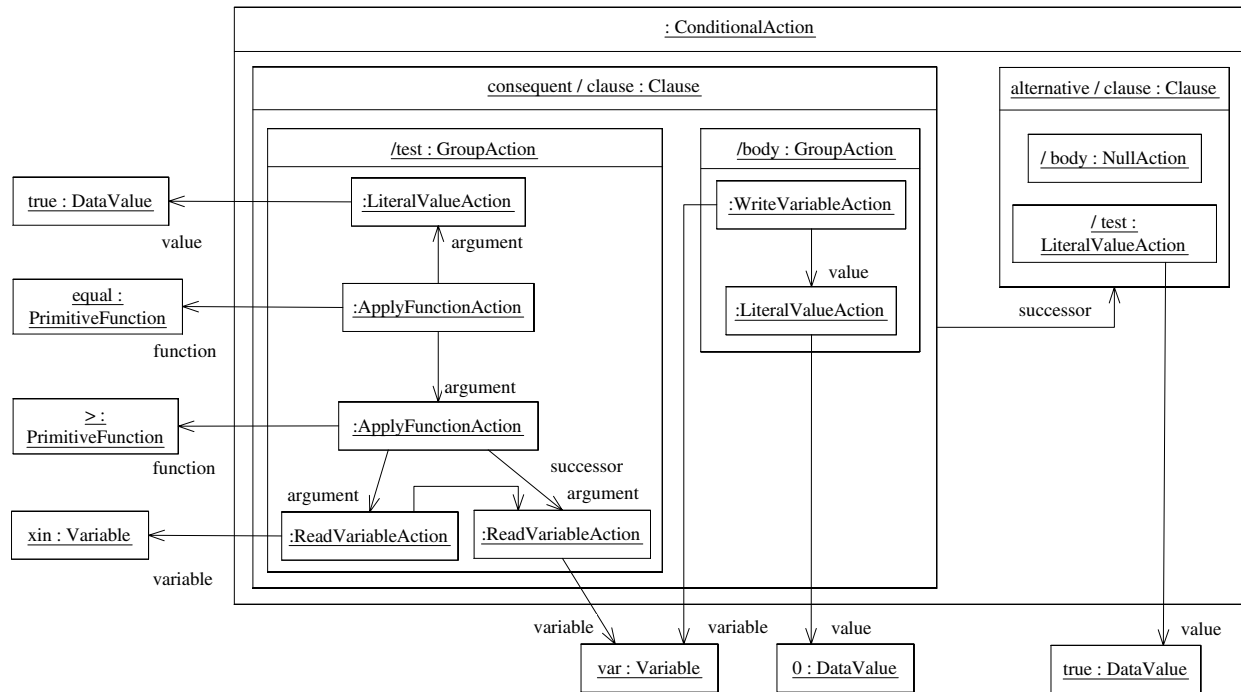


Figure 77. Mapping of Conditional Statement `if (xin > var) var := 0;`

Putting It All Together

A instance of an SDL procedure is interpreted in the following way:

A local variable is created for each in-parameter, having the type of the in-parameter. The variable is associated with the result of the expression given by the corresponding actual parameter, if present, by interpreting an assignment between the variable and the expression. A local variable is created for each out-parameter, having the type of the out-parameter. The variable is not initialized. A local variable is created for each local variable declared by the SDL procedure. Each inout-parameter denotes a variable defined in an outer scope that is given by the actual parameter expression.

The method derived from the body of the SDL procedure is interpreted. Before interpretation of a return statement, the out-parameters are given the values of the corresponding local variable.

Interpretation of the return statement causes all variables created by the interpretation of the SDL procedure to cease to exist, and the result of the return expression, if any, is returned to the calling context.

In the resultant UML model, an SDL procedure is represented as an operation; the body of the SDL procedure is represented as a Procedure that forms the body of a Method associated to the operation, as shown in Figure 78. We show only action detail specific to the definition of that Procedure. The mappings for the individual statements forming the body of the SDL procedure have already been given in the preceding object diagrams and are represented as subsystem symbols bearing the source text of each particular statement as a label. (These symbols are only used for convenience; in the full object diagram, the actual mappings given in the preceding diagrams would be inserted in place of these subsystem symbols. Note that links to these subsystem symbols are actually links to instances of actions in the mapping for the respective SDL statements.)

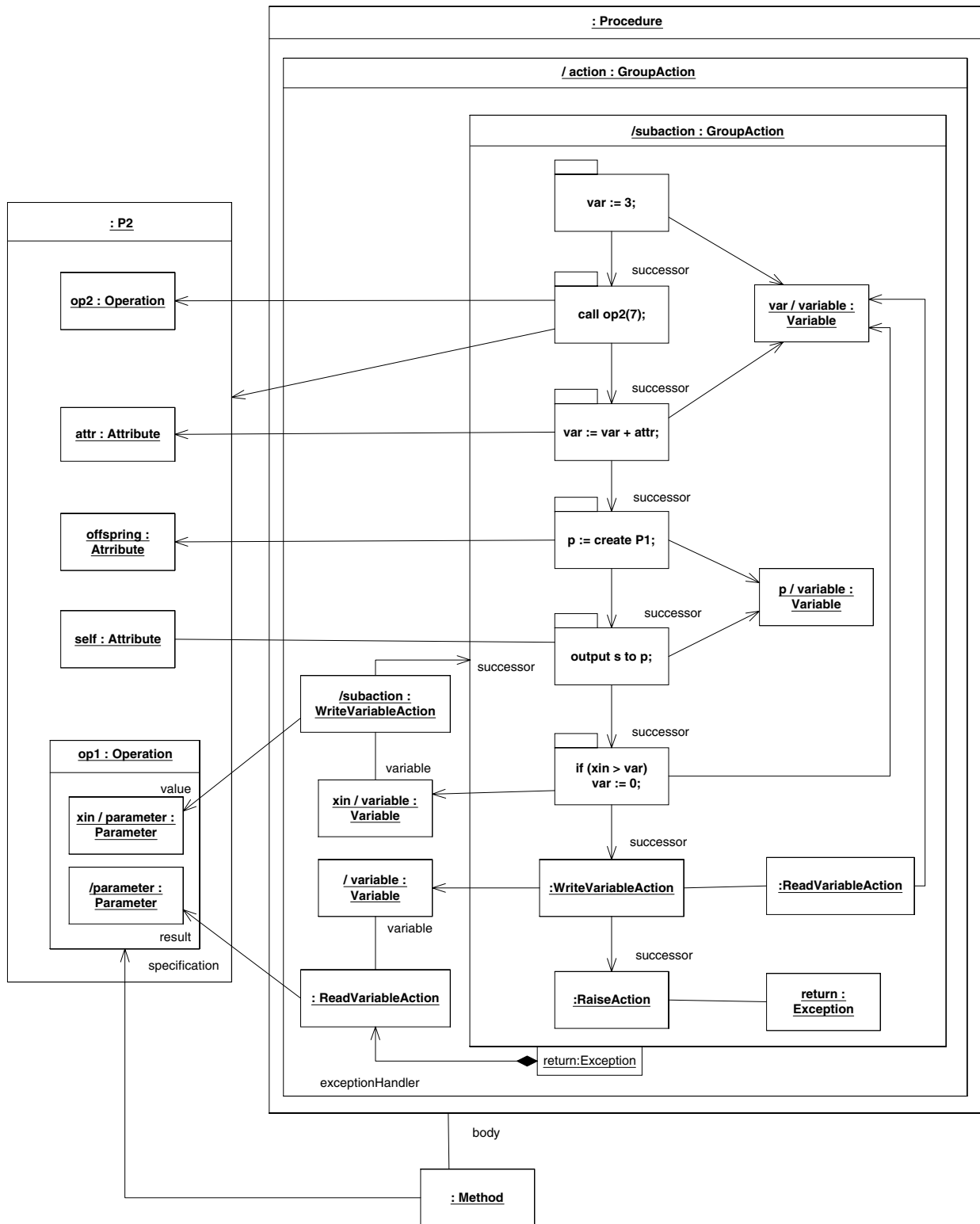


Figure 78. Mapping of Procedure `op2`;

General Index

A

- action 3
 - execution of 55
 - specifying 53
- action foundation 10
- association
 - navigation 92
- association manipulation actions 91
- attribute manipulation actions 90
- attribute writing 167, 168

C

- call 141
- changes to UML 153
- chapter structure 15
- class
 - description format 16
- clause 68, 69
 - execution 71, 72
- collection 127
- collection action 11
- collection actions 127
 - general rules 127
- complete example 179
- composite action 55, 67
- computation action 11
- computation actions 121
- conditional action 68
 - execution 72
- congruent collection 127
- context 55
- control flow 10, 54
- conventions 15
- correspond
 - constraint 128
- creating a link 172

D

- data flow 10, 54
- data value 31
- destroying a link 172
- document structure 15

E

- example
 - complete 179
- exception 12
- execution engine 6
- execution instance model 31
- execution model 8
- execution semantics
 - specification of 23
- extent 33
- extent of classifier
 - reading 95

F

- FAQs 12

G

- group action 10, 67, 68
 - execution 71

H

- host object navigation 95

I

- If-then-else logic 163
- input pin 54
- input to action
 - specification of 23
- invitation 142

L

- lifecycle
 - action 56
 - specification of 23
- link 32
 - creation 172
 - destroying 172
 - identifying 91
 - writing 94

General Index

link object 33
lnk
 reading 93
local variable 69
loop action 69
 execution 73
loop variable 69

M

mapping of languages 6
messaging actions 141
messaging examples 176
multi-way decision 164

N

navigating an association 174
navigation
 across association 92

O

object 31
Object Constraint Language 18
object creation
 with attribute assignment 166
object destruction 166
object manipulation actions 89
occurrence 46
OCL 18
output from action
 specification of 23
output pin 54
overview 3

P

pin 54
primitive action 54
primitive actions 5
procedural context 55
procedure
 execution 57
procedure call 141

R

read action 11
read actions 89

read link actions 93
run to completion 45

S

scope of document 3
selecting a subset of objects 169
semantics
 defining 6
send
 signal 141
 synchronous 142
sending a signal 178
shape 127
signal sending 141
simple object creation 165
slice 127
snapshot 8, 21
state machine
 execution 9
state machine execution 43
subaction 127
synchronous signal 142

T

time
 model of 8
transition 45

U

UML
 changes to 153
 relationship to 5
undefined semantics 4, 153
unordered execution 131

V

variable manipulation actions 95

W

well-formedness rule
 specifying 16
write action 11
write actions 89
write link action 94
writing attributes 167, 168

Index of Classes

A

Action 58
ActionExecutionIdentity 62
ActionExecutionSnapshot 62
ApplyFunctionAction 122
ApplyPrimitiveFunctionExecution 125
AssociationAction 97
AssociationExtentIdentity 36
AssociationExtentSnapshot 36
AttributeAction 98
AttributeValue 35

C

CallAction 142, 144
CallActionExecution 148
Change 36
ChangeOccurrence 47
ClassifierExtentIdentity 37
ClassifierExtentSnapshot 37
Clause 75
ClauseExecutionIdentity 80
ClauseExecutionSnapshot 80
CodeAction 123
CodeExecution 125
ConditionalAction 76
ConditionalExecutionIdentity 81
ConditionalExecutionSnapshot 81
ControlFlow 59
CreateLinkAction 99
CreateLinkObjectAction 100
CreateObjectAction 101

D

DataFlow 60
DataValueIdentity 38
DestroyLinkAction 102
DestroyObjectAction 103

E

ExecutionSnapshot 38

F

FilterAction 128
FilterExecution 137

G

GroupAction 77
GroupExecutionIdentity 83
GroupExecutionSnapshot 83

H

History 39

I

Identity 39
InputPin 60
InstanceIdentity 39
InviteAction 144
InviteActionExecution 149
InvitePacket 145
IterateAction 130
IterateExecution 138

L

LinkEndCreationData 104
LinkEndData 105
LinkEndValue 39
LinkIdentity 40
LinkObjectIdentity 40
LinkObjectSnapshot 40
LinkSnapshot 41
LiteralValueAction 124
LiteralValueExecution 125
LoopAction 78
LoopExecutionSnapshot 84

M

MapAction 132
MapExecution 138

Index of Classes

N

NullActionExecution 125

O

ObjectIdentity 41
ObjectSnapshot 41, 48
Occurrence 48
OutputPin 61

P

Pin 61
PinValue 64
PrimitiveAction 61
PrimitiveActionExecutionIdentity 65
Procedure 62
ProcedureExecutionIdentity 65
ProcedureExecutionSnapshot 65

Q

QualifierValue 105

R

ReadAssociationAction 105
ReadAttributeAction 107
ReadExtentAction 108
ReadIsClassifiedObjectAction 108
ReadLinkObjectEndAction 110
ReadLinkObjectQualifierAction 111
ReadSelfAction 112
ReadVariableAction 112
ReclassifyObjectAction 113
ReduceAction 134
ReduceExecution 137, 139

S

SendAction 145
SendActionExecution 146, 150
SendPacket 146
Signal 146
SignalOccurrence 48
Snapshot 42
StateMachineExecutionIdentity 48
StateMachineExecutionSnapshot 49
Step 42

T

TimeOccurrence 49

V

Variable 79
VariableAction 114
VariableExecutionIdentity 86
VariableExecutionSnapshot 86

W

WriteAttributeAction 115
WriteLinkAction 117
WriteVariableAction 118

Index of Features

A

accessor 42
action 60, 61, 62
actionExecution 64, 65
antecedent 58
argument 48, 62, 122, 124, 130, 134, 136, 142, 144, 146
association 36, 40
associationEnd 39
attribute 35, 98, 107, 115
attributeValue 37, 41
availableInput 58
availableOutput 58

B

body 75
bodyExecution 81, 84

C

cause 37
change 39
class 102
classifier 37, 41, 108, 109
clause 76, 78, 80
clauseExecution 81
collectionInput 132
conditionalExecution 80
conditionalExecutionID 81
configuration 49
consequent 58
context 62
current 49

D

destination 60

E

effect 42
encoding 124
end 105, 110

endData 97, 99, 101, 103, 105, 117
endData.insertAt 99, 101
endData.qualifier.value 99, 101, 103, 106
endData.value 99, 101, 103, 106
event 48
executionID 63, 65, 80, 86
executionId 49
extentId 36, 37

F

flow 60, 61
function 122

G

groupExecutionID 83

H

history 37
host 48, 65

I

identity 42
input 104, 109, 113
inputPin 58
insertAt 104, 115, 118
instance 37
isDeterminate 76
isDirect 109
isRemove 115, 118
isUnordered 131, 135

L

language 124
leftSubinput 136
link 36, 39
linkEnd 37, 41
linkId 41
loop 61
loopVariable 132

Index of Features

loopVariableInput 132

M

machine 48
multiplicity 61, 79

N

newClassifier 113

O

object 98, 107, 110, 111, 115
objectId 41
occurrence 48
oldClassifiers 113
operation 142
ordering 61, 79
output 75
outputPin 58

P

pin 64
pinValue 63, 65
postChange 42
preChange 42
predecessor 35, 37, 39, 59, 75
proceduralContext 62
procedure 60, 61, 65
procedureExecution 64

Q

qualifier 105, 111
qualifierValue 39

R

referent 38
result 62, 101, 102, 105, 106, 107, 108, 109, 110,
111, 112, 113, 122, 124, 130, 132, 134, 136,
142, 144
rightSubinput 136

S

scope 79

selectedClause 81
signal 144, 146
signalInstance 48, 145, 146
source 60
status 62, 65, 80, 145, 146
subaction 77, 129, 131, 134, 136
subactionExecution 83
subinput 129, 132, 134
suboutput 132, 134, 136
substatus 81, 84
subtest 130
successor 35, 37, 39, 59, 75

T

target 142, 144, 145, 146
test 75
testExecution 80, 84
testOutput 75
time 37
type 38, 61, 79

V

value 35, 39, 64, 86, 105, 115, 118, 124
variable 77, 86, 112, 114, 118
variableExecution 83

Index of OCL Operations

Symbols

<= 19, 96

A

actionExecutionStarted 66
allLinkEndObjects 97
allNestedActions 58
allows 19, 60
allParents 60
allPins 59
allSubactions 59
allSuccessors 59, 75
Association 97
at 35, 40
attribute 146
attributeValues 98
availableExecution 65
availableExecutions 63, 83, 84

B

bodyExecutionCompleted 82, 85
bodyExecutionStarted 82, 85

C

clauseExecutionsStarted 82
clauseOutput 82, 85
clausesExecuted 82
clauseSnapshots 82
compatibleWith 19, 60
completed 64
contains 19, 60
current 38
currentActionSnapshot 66
currentSnapshotOf 63
currentValueOfVariable 85

D

destroyed 38

E

executionOf 63
existingLink 117

G

getVariableValues 114

H

host 63
hostClassifier 96
hostElement 97

I

in_parameter 142
inputValuesUnchanged 64
is 19
isAccessibleBy 79
isAttributeValueExist 99
isBeingInitialized 118
isCompatibleWith 144
isMatchingLinkIdentity 117
isNew 42
isSubaction 59
isVariableValueExist 115

L

loopOutputsSet 85
loopVariablesInitialized 85
loopVariablesUnchanged 85
loopVariablesUpdated 85
lowerbound 19

M

method-lookup 148

N

nestedActions 58, 77

Index of OCL Operations

nestedExecutions 81
new 38

O

out_parameter 142

P

peerExecutions 63
pinValueAt 64
post 38
pre 38
preClauseSnapshots 82
predecessor 42
prerequisites 59, 80
preValueOfVariable 85
procedure 96
procedureExecution 62

S

sourceValueFor 63, 66
subactionExecutionsCompleted 84
subactionExecutionsStarted 83
subactions 59, 61, 76, 77, 78
successor 42
successors 59

T

terminates 42
testExecutionCompleted 85
testExecutionStarted 80, 85
testOutputValue 80

U

upperbound 19

V

valueOf 63, 66
variableExecutionsStarted 84
variableExecutionStopped 84