Date: 17 August 2009

Concrete Syntax for a UML Action Language v0.06

ad/2009-08-02

Submitted by International Business Machines

Supported by Ericsson AB

USE OF SPECIFICATION - TERMS, CONDITIONS & NOTICES

The material in this document details an Object Management Group specification in accordance with the terms, conditions and notices set forth below. This document does not represent a commitment to implement any portion of this specification in any company's products. The information contained in this document is subject to change without notice.

LICENSES

The companies listed above have granted to the Object Management Group, Inc. (OMG) a nonexclusive, royalty-free, paid up, worldwide license to copy and distribute this document and to modify this document and distribute copies of the modified version. Each of the copyright holders listed above has agreed that no person shall be deemed to have infringed the copyright in the included material of any such copyright holder by reason of having used the specification set forth herein or having conformed any computer software to the specification.

Subject to all of the terms and conditions below, the owners of the copyright in this specification hereby grant you a fully-paid up, non-exclusive, nontransferable, perpetual, worldwide license (without the right to sublicense), to use this specification to create and distribute software and special purpose specifications that are based upon this specification, and to use, copy, and distribute this specification as provided under the Copyright Act; provided that: (1) both the copyright notice identified above and this permission notice appear on any copies of this specification; (2) the use of the specifications is for informational purposes and will not be copied or posted on any network computer or broadcast in any media and will not be otherwise resold or transferred for commercial purposes; and (3) no modifications are made to this specification. This limited permission automatically terminates without notice if you breach any of these terms or conditions. Upon termination, you will destroy immediately any copies of the specifications in your possession or control.

PATENTS

The attention of adopters is directed to the possibility that compliance with or adoption of OMG specifications may require use of an invention covered by patent rights. OMG shall not be responsible for identifying patents for which a license may be required by any OMG specification, or for conducting legal inquiries into the legal validity or scope of those patents that are brought to its attention. OMG specifications are prospective and advisory only. Prospective users are responsible for protecting themselves against liability for infringement of patents.

GENERAL USE RESTRICTIONS

Any unauthorized use of this specification may violate copyright laws, trademark laws, and communications regulations and statutes. This document contains information which is protected by copyright. All Rights Reserved. No part of this work covered by copyright herein may be reproduced or used in any form or by any means--graphic, electronic, or mechanical, including photocopying, recording, taping, or information storage and retrieval systems--without permission of the copyright owner.

DISCLAIMER OF WARRANTY

WHILE THIS PUBLICATION IS BELIEVED TO BE ACCURATE, IT IS PROVIDED "AS IS" AND MAY CONTAIN ERRORS OR MISPRINTS. THE OBJECT MANAGEMENT GROUP AND THE COMPANIES LISTED ABOVE MAKE NO WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, WITH REGARD TO THIS PUBLICATION, INCLUDING BUT NOT LIMITED TO ANY WARRANTY OF TITLE OR OWNERSHIP, IMPLIED WARRANTY OF MERCHANTABILITY OR WARRANTY OF FITNESS FOR A PARTICULAR PURPOSE OR USE. IN NO EVENT SHALL THE OBJECT MANAGEMENT GROUP OR ANY OF THE COMPANIES LISTED ABOVE BE LIABLE FOR ERRORS CONTAINED HEREIN OR FOR DIRECT, INDIRECT, INCIDENTAL, SPECIAL, CONSEQUENTIAL, RELIANCE OR COVER DAMAGES, INCLUDING LOSS OF PROFITS, REVENUE, DATA OR USE, INCURRED BY ANY USER OR ANY THIRD PARTY IN CONNECTION WITH THE FURNISHING, PERFORMANCE, OR USE OF THIS MATERIAL, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.

The entire risk as to the quality and performance of software developed using this specification is borne by you. This disclaimer of warranty constitutes an essential part of the license granted to you to use this specification.

RESTRICTED RIGHTS LEGEND

TRADEMARKS

COMPLIANCE

The copyright holders listed above acknowledge that the Object Management Group (acting itself or through its designees) is and shall at all times be the sole entity that may authorize developers, suppliers and sellers of computer software to use certification marks, trademarks or other special designations to indicate compliance with these materials.

Software developed under the terms of this license may claim compliance or conformance with this specification if and only if the software compliance is of a nature fully matching the applicable compliance points as stated in the specification. Software developed only partially matching the applicable compliance points may claim only that the software was based on this specification, but may not claim compliance or conformance with this specification. In the event that testing suites are implemented or approved by Object Management Group, Inc., software developed using this specification may claim compliance or conformance with the specification only if the software satisfactorily completes the testing suites.

ISSUE REPORTING

All OMG specifications are subject to continuous review and improvement. As part of this process we encourage readers to report any ambiguities, inconsistencies, or inaccuracies they may find by completing the Issue Reporting Form listed on the main web page http://www.omg.org, under Documents & Specifications, Report a Bug/Issue.

1 Introduction

This specification is a response to the OMG RFP for a *Concrete Syntax for a UML™ Action Language* (ad/2008-09-09) [3]. It defines a textual language for defining UML behavior that can be mapped to the subset of UML actions defined in the *Foundation Subset for Executable UML Models* (fUML) specification (Beta version, ptc/2008-11-03) [1]. The language is referred to herein as UAL (UML Action Language.)

Resolution of RFP Requirements

1.1.1 Mandatory Requirements

RFP Requirement	Resolution
6.5.1 Proposals shall define a concrete textual syntax for the action language, so that modelers can use text in executable UML models.	UAL borrows heavily from Sun Java™ 1.5 syntax, as documented in [4]
6.5.2 Proposals shall define a computationally complete language.	UAL, when used to augment a visual UML model, requires no further elaboration in a 3GL or other target language. Models specified with diagrams augmented by UAL can be said to be computationally complete by that definition.
6.5.3 Proposals shall reuse existing OMG language specifications where possible, so that consistency is maintained within OMG standards.	UAL is driven semantically by conformance with [1] and [2]
6.5.4 Proposals shall provide a mapping from statements in the concrete syntax to the foundational subset of actions in the Executable UML Foundation, so that the concrete syntax has a defined semantics within UML. It is not necessary to employ every construct in Foundational UML.	Full mapping to the UML Foundation is provided for each statement type and UAL action. These mappings would be embedded within a larger target output context as determined by the containing object and / or diagram.
6.5.5 Proposals shall provide mechanisms for input/output that map to the corresponding input/output constructs specified in the Executable UML Foundation, so that modelers can at least write “Hello world.”	UAL presumes the use of an implementation of the UML Foundational Model Library.
6.5.6 Proposals shall provide mechanisms for interfacing to user-specified input/output packages, so that modelers may provide their own input/output packages.	UAL presumes the use of tooling and target-specific service libraries.

RFP Requirement	Resolution
6.5.7 Proposals shall include standard arithmetic and logical capabilities (directly through the syntax or by a combination of syntax and libraries), so that modelers may do arithmetic and perform tests. These capabilities must cover at least those defined in Clause 9.2 of the Executable UML Foundation.	UAL has a complete set of primitives and operators for arithmetic and logical operations. Extensions have been proposed to the foundational UML [2] in order to complete the set defined in that document.
6.5.8 Proposals shall include mechanisms to invoke user-specified operations, so that modelers may call new or legacy code.	UAL presumes the use of service libraries as part of the tooling and target environments. Invocation actions can be used to invoke any of these extensions to the basic environment.
6.5.9 Proposals shall include a notation for comments, so that modelers may communicate models to others.	UAL has two notations for comments, end-of-line and block. A special format for documentation and translator directives is also defined.
6.5.10 The language shall have a mechanism for transferring control to non-UML "programs," whether hardware or software, so that models may interact with non-UML code and hardware.	Control is transferred to legacy code through method calls on external interfaces supported by service libraries. These methods may wrap asynchronous or synchronous operations, the latter blocking the current thread of execution until the operation returns control. Modelers must take this potential into account and use parallel or background execution (a.k.a. threading) when appropriate.
6.5.11 The language shall allow inline embedding of target programming code, so that modelers may shoot themselves in the foot.	For embedding of languages other than UAL, a non-reformatting block comment contains the target code block. The use of embedded target code is tooling and target specific.
6.5.12 The language shall define a label so that modelers not fully committed to an action language may embed action language fragments in OpaqueExpressions.	The language label for UAL is "UAL"

1.1.2 Optional Requirements

RFP Requirement	Resolution
6.6.1 Proposals may identify extension points of the language, so that modelers may extend the language, consistent with the Executable UML Foundation.	The language is translated to either the foundational UML syntax or to target platform language syntax. A translator can be extended to react to an extension to the syntax that is implemented as a service library such that the translation is fUML compliant.
6.6.2 Proposals may define class libraries, such as collection libraries, I/O packages, and the like, to support their language.	This specification proposes additions to the fUML library, and a collection class library (tbd.)

1.2 Issues to Be Discussed

Issue	Resolution
Proposals shall discuss the relationship of the concrete syntax to existing OMG language specifications, particularly OCL.	This specification specifically proposes a layer zero syntax that has a strong C and Java legacy, but is based entirely on the foundational UML and UML Superstructure semantics. This layer is a pragmatic subset in a very familiar notation and is designed to be highly appealing to significant portions of all markets and industries. This specification further recognizes the need for extended syntax in a layer 1 to implement data handling and textual model definition. The data handling portions of the language will use an OCL-like syntax, as it is most appropriate for those tasks. This specification presumes the incorporation of the relevant sections of the Action Language for the Foundational UML (ALF) for layer 1.
Proposals shall discuss how they resolved the tension between the perceived marketability of imperative languages and both OCL and the data-flow nature of the action model.	The incorporation of both in a two-layer specification is our chosen approach. We concern ourselves only with layer zero in this specification, as there is a thorough and detailed description of layer 1 in the Action Language for the Foundational UML (ALF.)
Proposals shall discuss the relationship to formal specification languages and how an action language “program” may be proven	A mapping is provided to the foundational UML. Any model that can be translated to fUML is deemed compliant and can be proven to the same extent as any fUML model can be proven.

Issue	Resolution
Proposals shall discuss the ease of use and understandability from the point of view of the modeler.	The core UAL language syntax is in widespread use in all industries. A huge pool of Java programmers is already fluent with the syntax. The shift to UML semantics is quite natural in that set semantics are used to model structural features and links.
Proposals shall discuss the parsability of the language (e.g. LL1, LALR).	This surface syntax is based upon Java 5 syntax and can be parsed with the same ease as can Java.

2 Scope

The UML Action Language (UAL) is a textual surface language for the computational completion of UML models. The use of UAL is intended to allow a completed model to be directly executed in a simulator or translated for run-time engines with any architecture – for example both Von Neumann run-times and non-Von Neumann run-times are supported.

Key guiding principles and drivers in the design of UAL include the following:

- Translatability (mapping) to as much of fUML [1] as is practical;
- Translatability to common target run-time languages;
- Express in UAL **only** those parts of the model that are needed to complete the behavioral specification of typical systems – i.e. achieve computational completion;
- Layered conformance:
    - L0 to have a low barrier to entry for vendors and modelers alike;
    - L1 to provide advanced data flow semantics and textual model definition;
    - This submission is concerned with the definition of L0.
- A “very Java” concrete syntax (credit for the term to Stephen Mellor) for both marketing and pragmatic reasons.
- Express **all** semantics in terms of the UML and more specifically the foundational UML (fUML);
    - Corollary: **no** cases exist where Java semantics impose on the behavior of the language, although the syntax does introduce some minor limitations in expressivity.

The scope of this specification is

- specification of a surface language to enable creation of computationally complete UML models for translation to:
    - the foundational UML subset as defined in [1]; and
    - platform-specific target languages.

- exemplar translations to the foundational UML subset.

Given its fundamental nature, the subset assumes the most general type of system, including physically distributed and concurrent systems with no assumptions about global synchronization.

3 Conformance

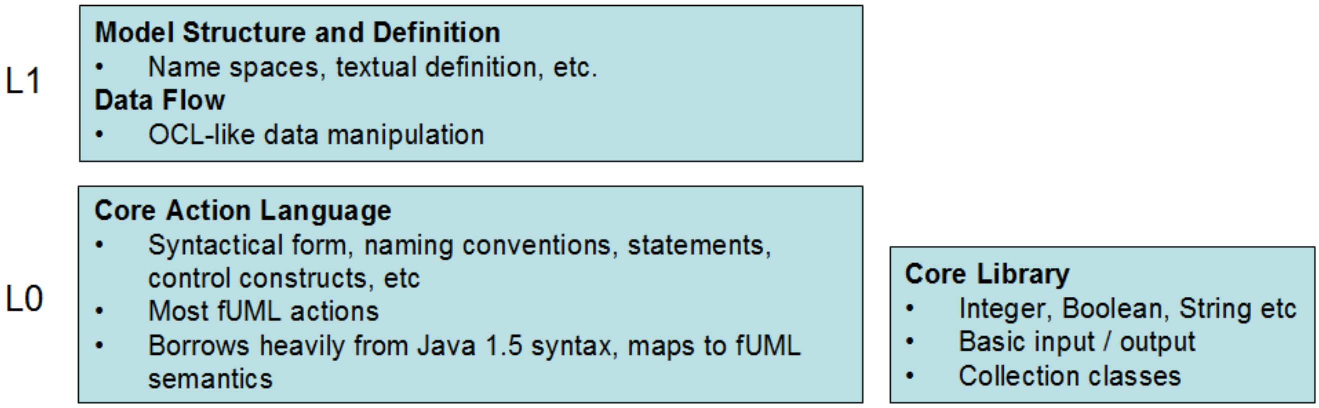
There are two layers of conformance defined for UAL.

The core portion of the language as defined in this specification is designed to permit the computational completion of visual models and resides in the lowest conformance layer (L0.) This specification concentrates on the definition of L0 because it is the minimum level of action language that can be used to achieve the RFP's stated goals.

However, this specification acknowledges the requirement for an L1 for advanced data-flow syntax and textual model definition.

This submission embraces the potential for a standard consisting of basic concrete syntax (UAL) at L0 and advanced data flow and structural syntax (the Action Language for Foundational UML – ALF) at L1.

Figure 1 shows this layering.



L1

Model Structure and Definition
- Name spaces, textual definition, etc.

Data Flow
- OCL-like data manipulation

L0

Core Action Language
- Syntactical form, naming conventions, statements, control constructs, etc
- Most fUML actions
- Borrows heavily from Java 1.5 syntax, maps to fUML semantics

Core Library
- Integer, Boolean, String etc
- Basic input / output
- Collection classes

Figure 1

4 Normative References

[1] *Semantics of a Foundational Subset for Executable UML Models*, Second Revised Submission, 25 August 2008, OMG Specification

[2] *OMG Unified Modeling Language (OMG UML), Superstructure, V2.1.2*, formal/2007-11-02, OMG Specification

[3] *Concrete Syntax for a UML Action Language, Request For Proposal*, OMG Document, ad/2008-09-09

4.1 Informative References

[4] *The Java™ Language Specification, Third Edition*, James Gosling, Bill Joy, Guy Steele, Gilad Bracha, ISBN 0-321-24678-0, First printing, May 2005

[5] *Action Language for the Foundational UML*, version 0.03, Ed Seidewitz, Model Driven-Solutions

5 Terms and Definitions

There are no formal definitions in this specification that are taken from other documents.

6 Symbols

There are no symbols defined in this specification.

7 Additional Information

7.1 Changes to Adopted OMG Specifications

Proposed changes to the Foundational UML Specification [1] included several additional operations for Integers and Strings.

7.2 How to Read this Specification

The rest of this document contains the technical content of this specification. As background for this specification, readers are encouraged to first read [3] and then [1]. Readers are assumed to be familiar with the content of [2] and [4].

7.3 Acknowledgements

The following companies submitted and/or supported parts of this specification:

- International Business Machines
- Ericsson AB

The following individuals within the actiona language submitters group had significant impact and input to this specification and the author would like to acknowledge and thank them for their contributions:

- Stephen Mellor – Leader of the submission team and author of *Executable UML A Foundation for Model-Driven Architecture,* the first reference I acquired when taking on this task.
- Ed Seidewitz – Author of the Action Language for the Foundational UML (ALF), which provided the author with a great deal of information and material to help fill in the blanks in the superstructure and fUML specifications. Many thanks to Ed for his strong work.
- Bran Selic – One of the originators of the UML, Bran has made significant contributions to the UML itself and more specifically my understanding of the UML.

8 Overview

A UAL *input text* is a concrete representation for UML model elements in the Foundational UML (fUML) abstract syntax subset (see fUML Specification, Clause 7). This input text is part of a wider scope UML model, parts of which are represented in UAL to provide computational completeness. This specification defines how concrete UAL input text is processed into an abstract syntax representation of UML model elements.

8.1 Model Translation and Compliance

Clause 7.1 of *"Semantics of a Foundational Subset for Executable UML Models"* [1] defines the rationale for the foundational UML subset. The fundamental purpose of fUML is "to serve as an intermediary between "surface subsets" of UML used for modeling and computational platform languages used as the target for model execution."

Figure 12 expresses this concept most clearly, and is reproduced here:

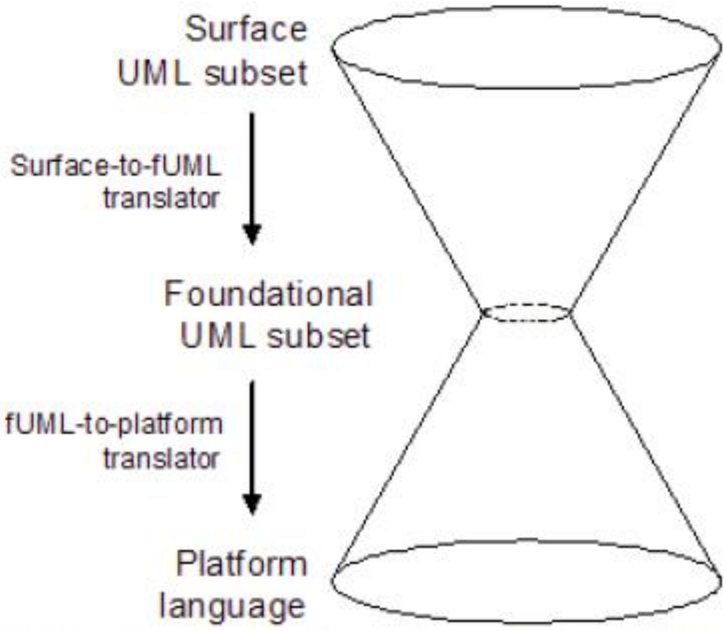


Figure 12 Translation to and from the foundational UML subset

[1] goes on to clarify the rationale in this way:

> In this context, the contents of the fUML subset have been largely determined by three criteria.
>
> - *Compactness.* The subset should be small to facilitate definition of a clear semantics and implementation of execution tools.
> - *Ease of translation.* The subset should enable straightforward translation from common surface subsets of UML to fUML and from fUML to common computational platform languages.
> - *Action functionality.* This specification only specifies how to execute the UML actions as they are currently defined with primitive functionality. Therefore, the fUML subset should not include UML functionality requiring coordinated sets of UML actions to

reproduce.

There is, of course, some tension between these criteria.

Direct translation using an intermediate format other than fUML may be common for the foreseeable future, making the second point of the rationale *"Ease of Translation"* a key to the definition of UAL. In fact, it is possible to assert a primary goal of UAL to be:

> -- *A fUML-compliant concrete syntax that can be translated to fUML, common target languages and common target platform architectures (both Von-Neumann and non-Von-Neumann.)*

UAL can, of course, be enhanced by tool vendors to provide more complete support for their tooling. Switching on fUML compliance checking should always lead to a model that can be executed on any fUML-compliant simulator or platform, or can be translated by any fUML-aware translator for a specific target platform.

8.2 Integration with UML Models

8.2.1 Introduction

The UML Superstructure specification defines the standard graphical and textual notations used to express a UML model.

With the scope of behavioral diagrams, the Activity diagram and its action language both define a surface syntax for the same abstract syntax. UAL statements and actions are mapped directly to sub-activity graphs that are interchangeable with their textual representation and share the same fUML-based semantics.

Therefore, UAL is an alternate (and more compact) notation representing portions of the model's behavior. More specifically, UAL is used to *complete* these behavioral portions that would be too complex or tedious to define visually.

While the use of a Java-like syntax brings with it great familiarity, it also brings with it a few limitations in syntax. These are easily addressed with extensions such as directives and service library calls. This retains the very familiar syntax while expanding the syntax to include all relevant parts of the UML, or more specifically the foundational UML.

All statements in a UAL fragment, including those that call into a service library, are translated to either the foundational UML or to a target platform, and therefore the syntax as a whole is the action language.

The service library is also a key extension mechanism for UAL, and is used to connect UAL into the tooling environment and to provide platform specificity (such as a semantic variation for polymorphic dispatching) when required.

The UAL environment will therefore provide access to the UML (and fUML specific) standard libraries as implemented for the local tooling environment, custom libraries designed to integrate into the local tooling environment, target platform specific libraries, and 3rd party libraries. Administrative and user configurability of the latter would be desirable.

The foundational UML standard library is defined in [1].

8.2.2 Visibility and Scope

A UAL fragment is mapped to the foundational UML as an integral part of the behavior in which the fragment appears. The containing behavior itself resides in a classifier that is known as the behavior's *context*.

All attributes and operations belonging to the context, whether owned or inherited, are visible to the behavior, and therefore to the UAL fragment.

All rules of visibility for behaviors, as defined in [2], apply to UAL fragments.

A UAL fragment is isolated within the model's scope and is restricted to the use of UML and tooling libraries in much the same way that a Java applet is isolated in a restricted scope inside a browser. This creates a secure environment within which UML semantics are not bypassed.

8.2.3 State Machines

The fUML subset does not include state machines, so direct translation of a state chart would require translation to an Activity, which further requires assumptions about the Foundational UML Virtual Machine (FVM from this point on) in order to provide state chart behaviors.

Since there is no standard fUML mapping for state charts, this specification avoids any attempt to map them directly to fUML. However, UAL fragments in state chart state bodies and transitions (for example) are in fact behaviors and thus follow the UML's Activity semantics.

This specification therefore **does** include those mappings that are more than a simple Activity or Structured Activity representing a code fragment or block.

8.2.4 Location and Storage of UAL Fragments

UAL fragments are embedded in the model wherever Action Language statements are expected – State Chart transitions, entry and exit actions, do behaviors, opaque behaviors, opaque action bodies, operation bodies – and likely within Activities and expressions.

Layer 0 conformance does not include a separate module or artifact format for UAL fragments.

8.2.5 Mapping of UAL Fragments to fUML

Mapping to the foundational UML will presume one of two possibilities:

Mapping for a simple "one statement with one action" fragment will presume the direct insertion of the mapped action into the containing Activity at the appropriate point and with the appropriate flow and pin connections.

Mapping for any other UAL fragment presumes a code block. The code block is, in fact, implied. That is, it is unnecessary to include block delimiters in a UAL fragment.

This example:

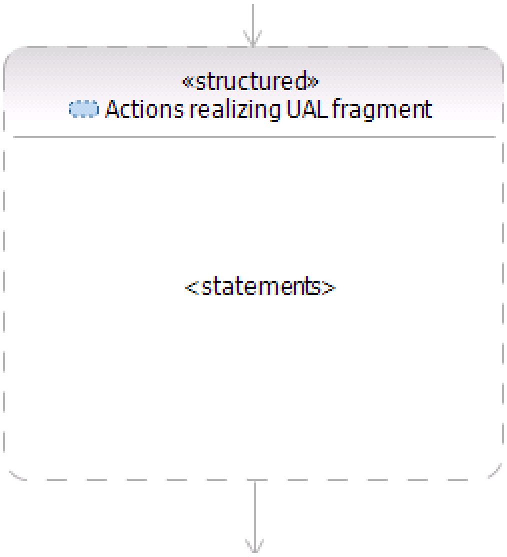
```
for  (Color c : allColors) {
   doSomethingWithAColor(c);
}
```

Does not have to be expressed as:

```
{
   for  (Color c : allColors) {
```

```
            doSomethingWithAColor(c);
        }
    }
```

Both examples result in the same mapping to fUML, which renders the fragment as a structured activity node, again inserted into the containing behavior at the appropriate point with the appropriate incoming and outgoing flows and pins.



8.3 Abstraction Level

UAL is based on the Java syntax, a very well-known syntax that has been all but ubiquitous in the education of computer scientists and software engineers for years. Thus, the familiarity that is called for in the RFP is satisfied to the largest possible extent.

However, the question “why not just use Java?” generally involves a discussion around raising the abstraction level of the solution. That is, there appears to be a general feeling that Java syntax is at the wrong abstraction level.

So how does UAL avoid being just plain old Java? The answer lies in the fact that UAL is a *surface syntax* that is by definition destined for translation to UML syntax using the foundational UML semantics as defined in [1]. Thus, UAL semantics are defined at the UML abstraction level and **not** the Java virtual machine level.

Further, UAL is defined for the purpose of completing an existing visual model that itself is defined in the UML’s syntax. It follows that UAL does not have to express every last concept in the foundational UML or in the UML Superstructure; rather it only has to make it possible to strike a practical balance between too much drawing and too much code. Further, any concept that is too advanced for UAL syntax is easily expressed visually instead.

UAL is therefore **not** defined as a separate language with its own semantics. It is **not** intended to run on a Java Virtual Machine. And it is **not** intended to be written as low level detailed algorithms, although some of that capability is retained for flexibility.

The ***for each*** statement is an example of a preferred method for processing data sets. This translates easily to any platform and either iterative or parallel algorithms. It is simplicity itself to code an expansion region in UAL, requiring only a for each loop with a designation as parallel, either through a directive or a default mode in the translator.

These differences from the Java language are not trivial. The design of UAL disallows its use as Java. It is defined as a surface syntax only, to be translated to either the foundational UML or to a target language and platform, at which time UML semantics are always in force.

Only a conforming implementation is relevant to this specification, and ***a conforming implementation cannot substitute Java semantics for UML semantics at the level of the fUML virtual machine***.

One further point – in UAL, the modeler works with UML concepts such as signals, ports and links directly. These use very familiar syntax based on constructor syntax for Signals and collection syntax for ports and links, both of which are modeled as a combination of set syntax for unordered sets and list syntax for ordered sets.

Therefore, it follows that UAL, despite its Java legacy, is in fact a concrete syntax for expression of systems at the UML abstraction level.

8.4 Specification Organization

The surface language lexical structure is discussed in depth in clause 9. Cause 10 discusses UAL blocks and statements. And clause 11 discusses the UML actions supported in UAL.

9 Lexical Structure

UAL borrows its lexical structure from [4]. Additional *translator directives* are included as commands to translators where useful to extend the UAL concrete syntax. These extensions are implemented using comments in a special format, which look similar to the very familiar *pragma* directives from the c++ preprocessor.

9.1 UAL Conformance Levels

This specification includes a full definition of an action language at conformance level zero (L0.) This includes basic syntax for statements, blocks, looping, choice and so on. It does not contain any detailed information for an advanced layer one (L1) conformance level as discussed in section 3.

There are a few small syntax additions between UAL L0 and UAL L1. In accordance with the desire expressed in the RFP [2] for a familiar and widely used language base for UAL, the syntax at UAL L0 for identifiers and qualified path names (for example) are those of Java. In UAL L0, this syntax is sufficient to claim conformance.

These constructs are easily read and understood by a massive target audience in many industries, and are equally easily translated by vendors to the two dominant target platform languages, Java and C++.

Note that L1 is expected to add the "::" syntax to L0's "." syntax for qualified names; also quoted identifiers with embedded spaces along with the commensurate mapping requirements for translation.

9.2 LL(k) Grammar

The following UAL grammar defines the core language syntax for L0.

```
Identifier:
        IDENTIFIER

Literal:
        IntegerLiteral
        | FloatingPointLiteral
        | CharacterLiteral
        | StringLiteral
        | BooleanLiteral
        | NullLiteral


Block:
        "{" [BlockStatements] "}"

BlockStatements:
        { BlockStatement }

BlockStatement :
        LocalVariableDeclarationStatement
        | [Identifier :] Statement


LocalVariableDeclarationStatement:
        Type VariableDeclarators ";"
```

```
Statement:
        Block
        | "if" "(" Expression ")" Statement ["else" Statement]
        | "for" "(" ForControl ")" Statement
        | "while" "(" Expression ")" Statement
        | "do" Statement "while" "(" Expression ")" ";"
        | "try" Block ( Catches | [Catches] "finally" Block )
        | "switch" "(" Expression ")" "{" SwitchBlockStatementGroups "}"
        | "return" [Expression] ";"
        | "throw" Expression ";"
        | "break"
        | "continue"
        | ";"
        | StatementExpression ";"
        | Identifier ":" Statement


SwitchBlockStatementGroups:
        { SwitchBlockStatementGroup }

SwitchBlockStatementGroup:
        SwitchLabel BlockStatements

SwitchLabel:
        "case" ConstantExpression ":"
        | "case" EnumConstantName ":"
        | "default" ":"

Type:
        Identifier {"." Identifier} {"[" "]"}
        | BasicType

StatementExpression:
        Expression

ConstantExpression:
        Expression


BasicType:
        "int"
        | "long"
        | "boolean"
        | "String"


VariableDeclarators:
        VariableDeclarator { "," VariableDeclarator }

VariableDeclarator:
        Identifier VariableDeclaratorRest

VariableDeclaratorRest:
        {"[" "]"} [ "="  VariableInitializer]

VariableInitializer:
        ArrayInitializer
        | Expression

ArrayInitializer:
        "{" [ VariableInitializer {"," VariableInitializer} [","] ] "}"

Catches:
        CatchClause {CatchClause}

CatchClause:
        "catch" "(" ["final"] Type Identifier ")" Block

ForControl:
        ForVarControl
        | ForInit ";" [Expression] ";" [ForUpdate]

ForVarControl
        Type Identifier ForVarControlRest

ForVarControlRest:
        VariableDeclaratorsRest ";" [Expression] ";" [ForUpdate]
        | ":" Expression

ForInit:
        StatementExpression MoreStatementExpressions

ForUpdate:
        StatementExpression MoreStatementExpressions

MoreStatementExpressions:
        { "," StatementExpression }

VariableDeclaratorsRest:
        VariableDeclaratorRest { "," VariableDeclarator }

Expression:
        Expression1 [AssignmentOperator Expression1]]

AssignmentOperator:
        "="
        | "+="
        | "-="
        | "*="
        | "/="
        | "&="
        | "|="
        | "^="
        | "%="
        | "<<="
        | ">>="
        | ">>>="

Expression1:
        Expression2 [Expression1Rest]

Expression1Rest:
        "?" Expression ":" Expression1
```

```
Expression2:
        Expression3 [Expression2Rest]

Expression2Rest:
        {InfixOp Expression3}
        | Expression3 "instanceof" Type

InfixOp:
        "||"
        | "&&"
        | "|"
        | "^"
        | "&"
        | "=="
        | "!="
        | "<"
        | ">"
        | "<="
        | ">="
        | "<<"
        | ">>"
        | ">>>"
        | "+"
        | "-"
        | "*"
        | "/"
        | "%"

Expression3:
        PrefixOp Expression3
        (   Expression | Type   )   Expression3
        Primary {Selector} {PostfixOp}

PrefixOp:
        "++"
        | "--"
        | "!"
        | "~"
        | "+"
        | "-"

PostfixOp:
        "++"
        | "--"

Primary:
        "(" Expression ")"
        | NonWildcardTypeArguments
                 (ExplicitGenericInvocationSuffix | "this" Arguments)
        | "this" [Arguments]
        | "super" SuperSuffix
        | Literal
        | "new" Creator
        | Identifier { "." Identifier } [IdentifierSuffix]
        | BasicType {"[" "]"} "." "class"
        | "void" "." "class"
```

```
Selector:
        "." Identifier [Arguments]
        | "." ExplicitGenericInvocation
        | "." "this"
        | "." "super" SuperSuffix
        | "." new [NonWildcardTypeArguments] InnerCreator
        | "[" Expression "]"

SuperSuffix:
        Arguments
        | "." Identifier [Arguments]

Arguments:
        "(" [ Expression { "," Expression } ] ")"

NonWildcardTypeArguments:
        "<" TypeList ">"

TypeList:
        Type { "," Type }

ExplicitGenericInvocation:
        NonWildcardTypeArguments ExplicitGenericInvocationSuffix

ExplicitGenericInvocationSuffix:
        "super" SuperSuffix
        | Identifier Arguments

Creator:
        [NonWildcardTypeArguments] CreatedName
                 ( ArrayCreatorRest | ClassCreatorRest )

CreatedName:
        Identifier [NonWildcardTypeArguments]
                {"." Identifier [NonWildcardTypeArguments]}

InnerCreator:
        Identifier ClassCreatorRest

ArrayCreatorRest:
        "[" "]" {"[" "]"} ArrayInitializer
        | "[" Expression "]" {"[" Expression "]"} {"[" "]"}

ClassCreatorRest:
         Arguments

IdentifierSuffix:
        "[" "]" {"[" "]"} "." "class"
        | "[" Expression "]"
        | Arguments
        | "." ("class"
               | ExplicitGenericInvocation
               | "this"
               | "super" Arguments
               | "new" [NonWildcardTypeArguments] InnerCreator
              )
```

9.3 Line Terminators

Translators divide the input sequence into *lines* by recognizing *line terminators*. This definition of lines determines the line numbers reported by translators and UAL debuggers. It also specifies the termination for end-of-line comments.

9.3.1 Notation

LineTerminator:

  the Unicode platonic LF character, also known as “newline”
  the Unicode platonic CR character, also known as “return”
  the Unicode platonic CR character followed by the Unicode platonic LF character

InputCharacter:

  InputCharacter but not CR or LF

9.4 Input Elements and Tokens

The input elements and tokens that result from input line recognition are reduced to a sequence of input elements. Those input elements that are not white space or comments are considered *tokens*. The tokens are terminal symbols of the syntactic grammer.

9.4.1 Notation

Input:

  InputElements$_{opt}$ *Sub*$_{opt}$

InputElements:

  InputElement
  InputElements InputElement

InputElement:

  WhiteSpace
  Comment
  Token

Token:

  Identifier
  Keyword
  Literal
  Separator
  Operator

Sub:

  the Unicode platonic SUB character, also known as “control-Z”

9.4.2 Semantics

White space and comments can separate tokens that, when adjacent, might be tokenized in another manner. An example is the **“-=”** operator, which can only form the operator token when there is no intervening white space or comment.

Although an input stream is usually displayed as a sequence of lines in two dimensions, a token that appears later in the input stream is said to be “to the right” of a token that appeared before it. For example, in this short fragment:

```
    for (x,y,z) {
    }
```

We say that the `}` token appears to the right of the `{` token in the input stream despite its appearance in text of being downward and to the left.

The `SUB` character is taken as the last character of the fragment if it appears at all.

9.5 White Space

White space is defined as the Unicode platonic space, horizontal tab, and form feed characters, as well as line terminators.

9.5.1 Notation

WhiteSpace:
the Unicode platonic SP character, also known as “space”
the Unicode platonic HT character, also known as “horizontal tab”
the Unicode platonic FF character, also known as “form feed”
LineTerminator

9.6 Comments

UAL has two notations for comments – end-of-line and block.

The symbol “//” is used to start an *end-of-line comment*. The symbol and all following text, up to and including the line terminator character(s), are ignored. The symbols “/*” and “*/” are used to start and end either a *single-line* or a *multi-line* comment.

These symbols and all characters in between these symbols including white space and new-line characters are ignored unless they are of the specific format for translator directives or documentation as defined elsewhere in this specification.

9.6.1 Notation

Comment:
BlockComment
EndOfLineComment
BlockComment:
/ * *CommentTail*
EndOfLineComment:
/ / *CharactersInLine*$_{opt}$
CommentTail:
* *CommentTailStar*
NotStar CommentTail
CommentTailStar:
/
* *CommentTailStar*
NotStarNotSlash CommentTail
NotStar:
InputCharacter but not *
LineTerminator
NotStarNotSlash:
InputCharacter but not * or /

    LineTerminator
CharactersInLine:
    InputCharacter
    CharactersInLine InputCharacter

9.6.2 Semantics

Comments do not nest.

`/*` and `*/` have no special meaning in end-of-line comments.

`//` has no special meaning inside block comments.

The following fragment is a single, complete comment:

```
/* this comment /* // /** ends here: */
```

The use of a block comment with the prefix “`/*-`“ will be understood by editors, formatters and translators as immunity from reformatting.

A comment cannot occur within a name or a string literal.

Documentation: The use of a block comment with the prefix “`/**`” will be understood by editors, formatters and translators as *documentation* for the following UAL fragment. These comments must be extracted by the tooling environment and mapped to UML comments.

They may appear at the beginning of any block. Translators shall copy them to the target as comments in the target language format.

Note that comments that are not specifically in the documentation format may be mapped to UML comments as well, the mapping of which is defined in the following sections.

9.6.3 Example

```
<ual statements>
// an end-of-line comment
<ual statement> // an end-of-line comment
<more ual statements>
/*
                a
                multi-line
                comment
*/
<more ual statements>
/* a single line comment */
```

9.6.4 Mapping

A generalized mapping for end of line comments cannot preserve their meaning exactly. However, the following example describes a mapping for translation.

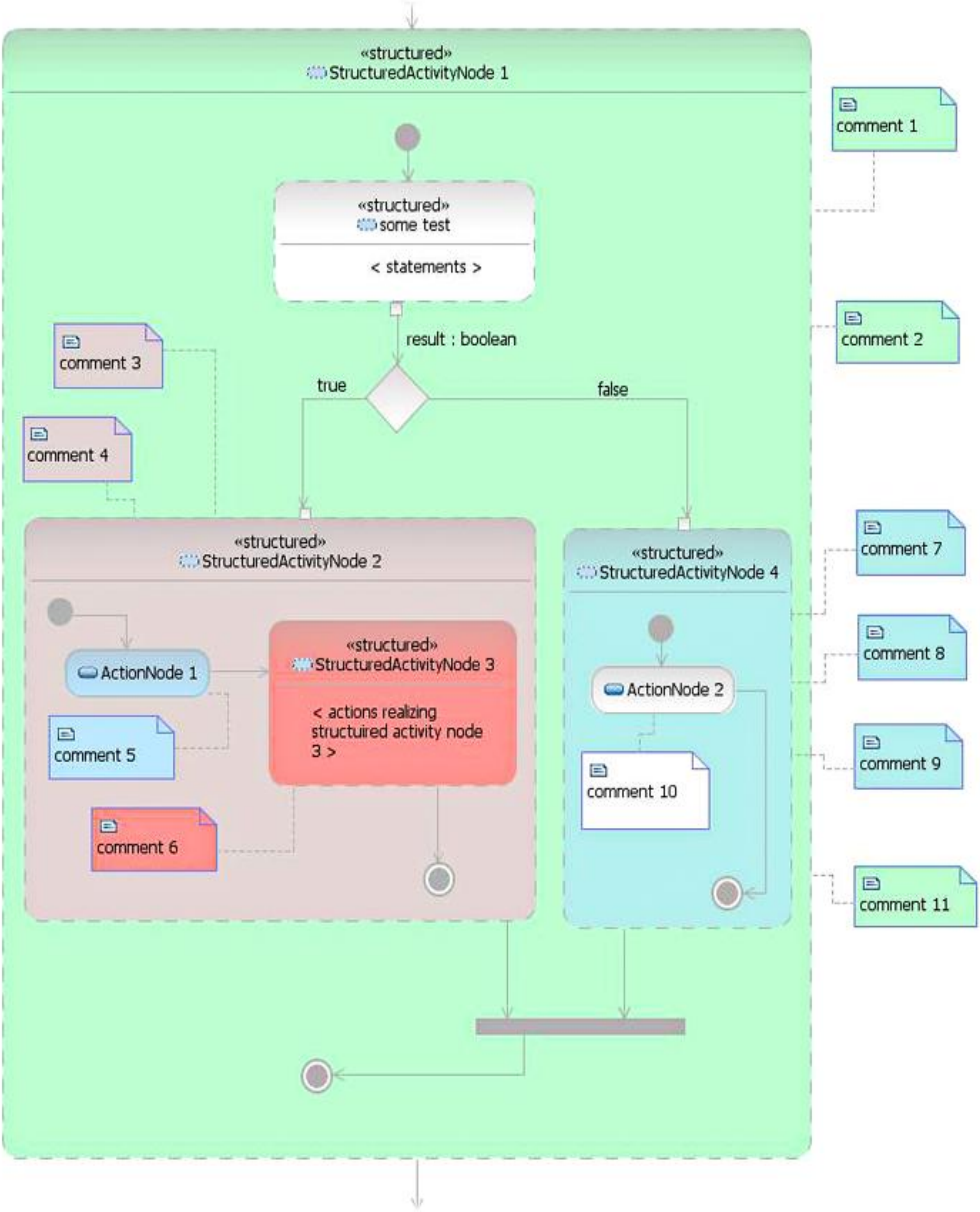
```
{ // EOL comment 1
   /* block comment 2 */
   if (some test) { // EOL comment 3
      // EOL comment 4
      < statement 1 > // EOL comment 5
      { // EOL comment 6
         < local var 1 >
         < statement 2 >
      }
```

```
        } else // EOL comment 7
        /* block comment 8 */
        { // EOL comment 9
           < statement 3> // EOL comment 10
        }
        // EOL comment 11
     }
```

A UML comment can be generated for each of the 11 comments shown in this example block, but each of those must associate with some fUML construct. So, in order:

- The outer block maps to structured activity node 1
- EOL comment 1 associates to structured activity node 1
- BLOCK comment 2 also associates to structured activity node 1 *if it is not a directive, in which case it is associated to the following statement – this condition will be stated only this one time*
- The THEN block maps to a nested structured activity node 2
- EOL comment 3 associates to structured activity node 2
- EOL comment 4 also associates to structured activity node 2
- Statement 1 maps to an appropriate action node 1
- EOL comment 5 associates to action node 1
- The nested block inside the THEN block maps to nested structured activity node 3 inside structured activity node 2
- Local var 1 maps to a block frame object (as described in clause 10.3) inside structured activity node 3 containing appropriate action nodes
- Statement 2 maps to an appropriate action node 2 inside structured activity node 3
- EOL comment 6 associates to structured activity node 3
- The ELSE block maps to a nested structured activity node 4 inside structured activity node 1
- EOL comment 7 associates to structured activity node 4
- BLOCK comment 8 associates to structured activity node 4
- EOL comment 9 associates to structured activity node 4
    - Note that there can be no logical mapping difference between those comments that appear after the else but outside the block versus those comments that appear inside the block but are not obviously associated with a statement
- Statement 3 maps to an appropriate action node 3 inside structured activity node 4
- EOL comment 10 associates to action node node 3
- EOL comment 11 associates to structured activity node ActionNode 1

The following color-coded diagram shows these associations.



9.7 Translator Directives

UAL *translator directives* provide a generalized ability to communicate with tools and translators. This is needed in a surface language that is *intended for translation* into another format before execution.

UAL directives use the familiar *pragma* style within a comment in order that any block or statement may be translated with a specific output pattern. This is the equivalent of a stereotype application on a statement or block.

9.7.1 Notation

Directive:
    //@<Keyword>
Keyword:
    parallel
    iterative
    stream
    inline <Language>
Language:
    <string>

TBD: Are there more?

9.7.2 Semantics

@iterative

Indicates a requirement that an element list is to be processed in sequential order. The translator does not have permission to generate parallel processing for the list elements.

@parallel

Indicates no requirement for sequential processing of an element list. The translator is free to generate parallel processing for the list elements. The translator is *not required* to generate parallel processing for list elements, for example when the target environment is known to support only sequential processing.

@stream

Indicates that the incoming set is in fact a stream and that the translator shall generate a streamed expansion region.

@inline <language>

Signifies the presence of an inline sequence of code that is to be copied to the target code body by the translator. The inline sequence is ignored by translators and debuggers.

The language keyword is a string. The use of inline code is a target-specific technique, and therefore has no standard list of supported languages. Recommended language strings are, however, useful for the potential interoperability of commonly embedded language fragments. These include:

- Java
- C
- C++

- C#
- OCL

9.7.3 Defaults

A target environment may be configured so that the default mapping mode for all qualifying looping constructs implies the `@parallel` directive. T*he mechanism for this configuration is implementation-dependent.*

Parallel execution is therefore mapped based on:

- An `@parallel` directive in the surface language code fragment preceding a `for each` loop construct or a translator that is configured to map to parallel execution by default; and
- A target that supports parallel execution; and

In any tooling environment that is configured to generate either iterative or parallel execution by default, a translator directive *allows* the translator to generate a parallel threading algorithm.

Parallel directives **do not** force the generation of parallel target code.

Iterative directives **do** force the generation of iterative target code.

9.7.4 Example

```
//@parallel
for (ElementType element : aList) {
   doSomethingWith(element);
}
```

Note: in order to be recognized as a directive rather than a simple comment, the directive must immediately follow the end-of-line comment marker. There can be no intervening space between the `//` and the `@keyword`.

Directives of either style serve as input to translators, causing specific *output patterns* to be generated into the target stream.

Note that new directives may not be *defined* in core UAL language fragments, but the standard set of *translator directives*, as defined in clause **Error! Reference source not found.** may be used.

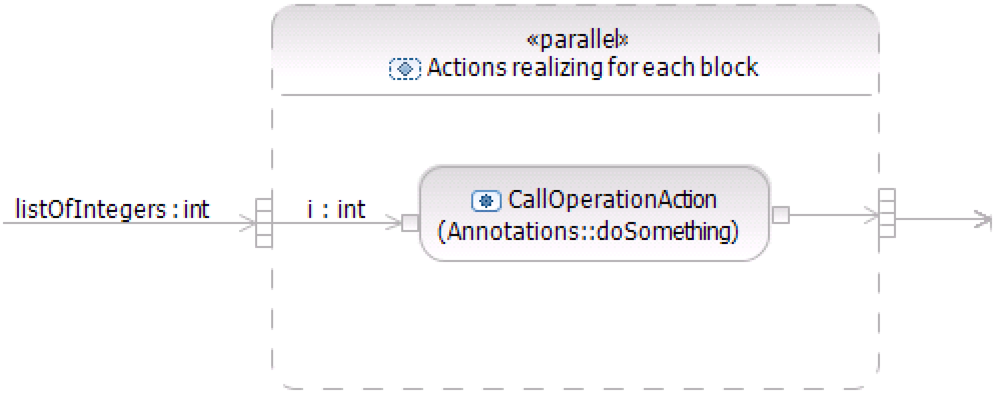
9.7.5 Mapping

9.7.5.1 @parallel

The @parallel translator directive will cause each loop within its scope to be mapped to an expansion region. Iterative maps the same way and is not shown.

```
//@parallel
for (int i : listOfIntegers) {
   doSomething(i);
}
```

Maps to:

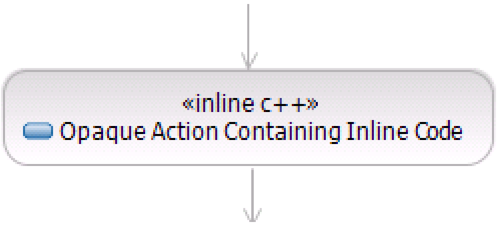


9.7.5.2 @inline <language>

The @inline <language> translator directive will cause the following code (contained within the same block comment) to be copied to the target implementation without further interpretation.

```
/*@inline c++
<c++ statements>
*/
```

Maps to:



9.8 Documentation Text

Block comments beginning with the token `/**` are considered to be *documentation text*.

Documentation text typically appears at the beginning of a fragment or block. Documentation text is intended to appear in reports, generated code, and documentation such as help, developer guides and potentially user guides.

9.8.1 Mapping

Documentation text is mapped to a UML comment element stereotyped as <<documentation>>. When the translation target is a text language, the documentation text can be emitted into the source files in the language's documentation text format.

Documentation text can appear once per block of code.

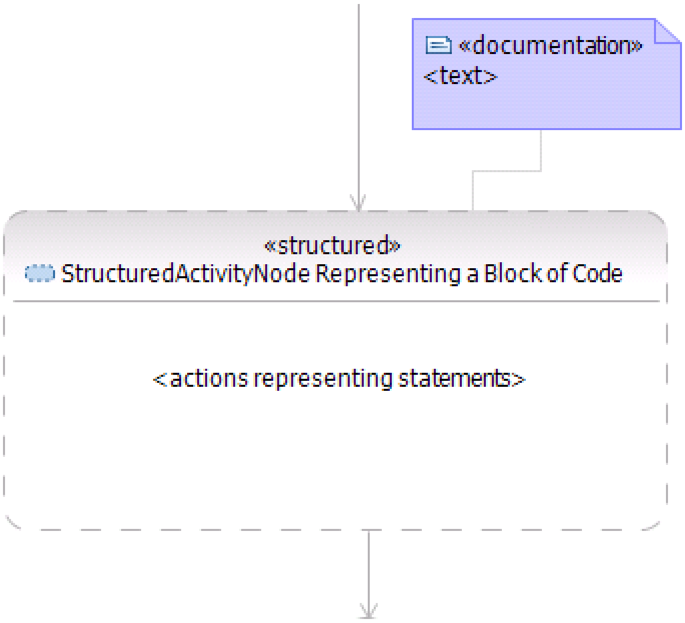
```
/**
<text>
*/
```

or

```
/**
```

```
 * <text>
 */
```

Both map as:


«documentation»
<text>
«structured»
StructuredActivityNode Representing a Block of Code
<actions representing statements>

9.9 Names (Identifiers)

Each target environment will have its own rules for naming of identifiers and the appropriate mapping will generally require mapping and translation of identifiers to something suitable for the target language.

In order to provide a high degree of simplicity and reliability for the default mappings, conforming identifiers in UAL have a strict definition that aligns well with common target languages like Java and C++.

9.9.1 Notation

An *identifier* is an unlimited-length sequence of UAL letters and UAL digits, and must begin with a UAL letter.

Identifier:
    IdentifierChars but not a *Keyword* or *BooleanLiteral* or *NullLiteral*
IdentifierChars:
    UALLetter
    IdentifierChars UALLetterOrDigit
UALLetter:
    any Unicode character that is a UAL letter
UALLetterOrDigit:
    any Unicode character that is a UAL letter-or-digit

9.9.2 Semantics

An identifier cannot have the same spelling as a keyword, a Boolean literal, or the null literal.

A UAL letter is any of the Unicode platonic characters within the ranges “a-z” and “A-Z”. It can also be the Unicode platonic underscore character “_”.

A UAL digit is any of the Unicode platonic characters within the range “0-9”.

9.10 Qualified Names

The familiar dot notation is used in UAL to reference a name that is not within the local scope. Thus, to reference an item in a top-level library “LIB”, one would use the sequence:

```
LIB.doSomething();
```

9.10.1 Mapping

The dot “.” will map to the UML symbol “::” when translating to qualified paths in fUML.

9.10.2 Semantics

A UAL fragment may define a new unqualified name as a local variable with visibility restricted otg the block within which it is defined. See clause 10.3.2.

A local variable may shadow an existing name, in which case the existing name must be referenced with its complete path. A name within the context of the current behavior can be referenced with the `this.attribute` notation. A name within a service library may be referenced with its qualified path, for example `LIB.doSomething`.

All referenced names must be visible within the current scope.

9.11 Reserved Words

The following are *reserved* for use as keywords in UAL and cannot be used as identifiers.

```
for  break switch  default  if  boolean  do  this  throw  else  import
throws  case  instanceof  return  catch  int  long try  void  finally
long  super  while
```

9.12 Literals

A literal represents a value of a primitive type.

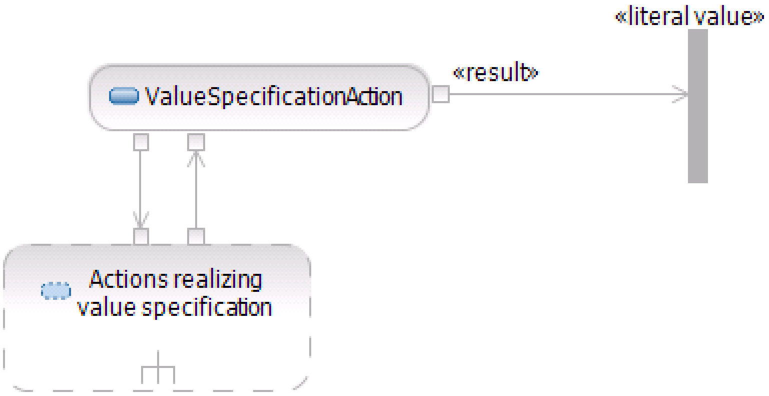
9.12.1 Notation

Literal:

IntegerLiteral
BooleanLiteral
CharacterLiteral
StringLiteral
NullLiteral

9.12.2 Mapping

Literals map to the Value Specification Action with a block for the value specification and an outgoing pin with the resulting constant and of a matching type.



9.12.3 Integer Literals

An integer literal may be expressed in decimal, hexadecimal or octal.

9.12.3.1 Notation

IntegerLiteral:
  DecimalIntegerLiteral
  HexIntegerLiteral
  OctalIntegerLiteral
DecimalIntegerLiteral:
  DecimalNumeral IntegerTypeSuffix$_{opt}$
HexIntegerLiteral:
  HexNumeral IntegerTypeSuffix$_{opt}$
OctalIntegerLiteral:
  OctalNumeral IntegerTypeSuffix$_{opt}$
IntegerTypeSuffix: one of
  l L

DecimalNumeral:
  0
  NonZeroDigit Digits$_{opt}$
Digits:
  Digit
  Digits Digit
Digit:
  0
  NonZeroDigit
NonZeroDigit: one of
  1 2 3 4 5 6 7 8 9

HexNumeral:
  0 x *HexDigits*
  0 X *HexDigits*

HexDigits:
HexDigit
HexDigit HexDigits

HexDigit: one of
0 1 2 3 4 5 6 7 8 9 a b c d e f A B C D E F

OctalNumeral:
0 *OctalDigits*
OctalDigits:
OctalDigit
OctalDigit OctalDigits
OctalDigit: one of
0 1 2 3 4 5 6 7

9.12.3.2 Semantics

The largest decimal literal of type `int` is `2147483647` (2^{31}-1).

The largest positive hexadecimal and octal literals of type `int` are 0x7fffffff and 017777777777 respectively, both of which equal $2^{31}-1$.

Octals have two or more characters in them, the single digit `0` is always assumed to be decimal, although in practice this is irrelevant.

A trailing L or l types the literal as a long integer, an expanded range integer (2^{63}-1). The “`L`” is preferred as the lower case “`l`” is easily confused with the digit 1.

9.12.4 Boolean Literals

The Boolean type has two values, represented by the literals `true` and `false`. A Boolean literal is always of type boolean.

9.12.4.1 Notation

BooleanLiteral: one of
true false

9.12.5 String Literals

A string literal consists of zero or more characters enclosed in double quotes.

9.12.5.1 Notation

StringLiteral:
" *StringCharacters$_{opt}$* "
StringCharacters:
StringCharacter
StringCharacters StringCharacter
StringCharacter:
InputCharacter but not " or \
EscapeSequence
EscapeSequence:

\ b /* \u0008: backspace BS */
\ t /* \u0009: horizontal tab HT */
\ n /* \u000a: linefeed LF */
\ f /* \u000c: form feed FF */
\ r /* \u000d: carriage return CR */
\ " /* \u0022: double quote " */
\ ' /* \u0027: single quote ' */
\ \ /* \u005c: backslash \ */
OctalEscape /* \u0000 to \u00ff: from octal value */

OctalEscape:
\ *OctalDigit*
\ *OctalDigit OctalDigit*
\ *ZeroToThree OctalDigit OctalDigit*

OctalDigit: one of
0 1 2 3 4 5 6 7

ZeroToThree: one of
0 1 2 3

9.12.5.2 Mapping

A string literal should always be mapped to the same instance of class String. The same holds true for boolean literals.

9.12.5.3 Semantics

An escape sequence (\”) may be used to allow the use of the literal character “ inside a String.

Line terminators cannot be used as input characters.

A long string can be broken into multiple strings on separate lines and joined using the string concatenation operator.

9.12.5.4 Examples

```
“”                        // the empty string
“\””                      // a string containing only a single double-quote
“this is a string”        // a string of 16 characters
“this is a “ +
    “string”              // the same string
```

9.12.6 Escape Sequences

Non-graphic characters can be represented using a standard list of UAL escape sequences.

```
\b              // backspace BS
\t              // horizontal tab HT
\n              // linefeed LF
\f              // formfeed FF
\r              // carriage return CR
\”              // double quote
\’              // single quote
\\              // backslash
```

9.13 Null

The null literal has one value, the null reference.

9.13.1 Notation

NullLiteral:

    null

9.13.2 Mapping

The null literal is used in UAL as a placeholder for a missing optional parameter.

9.14 Separators (Punctuation)

9.14.1 Notation

Separator: one of

```
( ) { } [ ] ; , .
```

9.15 Operators

9.15.1 Notation

Operator: one of

```
= > < ! ~ ? :
== <= >= != && || ++ --
+ - * / & | ^ % << >> >>>
+= -= *= /= &= |= ^= %= <<= >>= >>>=
```

10 Blocks and Statements

10.1 Blocks

A *block* is a sequence of statements and local variable declaration statements within braces.

10.1.1 Notation

Block:

{ *BlockStatements*$_{opt}$ }

BlockStatements:

BlockStatement

BlockStatements BlockStatement

BlockStatement:

LocalVariableDeclarationStatement

Statement

A block is translated by translating each of the local variable declaration statements and other statements in order from first to last (left to right). Contained blocks are translated recursively.

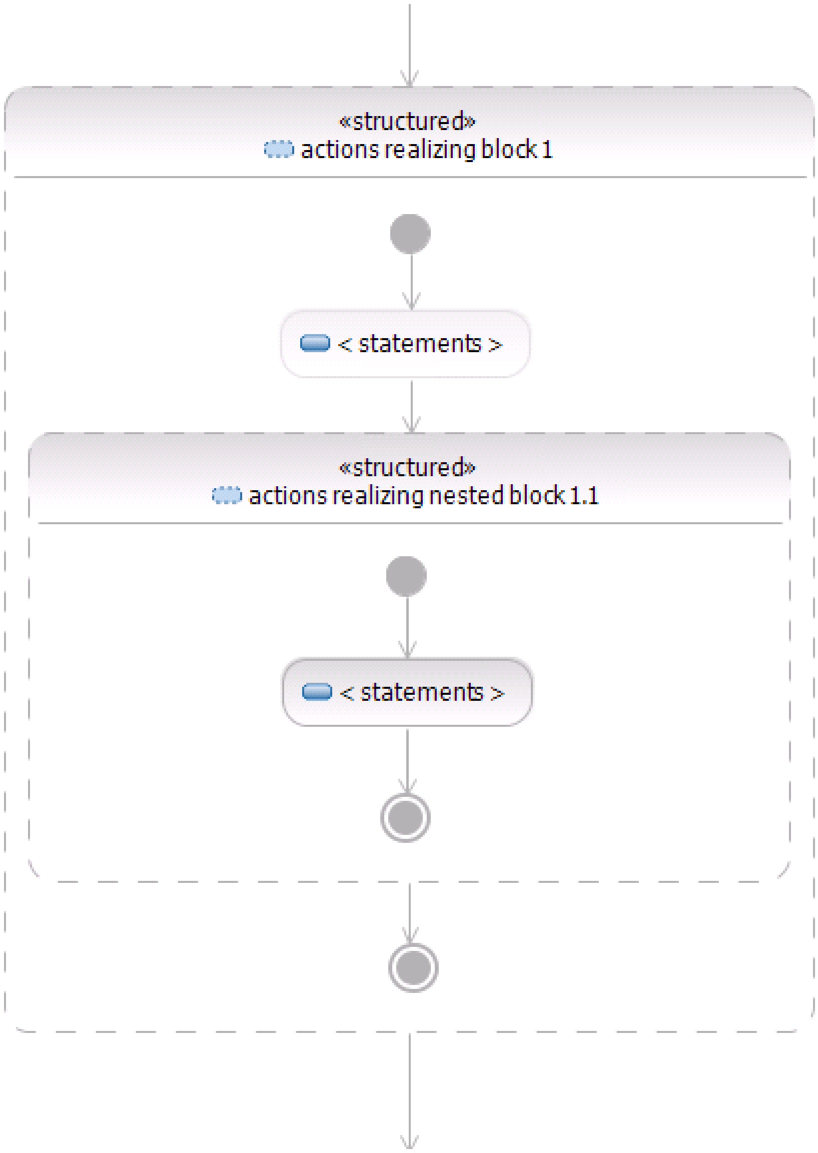
10.1.2 Example

```
{
   < statements in block 1 >
   {
      < statements in nested block 1.1 >
   }
}
```

10.1.3 Mapping

A block maps to a structured activity node contained within its parent block. Each individual statement type will be mapped separately in the following sections.

The example maps to:



10.2 Statements

There are several kinds of statements in UAL. They show their C and Java legacy and correspond quite closely to their equivalents in those languages.

10.2.1 Notation

Statement:

  StatementWithoutTrailingSubstatement
  IfThenStatement

IfThenElseStatement
WhileStatement
ForStatement

StatementWithoutTrailingSubstatement:
Block
EmptyStatement
ExpressionStatement
AssertStatement
SwitchStatement
BreakStatement
ContinueStatement
DoStatement
ReturnStatement
ThrowStatement
TryStatement

StatementNoShortIf:
StatementWithoutTrailingSubstatement
IfThenElseStatementNoShortIf
WhileStatementNoShortIf
ForStatementNoShortIf

IfThenStatement:
if (*Expression*) *Statement*
IfThenElseStatement:
if (*Expression*) *StatementNoShortIf* else *Statement*
IfThenElseStatementNoShortIf:
if (*Expression*) *StatementNoShortIf* else *StatementNoShortIf*

10.2.2 Mapping

Each individual statement type is mapped to fUML in the following sections.

10.3 Local Variable Declaration Statements

A *local variable declaration statement* declares one or more local variable names.

10.3.1 Notation

LocalVariableDeclarationStatement:
LocalVariableDeclaration ;
LocalVariableDeclaration:
VariableModifiers Type VariableDeclarators

VariableDeclarators:
VariableDeclarator
VariableDeclarators , *VariableDeclarator*
VariableDeclarator:
VariableDeclaratorId
VariableDeclaratorId = *VariableInitializer*
VariableDeclaratorId:

Identifier
VariableDeclaratorId [ ]

VariableInitializer:
Expression
ArrayInitializer

Every local variable declaration statement is immediately contained by a block. Local variable declaration statements may be intermixed freely with other kinds of statements in the block.

10.3.2 Semantics

A local variable declaration can also appear in the header of a `“for”` statement. In this case it is executed in the same manner as if it were part of a local variable declaration statement.

The scope of a local variable is the block within which it is declared. The translator should flag a warning if the local variable is redefined (shadowed) within an enclosed block, as the outer variable thus becomes inaccessible. Otherwise, local variable is always accessible using just its name.

A local variable can *shadow* the name of an attribute of the context within which the block of UAL containing the local variable definition is itself defined. The outer declaration is then shadowed throughout the scope of the local variable. The outer attribute is then only accessible using the form `this.x`.

10.3.3 Examples

A block with local variables will look like:

```
{
   int a, b;
   boolean done = false;

   <statements>
}
```

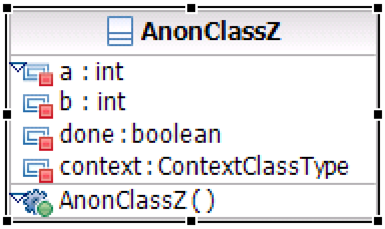
10.3.4 Mapping

A UAL block *with local variables* is translated into an instance of an auxiliary class, whose attributes are the same as the local variables. The constructor of this class sets the initial values of the local variables (null or some other specified value).

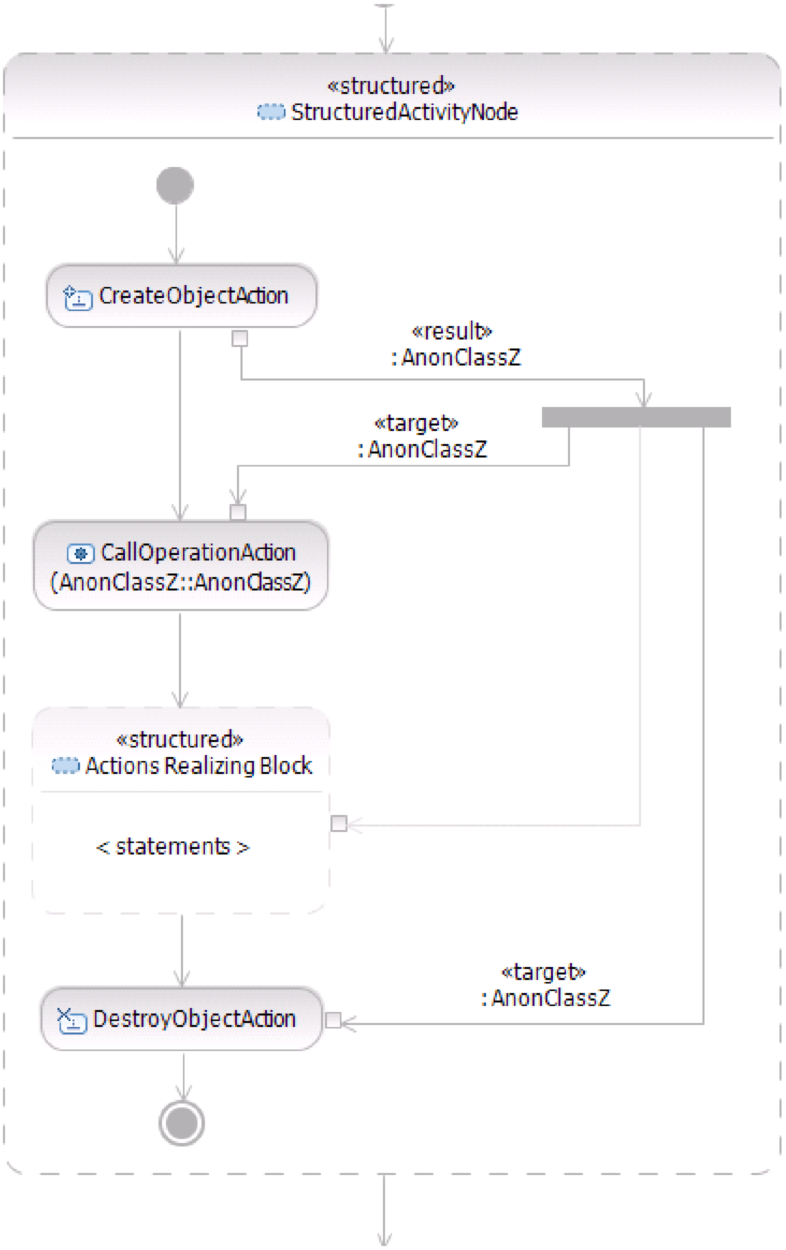
This object is called the *block frame* object and corresponds roughly to a stack frame in Algol-like programming languages. The context that contains the behavior in which this block appears is set as a reference of the same type as the class of the object that is the context for the behavior. In this way, local variables are accessed using `this.var` while attributes of the context class are accessed using `context.var`. This transparently supports shadowing of context attributes by local variables.

When the block is exited, the block frame object is destroyed.

The example in this section would result in the generation of the following anonymous class:



The initialization of default values (e.g. zero) for the `int` variables and the value of `false` to the `boolean` variable would occur in the constructor that is emitted in fUML syntax. The mapping looks like:

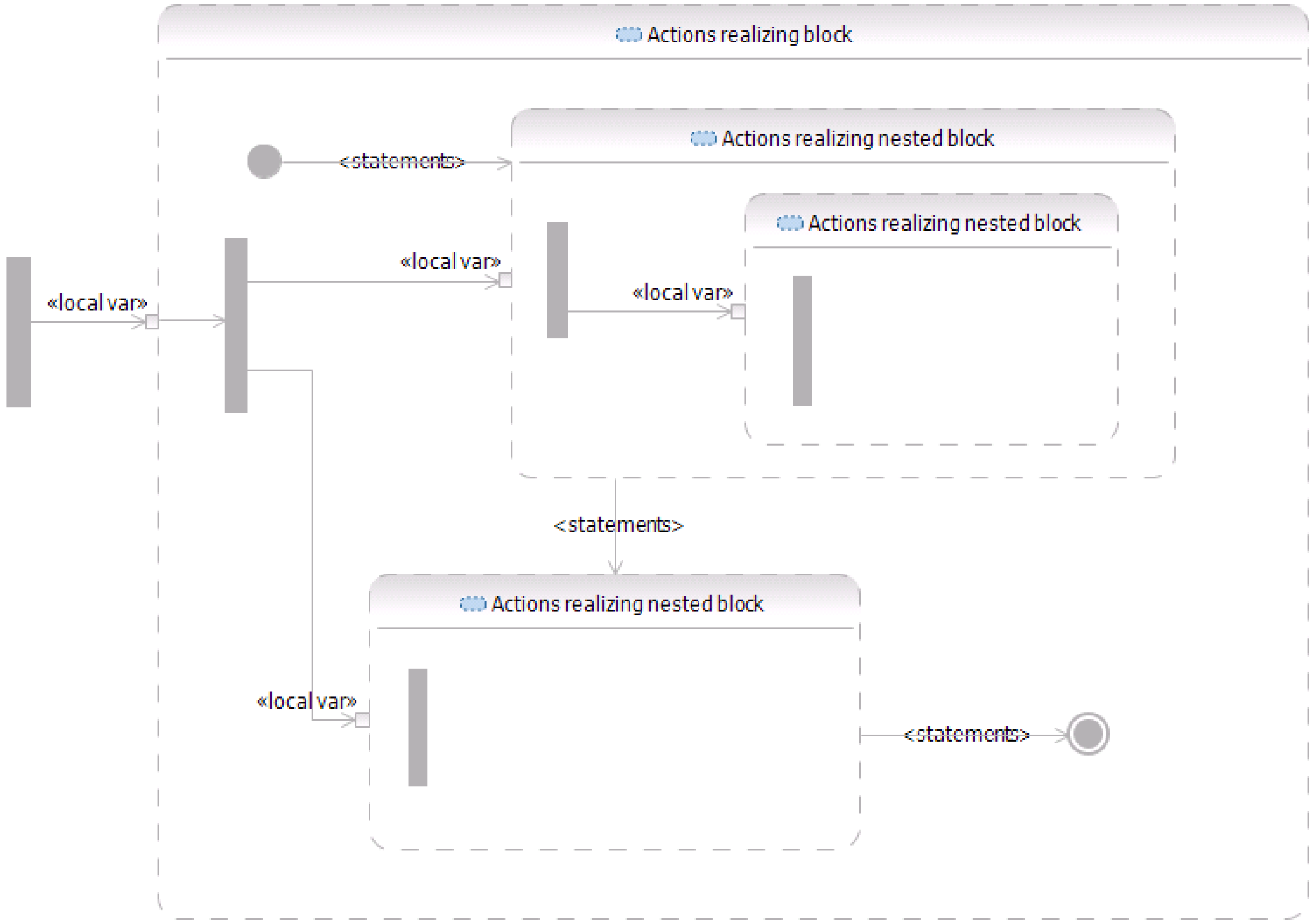


10.4 Local Variable Use

A local variable, once declared, is used in fUML translations by passing it on an input pin to every nested block in which it is referenced. Thus, it is made visible to all blocks contained within its scope.

If the local variable "shadows" an Attribute of the same name, the attribute may be accessed using "this."

10.4.1 Mapping



10.5 Expression Statements

10.5.1 Notation

Certain kinds of expressions may be used as statements by following them with semicolons:

ExpressionStatement:
    StatementExpression ;
StatementExpression:
    Assignment
    PreIncrementExpression

PreDecrementExpression
PostIncrementExpression
PostDecrementExpression
MethodInvocation
ClassInstanceCreationExpression

10.5.2 Mapping

Individual statement types are mapped within their own sections.

10.5.3 Semantics

The translator shall generate a statement that is executed by evaluating the expression. If the expression has a value, that value shall be discarded.

10.6 If Statement

10.6.1 Notation

The if statement allows conditional execution of a statement or a conditional choice of two statements, executing one or the other but not both.

IfThenStatement:
if (*Expression*) *Statement*
IfThenElseStatement:
if (*Expression*) *StatementNoShortIf* else *Statement*
IfThenElseStatementNoShortIf:
if (*Expression*) *StatementNoShortIf* else *StatementNoShortIf*

In UAL, the expression is a Boolean expression.

10.6.2 Examples

10.6.2.1 If Statement with Then Statement

```
if (x == 1) then {
   x += 2;
}
```

10.6.2.2 If Statement with Then and Else Statements

```
if (x == 1) then {
   x += 2;
} else {
   x +=3;
}
```

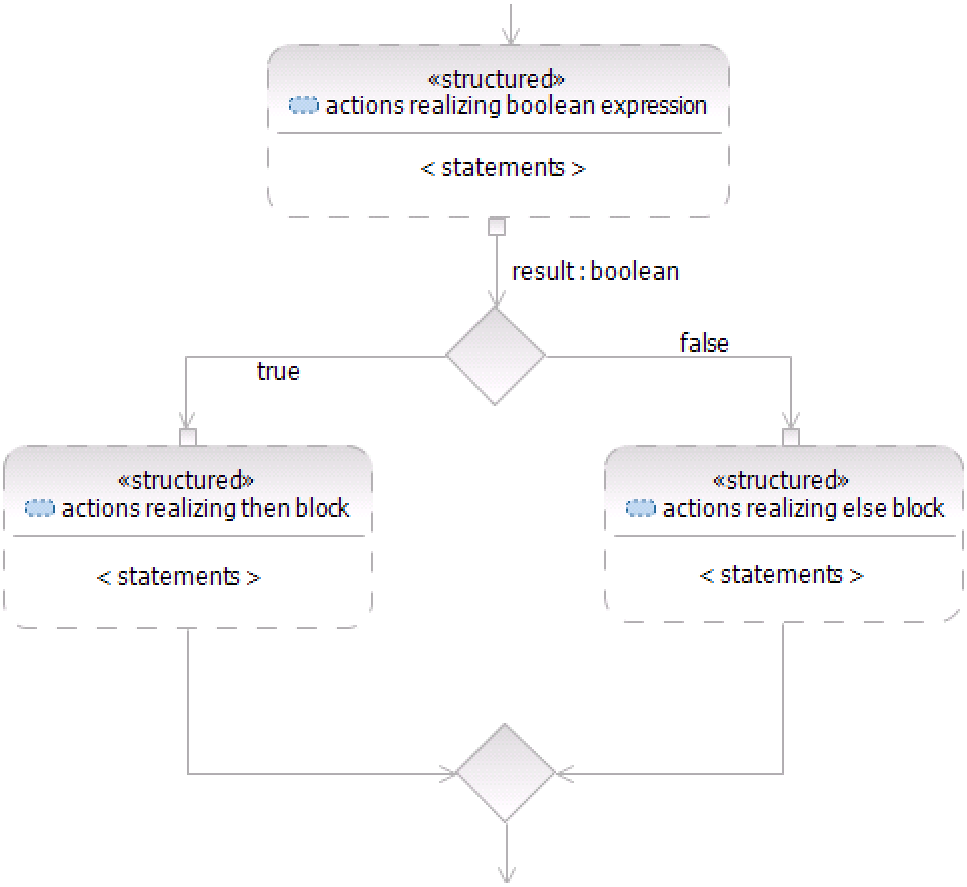
10.6.3 Mapping

The *expression* maps to a structured activity with an output pin carrying a result of type Boolean. This value flows into a decision node, with two branches, one for true and one for false. If the value is true, then the *then* block is executed, else the *else* block is executed. Each of these contained blocks further maps to a structured activity node.

10.6.3.1 If Statement with Then Statement

See next section, else block would be missing and false connection would go directly to the join.

10.6.3.2 If Statement with Then and Else Statements



10.6.4 Semantics

For an if-then statement, the intent is to execute the contained statement or block iff the expression evaluates to true.

For an if-then-else statement, the intent is to execute the first contained statement iff the expression evaluates to true, otherwise the second contained statement is executed.

10.6.5 Disambiguation

UAL suffers, as do C and Java, from the *dangling else statement* issue. The following is a classic example where misleading formatting implies that the dangling else statement is intended to belong to the outer if statement.

```
if (door.isOpen())
   if (resident.isVisible())
      resident.greet("Hello!");
else door.bell.ring(); // A “dangling else”
```

For translation purposes, however, the dangling else is in fact resolved to belong to the *innermost* if statement *to which it can belong*. So in the example, the dangling else statement belongs to the isVisible condition and not to the isOpen condition.

10.7 Switch

The switch statement transfers control to one of several statements depending on the value of an expression.

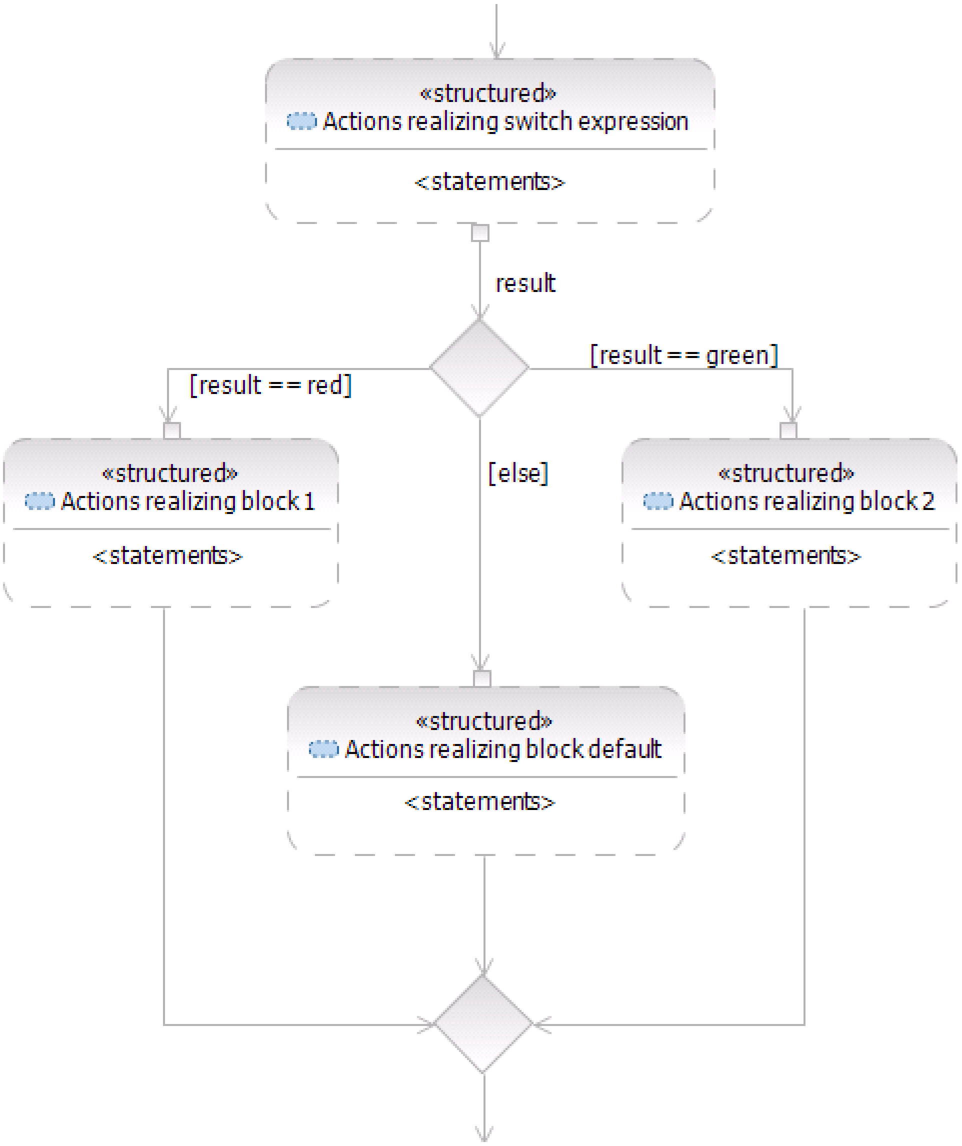
10.7.1 Notation

SwitchStatement:
    switch (*Expression*) *SwitchBlock*
SwitchBlock:
    { *SwitchBlockStatementGroups*$_{opt}$ *SwitchLabels*$_{opt}$ }
SwitchBlockStatementGroups:
    SwitchBlockStatementGroup
SwitchBlockStatementGroups SwitchBlockStatementGroup
    SwitchBlockStatementGroup:
    SwitchLabels BlockStatements
SwitchLabels:
    SwitchLabel
    SwitchLabels SwitchLabel
SwitchLabel:
    case *ConstantExpression* :
    case *EnumConstantName* :
    default :
EnumConstantName:
    Identifier

10.7.2 Mapping

The switch statement maps directly to the decision node and the default statement maps to the else outgoing edge, as shown in this example:

```
switch (expression) {
   case red: {block 1}
   case green: {block 2}
   default: {block default}
}
```



10.7.3 Semantics

The type of the expression must be one of integer, UnlimitedNatural or enumeration. Each outgoing edge has a guard expressed as a constant expression and the modeler must ensure that these expressions have unique values; else the outgoing path is non-deterministic. The outgoing guards are not guaranteed to be evaluated in any particular order – the exception being the else edge, which is followed if no other edge is followed; hence it is critical that expressions in guards have no side effects.

Every case constant expression associated with a switch statement must be assignable to the type of the switch expression.

No switch label can be null.

No two of the constant expressions associated with a switch statement may have the same value.

At most one default label may be associated with a switch statement. It will map to the else edge, on which there is the same restriction.

The translator should provide a warning if a switch statement on an enumeration does not have a default statement.

If there is no matching switch label and there is no default label, then the switch statement is intended to complete normally with no action taken.

10.8 Break

A break statement terminates any enclosing *switch*, *while*, *do*, *for* or *for each* statement normally.

10.8.1 Notation

BreakStatement:

```
        break;
```

10.8.2 Semantics

No further processing will take place in the switch, while, do, for or for each statement after the break is executed. However, should the break be executed from inside a *try* block, and that *try* block have one or more *finally* clauses, then the *finally* clauses will execute in order before the enclosing statement terminates.

A key difference in behavior occurs when loops are executed in <<parallel>> versus <<iterative>> or <<stream>>. With <<parallel>>, iterations of the loop are executed independently of all others, so the break statement provides early termination of that iteration only.

With <<iterative>> or <<stream>>, the loop is executed in serial order and the break statement provides early termination of the entire loop.

As always, modelers and developers must be keenly aware of the behaviors they are managing when using parallel processing.

If early termination of a loop is desirable, then the loop must be modeled as <<iterative>>.

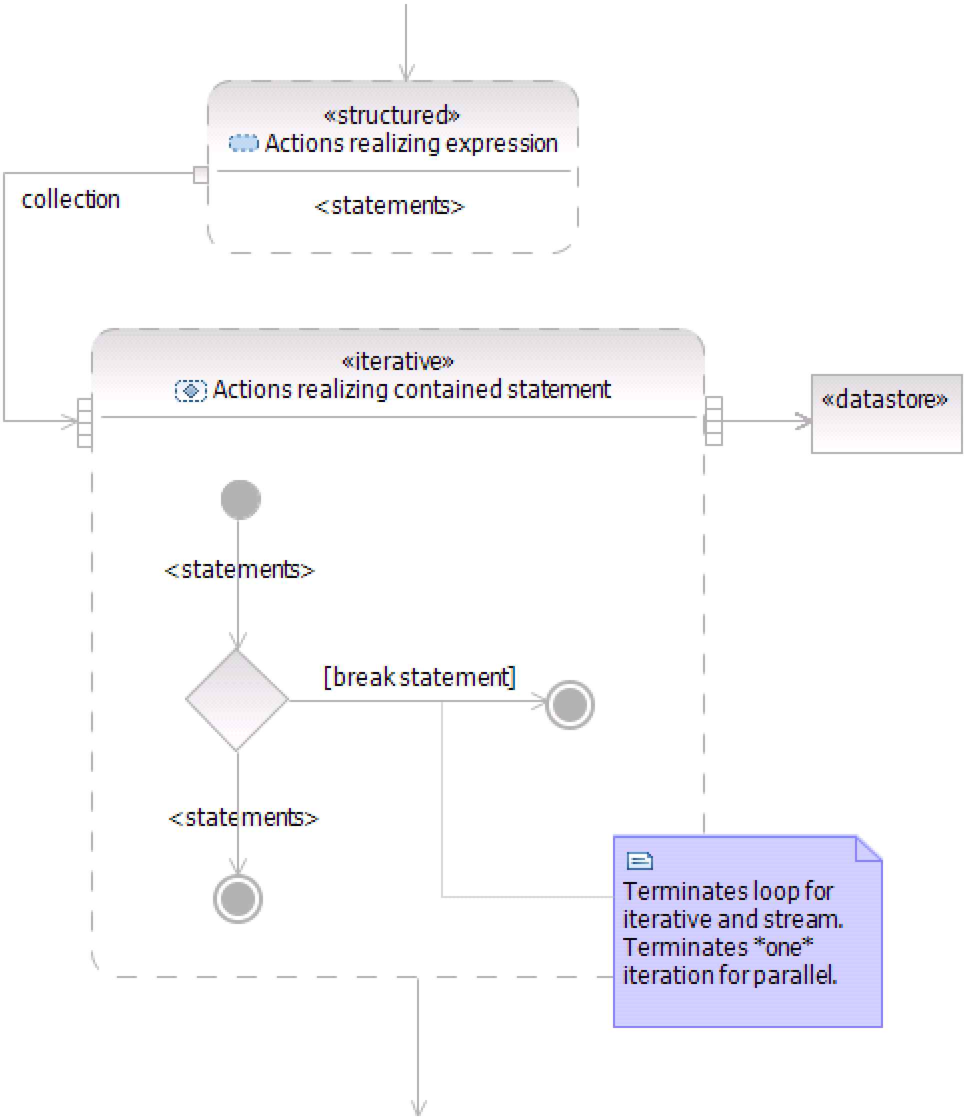
10.8.3 Example

The classic use of break is to terminate a search early without the use of a while loop and a control variable of type Boolean.

```
for (House house : houses) {
   if (house.owner == fred) {
      // do something with fred’s house
      break;  // done
   }
}
```

10.8.4 Mapping

The following maps the for each / expansion region processing for the break statement. The mapping is the same in all blocks, with direct transfer of control to an activity final node.


«structured»
Actions realizing expression
<statements>
collection
«iterative»
Actions realizing contained statement
«datastore»
<statements>
[break statement]
<statements>
Terminates loop for iterative and stream. Terminates *one* iteration for parallel.

10.9 Continue

A continue statement immediately terminates processing of the current loop iteration and transfers control to the end of the enclosing loop. The iteration then completes normally and the loop continues with the next iteration.

10.9.1 Notation

ContinueStatement:

```
        continue;
```

10.9.2 Semantics

No further processing will take place in the current iteration of the while, do, for or for each statement after the continue statement is executed.

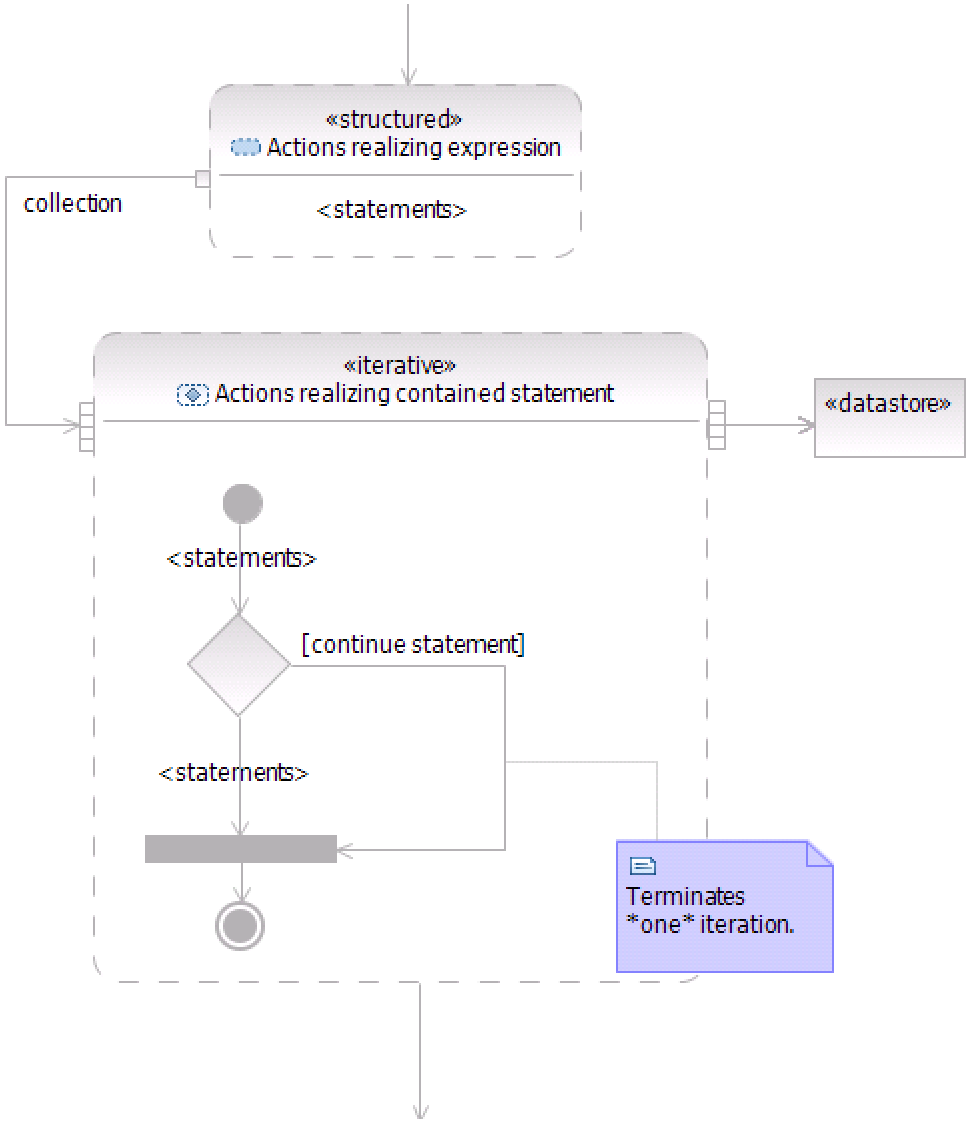
For parallel loops (i.e. expansion regions), the continue statement and break statement provide the same behavior, that of terminating only the current iteration.

10.9.3 Example

The continue statement always executes after a choice, as it hides any code that appears in-line after it. It can be used to provide selective processing, or skip the rest of the block when a specific condition occurs. It is often used simply to avoid deep nesting of if-then-else statements to improve readability.

```
for (Color c : colors) {
   if (c == Color.BLUE) continue; // skip BLUE
   // process all other colors
}
```

10.9.4 Mapping



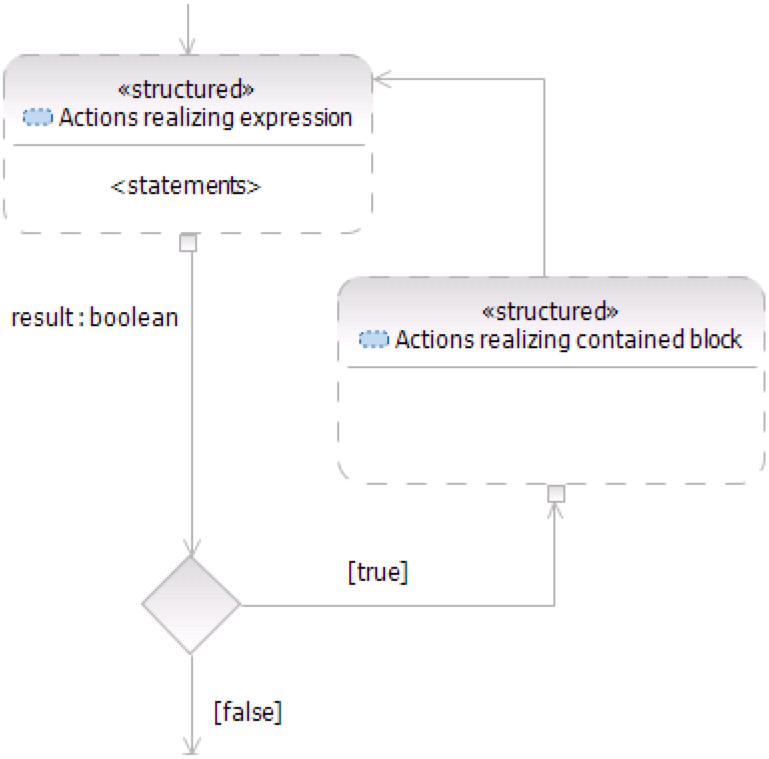
10.10 While

The while statement executes an *Expression* and a *Statement* repeatedly until the value of the *Expression* is false.

10.10.1 Notation

WhileStatement:
    while (*Expression*) *Statement*
WhileStatementNoShortIf:
    while (*Expression*) *StatementNoShortIf*

10.10.2 Mapping



10.10.3 Semantics

The expression must have type Boolean.

If the expression is false when entering the while statement, its statement is never executed and the while statement completes normally.

Else the statement is executed and the expression is again evaluated. This repeats until the expression evaluates to false.

10.11 Do

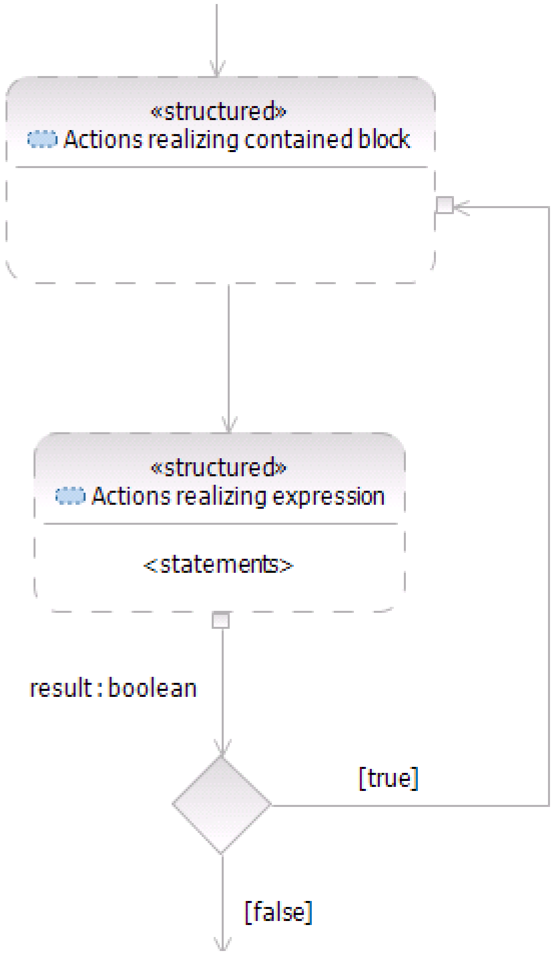
The do statement executes a *statement* and then an *expression* repeatedly until the *expression* evaluates to *false*.

10.11.1 Notation

DoStatement:

    do *Statement* while (*Expression*) ;

10.11.2 Mapping



10.11.3 Semantics

If the statement completes normally, the expression is evaluated. If the expression evaluates to *true*, then the statement is again executed. This continues until the *statement* completes abnormally or the *expression* evaluates to *false.*

When the expression evaluates to *false*, no further action is taken and the do statement completes normally.

10.12 For

The for statement executes some initialization code, then executes an *Expression*, a *Statement*, and some update code repeatedly until the value of the *Expression* is false.

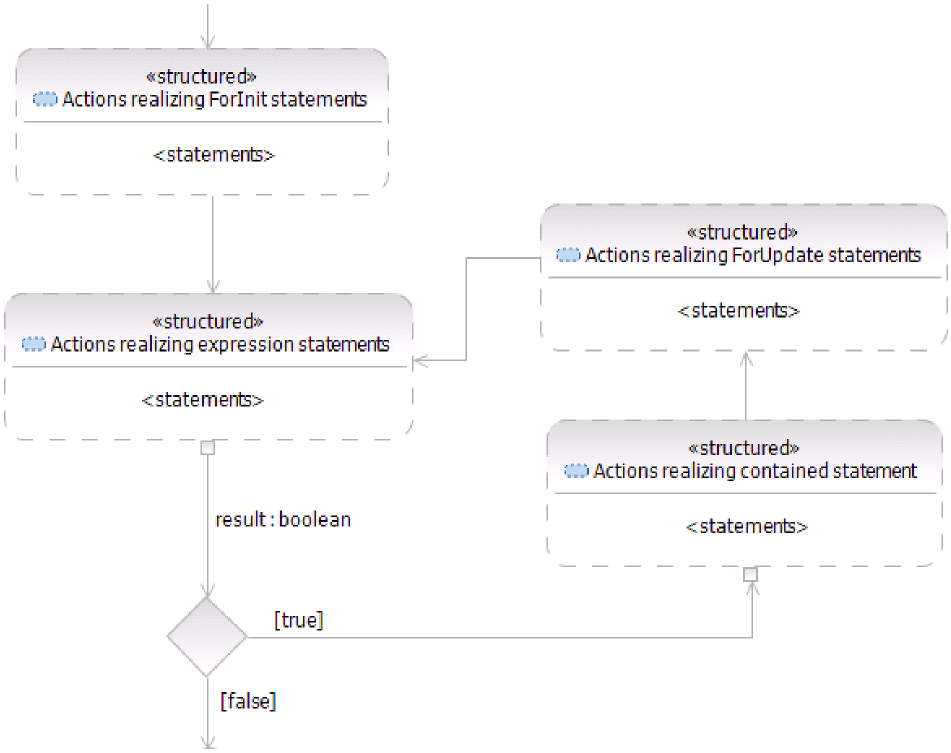
10.12.1 Notation

ForStatement:
  for (*ForInitopt* ; *Expressionopt* ; *ForUpdateopt*) *Statement*
ForStatementNoShortIf:
  for (*ForInitopt* ; *Expressionopt* ; *ForUpdateopt*) *StatementNoShortIf*
ForInit:
  StatementExpressionList
  LocalVariableDeclaration
ForUpdate:
  StatementExpressionList
StatementExpressionList:
  StatementExpression
  StatementExpressionList , *StatementExpression*

10.12.2 Mapping

If the *for* statement defines local variables in the *ForInit* block, then it will map into a structured activity node with a block frame exactly as described in clause 10.3. Inside that mapping diagram is a block within which the contents of the following mapping will appear inside the structured activity node named *Actions realizing block.* For brevity, the local variable declaration mapping is not repeated here.

Otherwise, the mapping to fUML is the same regardless of whether local variables are declared or not.


«structured»
Actions realizing ForInit statements
<statements>
«structured»
Actions realizing ForUpdate statements
<statements>
«structured»
Actions realizing expression statements
<statements>
«structured»
Actions realizing contained statement
<statements>
result : boolean
[true]
[false]

10.12.3 Semantics

The expression must evaluate to a Boolean value.

The *ForInit* code is a list of statement expressions, each of which is evaluated left to right. Their values, if any, are discarded.

If the *ForInit* code is a local variable declaration, its scope is its own initializer, any further declarators to its right, the *expression* and *ForUpdate* parts of the *for* statement, and the contained *statement*.

Iteration is performed as follows:

- If *expression* is present, it is evaluated. If the result is true, then the *statement* is evaluated.
- If the *expression* is not present, then the *statement* is evaluated.
- If *expression* is present and it evaluates to *false*, then no further action is taken and the *for* statement completes normally.
- When *statement* completes normally, and the *ForUpdate* part is present, the statement expressions are evaluated in order from left to right. Their values, if any, are discarded. Then, another iteration is performed.

10.13 For Each

The *For Each* statement provides a mechanism for iterating a collection of items without knowledge of data structure. This is the **recommended** method for processing collections such as links in the UML.

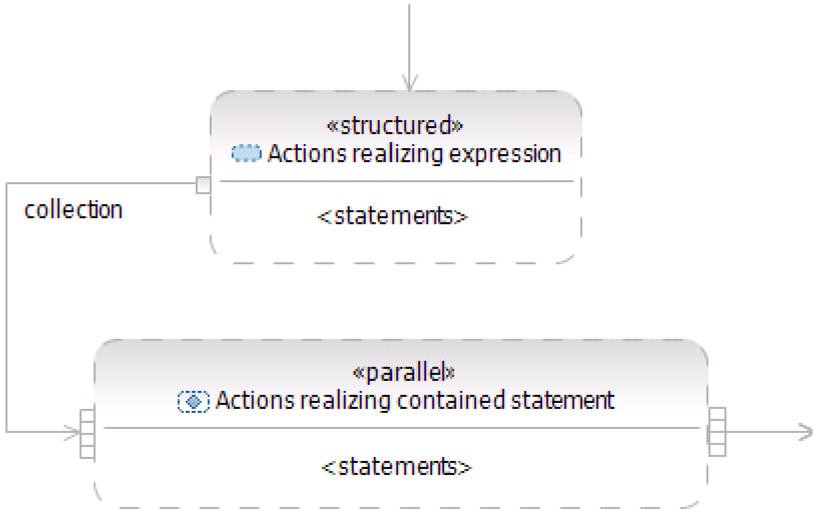
This statement effectively embodies the concept of the expansion region from the UML. In an environment where parallel execution has been defined as the default (the mechanism for which is tooling environment implementation dependent), the expansion region is in fact annotated for parallel operation.

10.13.1 Notation

ForEachStatement:
for (*VariableModifiers*$_{opt}$ *Type Identifier*: *Expression*) *Statement*

10.13.2 Mapping

The *For Each* statement maps directly to an expansion region. The default annotation for an expansion region is *iterative*, however this can be changed using an implementation-specific mechanism, or it can be overridden using a *translator directive* for the individual *for each* statement.


«structured»
Actions realizing expression
<statements>
collection
«parallel»
Actions realizing contained statement
<statements>

The parallel expansion region is shown here, however the expansion region would be stereotyped iterative or stream were the *For Each* statement annotated with a *translator directive* or the tooling environment set to generate those mappings by default.

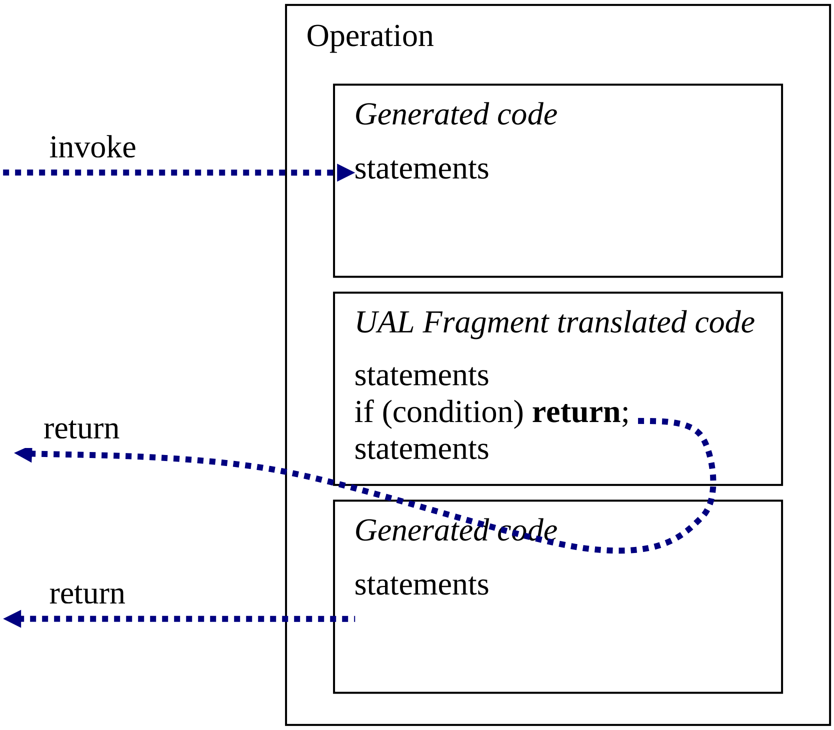
10.13.3 Semantics

The *expression* must evaluate to a collection (i.e. be iterable.)

Each element that exists in the collection represented by the *expression* is assigned to the *identifier* one time, and the *statement* is then executed. The *for each* statement completes when the iterator has no more items.

10.14 Return

The *return* statement exits the containing behavior abruptly. A return leaves the operation regardless of the nesting level of block in which the statement appears.



10.14.1 Notation

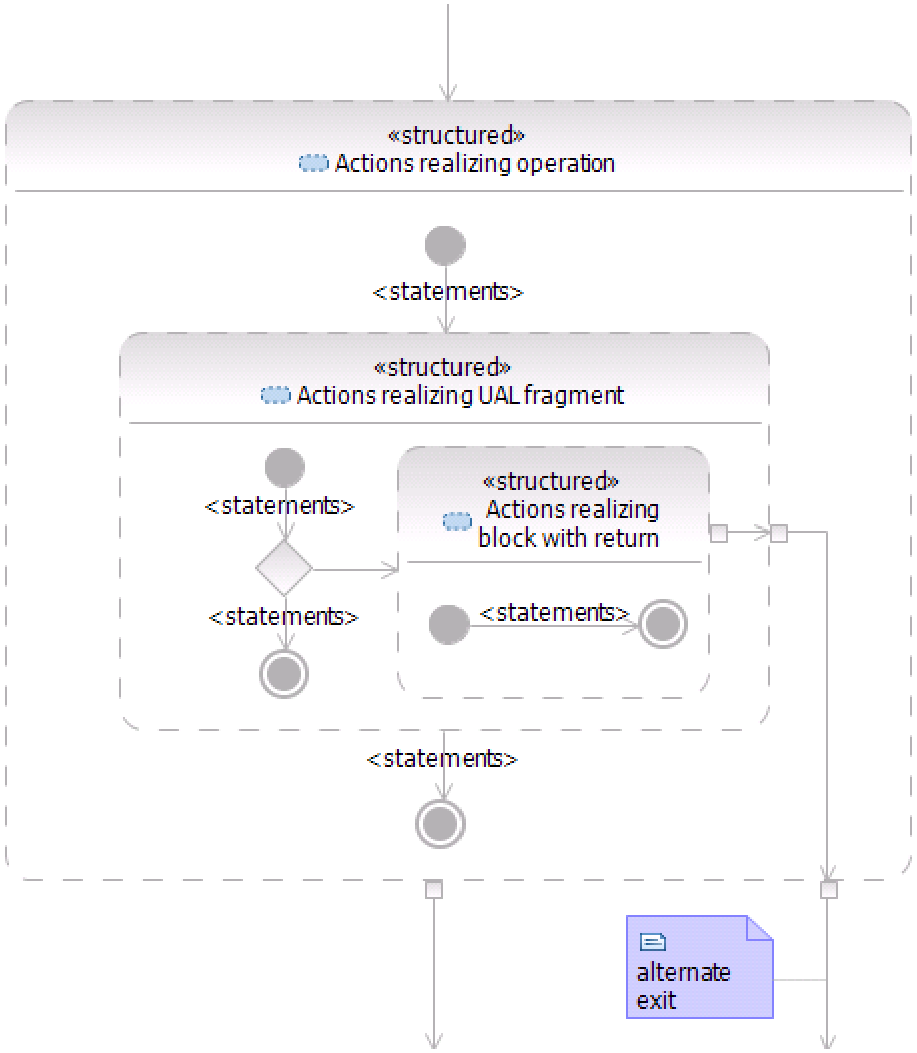
ReturnStatement:

return *Expression*opt ;

10.14.2 Mapping

The return statement and the Exception handlers are examples of the deep nested alternate exit protocol that translators must implement in order to implement this feature for the fUML Virtual Machine. This protocol provides an alternate exit path from any block, passing a token of the same type as the optional *expression* in the statement.

Each Activity and Structured Activity requires an alternate exit path for early returns.



10.14.3 Semantics

The optional expression must be of the same type as the return output parameter for the operation.

When a token appears on the alternate exit path, a null token is placed on the primary exit path.

At the top level activity for the operation's behavior, the two paths will join before existing.

10.15 Exceptions

Exceptions are excluded from the foundational UML. However, exceptions are a first-class concept in popular target languages like Java and C++. Since translators for most existing tools translate directly to the target platform and will continue to do so for the foreseeable future, exceptions are permitted within UAL fragments.

The Exception type is not present in fUML, however there is a Notification type that contains an error code. Negative values are errors and positive values are informational, with zero having the generic meaning *success*.

10.15.1 Semantics

As with the *Return* statement, *Exceptions* are abnormal exits and can exit all blocks in the containing operation, regardless of nesting. Also like the *return* statement, an *exception* uses the alternate exit protocol for code generation.

Unlike the *return* statement, an *exception* can be intercepted and handled at any containing block level. And again unlike the *return* statement, the alternate exit path for *exceptions* return a value of type *Notification* from the fUML library in order to pass the return code.

To raise an exception, the *throw* statement is used in a UAL fragment.

To catch an exception, the *try statement* is used to define a block and one additional *catch* statement is used for each **handled** notification code. Notification codes that do not appear in a *catch* clause automatically propagate to the abnormal exit path of the enclosing block.

A *finally* clause can be used in order to perform processing before exist the overall try block, whether the exception is handled or not.

A *throw* statement appearing inside a catch block will cause the exception handler to be reentered.

10.15.2 Notation

ThrowStatement:
        throw *Expression* ;

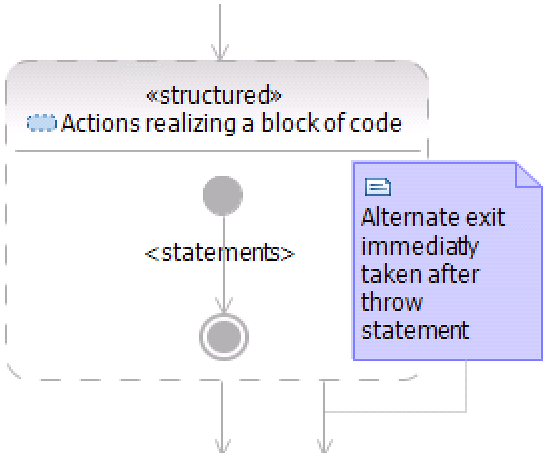
TryStatement:
    try *Block Catches*
    try *Block Catches*$_{opt}$ *Finally*
Catches:
    CatchClause
    Catches CatchClause
CatchClause:
    catch (*FormalParameter*) *Block*
Finally:
    finally *Block*

FormalParameter:
    VariableModifiers Type VariableDeclaratorId

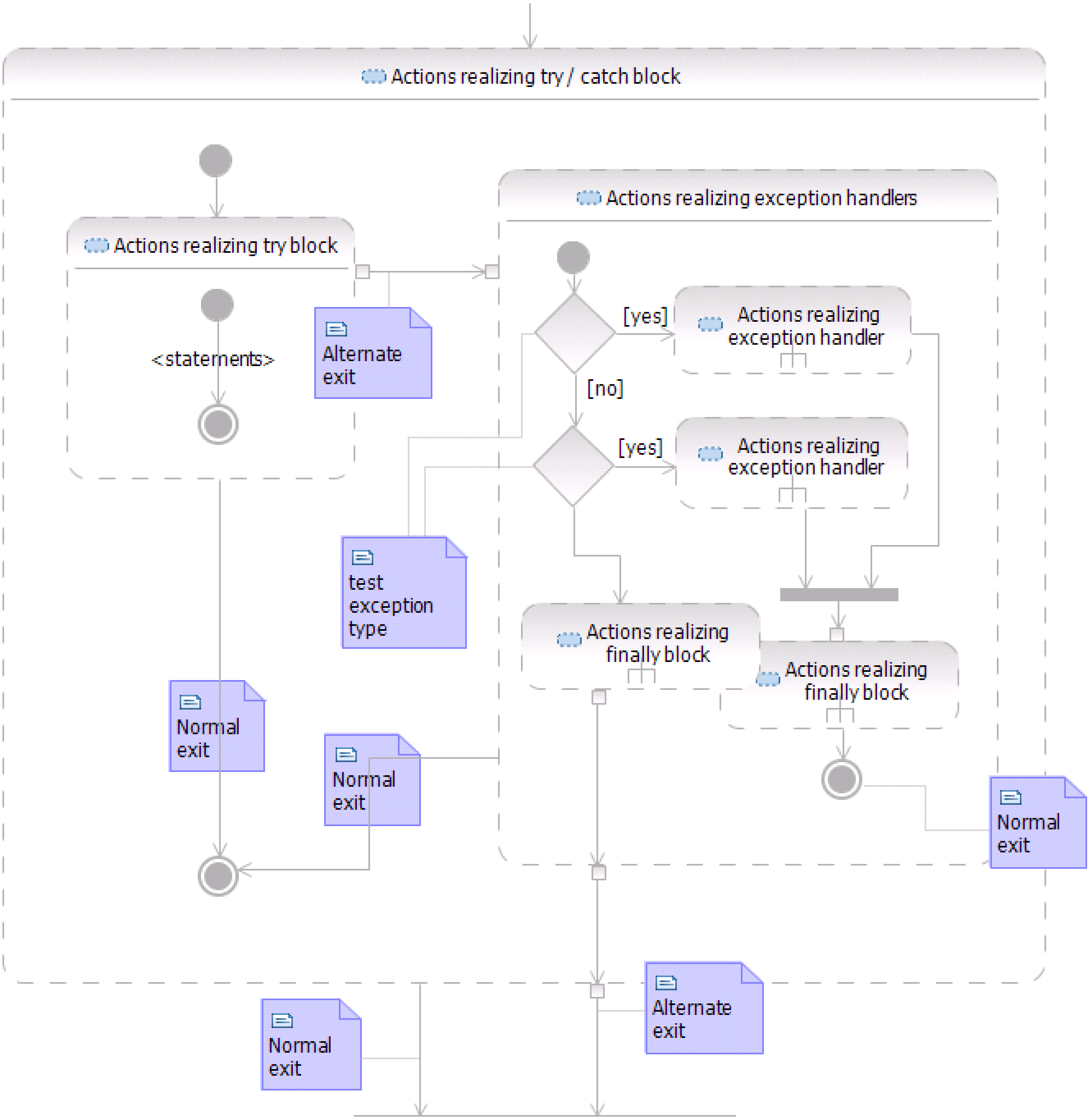
VariableDeclaratorId:
    Identifier
    VariableDeclaratorId [ ]

10.15.3 Mapping

10.15.3.1 Throw


«structured»
Actions realizing a block of code
<statements>
Alternate exit immediatly taken after throw statement

10.15.3.2 Try / Catch


Actions realizing try / catch block
Actions realizing try block
<statements>
Alternate exit
Actions realizing exception handlers
[yes]
Actions realizing exception handler
[no]
[yes]
Actions realizing exception handler
test exception type
Actions realizing finally block
Actions realizing finally block
Normal exit
Normal exit
Normal exit
Normal exit
Alternate exit

11 Actions

The UAL surface language provides syntax for basic, intermediate and complete actions. This layering is taken from the UML Superstructure [2] and is further refined in the UML Foundation [1] in Clause 8.6.

11.1 Basic Actions

The UML Foundation BasicActions package includes the basic specification for actions and pins, plus the invocation actions: call behavior action, call operation action and send signal action (see [1], Subclause 7.5.2).

This subclause includes corresponding specifications for action expressions and invocation action expressions. It also defines the syntax for library primitive behaviors.

11.1.1 Operators

Operator syntax has a Java legacy. It can be used for certain of the primitive behaviors from the Foundational Model Library (see Foundational UML Specification [1], Clause 9) and for testing for equality and inequality. This provides the usual prefix and infix notations for arithmetic, Boolean and relational operators.

The following sections map directly to the Foundation UML Model Library.

11.1.1.1 Primitive Types

Type Name	Description in the Context of UAL Fragments
Boolean *boolean*	fUML’s Boolean is represented by a primitive type *boolean,* with the two values true and false.
Integer *int*	fUML’s Integer is represented by a primitive type int, with a range defined by the target implementation. If a translator maps to Java’s *int*, the range is -2,147,483,648 to 2,147,482,647. If mapped to Java’s *long*, the the range is -9,223,372,036,854,775,808 to 9,223,372,036,854,775,807. And so on.
String	fUML’s String is represented by a class String in UAL. String provides a standard set of methods and the special concatenation operator “+”.
UnlimitedNatural	fUML’s UnlimitedNatural is represented by a primitive type long, with a restricted range to positive integers only. Translators shall include range checking to ensure that values cannot go negative.
List	[1] does not include the List type in its summary in clause 9.1, however it does include a section for List in the primitive behaviors. UAL represents List as a generic collection class. UAL also provides a full set of collection classes in clause ??? that provide for tuning at the UAL level.

11.1.1.2 Notation

11.1.1.2.1 Boolean

Function Signature in Foundational UML	UAL Expression
Or(x : Boolean, y : Boolean) : Boolean	x || y
Xor(x : Boolean, y : Boolean) : Boolean	x ^ y
And(x : Boolean, y : Boolean) : Boolean	x && y
Not(x : Boolean) : Boolean	!x
Implies(x : Boolean, y : Boolean) : Boolean	FUML.Implies(x,y)
ToString(x : Boolean) : String	x.toString()
ToBoolean(x : String) : Boolean[0..1]	Boolean.parseBoolean(x) can return empty set (null) if no Boolean expression found

11.1.1.2.2 Integer

Function Signature in Foundational UML	UAL Expression
Neg(x: Integer): Integer	-x
+(x: Integer, y: Integer): Integer	x + y
-(x: Integer, y: Integer): Integer	x - y
*(x: Integer, y: Integer): Integer	x * y
Abs(x: Integer): Integer	FUML.abs(x)
Div(x: Integer, y: Integer): Integer	Y / y
Mod(x: Integer): Integer	x % y
Max(x: Integer, y: Integer): Integer	FUML.max(x,y)
Min(x: Integer, y: Integer): Integer	FUML.min(x,y)
<(x: Integer, y: Integer): Integer	x < y
>(x: Integer, y: Integer): Integer	x > y
<=(x: Integer, y: Integer): Integer	x <= y
>=(x: Integer, y: Integer): Integer	x >= y
ToString(x: Integer): Integer	x.toString()
ToUnlimitedNatural(x: Integer): UnlimitedNatural[0..1]	UnlimitedNatural.valueOf(x)
ToInteger(x: Integer): Integer[0..1]	Integer.valueOf(x) can return empty set (null) if no Integer expression found
The following entries are proposed additions to the Foundational UML	
And(x: Integer, y: Integer): Integer	x & y
Or(x: Integer, y: Integer): Integer	x | y
XOr(x: Integer, y: Integer): Integer	x ^ y
Complement(x: Integer): Integer	~x

Increment(x: Integer): Integer	++x, x++
Decrement(x: Integer): Integer	--x, x--
Lhift(x: Integer, d: Integer): Integer	x << d
RShift(x: Integer, d: Integer): Integer	x >> d
RUShift(x: Integer, d: Integer): Integer	x >>> d

11.1.1.2.3 String

Function Signature in Foundational UML	UAL Expression
Concat(x: String, y: String): String	x + y
Size(x: String): Integer	x.length()

SubString(x: String, lower: Integer, upper: Integer): String	x.substring(lower, upper)
	x.substring(lower)

The following entries are proposed additions to the Foundational UML	
Concat(x: String, y: Integer): String	x + y
Concat(x: String, y: Boolean): String	x + y
Concat(x: String, y: UnlimitedNatural): String	x + y

11.1.1.2.4 UnlimitedNatural

Function Signature in Foundational UML	UAL Expression
Max(x: UnlimitedNatural, y: UnlimitedNatural): UnlimitedNatural	FUML.max(x,y)
Min(x: UnlimitedNatural, y: UnlimitedNatural): UnlimitedNatural	FUML.min(x,y)
<(x: UnlimitedNatural, y: UnlimitedNatural): Boolean	x < y
>(x: UnlimitedNatural, y: UnlimitedNatural): Boolean	x > y
<=(x: UnlimitedNatural, y: UnlimitedNatural): Boolean	x <= y
>=(x: UnlimitedNatural, y: UnlimitedNatural): Boolean	x >= y
ToString(x: UnlimitedNatural): String	x.toString()
ToInteger(x: UnlimitedNatural): Integer[0..1]	x.intValue() returns primitive *int*
ToUnlimitedNatural(x: String): UnlimitedNatural[0..1]	x.decode() returns primitive *long*

11.1.1.2.5 List

Function Signature in Foundational UML	UAL Expression
ListSize(list[*]): Integer	list.size()
ListGet(list[*]{ordered}, index: Integer) [0..1]	list.get(index)

11.1.1.3 Examples

11.1.1.3.1 Arithmetic

```
amount * interestRate
initialPosition + velocity * time
-abs(energy)
duration / timeStep
length % unit
```

11.1.1.3.2 Relational

```
sensorReading > threshold
count <= limit
recordLimit == maximum
currentTime != endTime
```

11.1.1.3.3 Boolean

```
!(sensorOff || sensorError)
i > min && i < max || !limited
implies(p ^ q, !(p and q))
```

11.1.1.3.4 String

```
firstName + " " + lastName
buffer.length()
```

11.1.1.3.5 List

```
this.getTypes.get(1)
houses.size() < 2
records.size() != 0
records.isEmpty()
```

11.1.1.4 Semantics

The `&` and `|` operators always evaluate both expressions while the `&&` and `||` operators are short circuit expressions, evaluating only the first expression if the result is already determined. For example, with the expression `(true | a < 100)` the second expression is never evaluated. This matters when Boolean expressions appear on each side and it is necessary to ensure that both expressions are evaluated every time.

11.1.1.4.1 Disambiguation

When binary operators appear between multiple operands without ellipses for disambiguation, the following operator precedence (Table 10-1) shall be used to group the sub-expressions appropriately.

Table 11-1 Operator Precedence (highest to lowest)

Operators	Precedence
`Postfix`	`expr++ expr--`
`Unary`	`++expr –expre +expr –expr - !`
`Multiplicative`	`* / %`
`Additive`	`+ -`

Operators	Precedence
Shift	`<< >> >>>`
relational	`< > <= >= instanceof`
equality	`== !=`
Bitwise AND	`&`
Bitwise exclusive OR	`^`
Bitwise inclusive OR	`|`
Logical AND	`&&`
Logical OR	`||`
Ternary	`? :`

Assignment	`= += -= *= /= %= &= ^=`
	`|= <<= >>= >>>=`

For example:

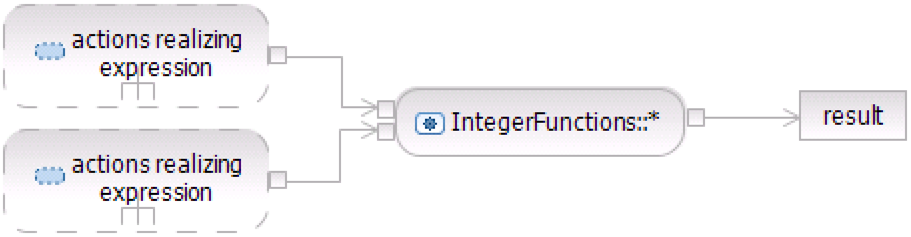
```
        bool = a + b * c <= l / m + n++ || b == j;
```

is disambiguated as:

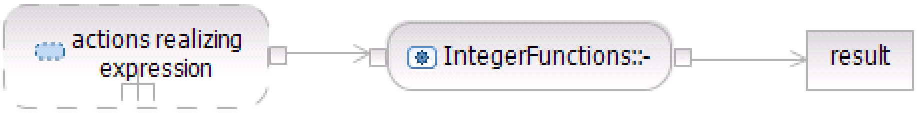
```
        bool = ((a + (b * c)) <= ((l / m) + (n++))) || (b == j);
```

11.1.1.5 Mapping

An operator expression maps to a call operation action. Mapping to fUML is straightforward, an example for binary operators being:

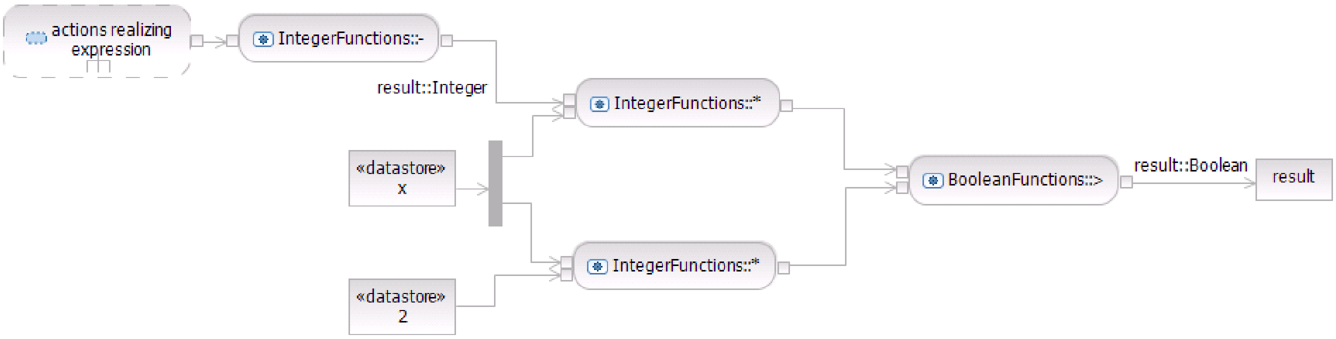


A unary operator would map thusly:



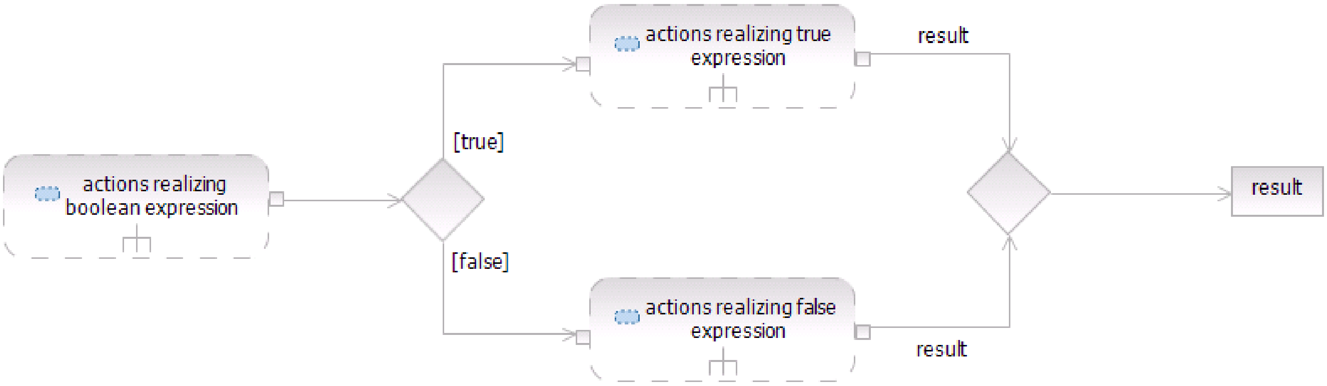
Complex expressions will map to a cascade of such expressions, connecting output pins to input pins of the next operation in the expression. An expression must return a value so that error checking is not required in a fUML expression sequence.

Map the example expression x*2 > -y*x



The ternary operator is mapped using this expression:

```
    <boolean expr> ? <true expr> : <false expr>;
```


actions realizing true expression
result
[true]
actions realizing boolean expression
result
[false]
actions realizing false expression
result

On mapping of prefix versus postfix operators. The variable or attribute upon which the operator operates will flow into a form node and be used from there. With a prefix operator, the operation occurs before the fork and all downstream actions see the new value. With a postfix operator, the operation occurs after the fork and all downstream actions see the original value. There is no difference in the actual operation mapped in fUML, since that is simply the addition of 1.

11.1.2 Invocation Actions

Invocation actions include the *Call Operation Action*, the *Call Behavior Action*, and the *Send Signal Action*. The *Broadcast Signal Action* is excluded from the foundational UML. Invocation Actions are always synchronous in fUML (see [1], clause 7.5.2.)

11.1.2.1 Notation

ArgumentList:
    Argument
    ArgumentList , *Argument*

MethodInvocation:
    MethodName (*ArgumentListopt*)
    Primary . *NonWildTypeArgumentsopt Identifier* (*ArgumentListopt*)
    super . *NonWildTypeArgumentsopt Identifier* (*ArgumentListopt*)
    ClassName . super . *NonWildTypeArgumentsopt Identifier* (*ArgumentListopt*)
    TypeName . *NonWildTypeArguments Identifier* (*ArgumentListopt*)

11.1.2.2 Call Operation and Call Behavior Actions

In [1], the two call actions are described as being identical in behavior, but not in how they are instantiated.

> *Other than for how this execution object is instantiated, the semantics of call behavior and call operation actions are the same: values of argument input pin activations are passed as input parameter values to the execution object, the execute operation is called on the execution and then any output parameter values are placed on result output pin activations. Once execution is complete, the execution object is destroyed.*

As instantiation, marshalling of arguments and polymorphic dispatching are properties of the fUML runtime, only the execution of the call operation is mapped to fUML as it is effectively identical to the call behavior at that point.

UAL uses the Java method call syntax for invocation action expressions. Arguments to the invocation action appear in sequence and are matched to parameters in the same sequence in the behavior's signature.

So, for a behavior B with input parameters x and y, `B` is invoked with input values for its parameters:

```
B(1, 2)
```

For a parameter with multiplicity lower bound of zero (0), "null" can be used as an argument to invocation. The translator shall infer a missing parameter from "null."

```
B(1, null)
```

The result output parameter is connected to the output target for the invocation, which can be an assignment to a variable or attribute or an argument to another expression.

An assignment statement for an attribute can therefore use the result of B:

```
this.A = B(1, null);
```

Or B can be used in an expression or as the input to another invocation:

```
C(B(1, null), 2, 3)
if (a == B(1, null)) {//do something}
```

Superclass

The familiar **`"super.op"`** notation is provided to invoke an inherited operation as defined in the immediate superclass instead of the local context object's class. The super class implementation may be owned or inherited.

Disambiguation of conflicting inherited methods uses the redefinition mechanism since the conflicting operations are otherwise invisible in the context object.

11.1.2.2.1 Examples

```
computeInterest(amount)
group.activate(nodes, edges)
actuator.Initialize(systemMonitor)
super.run()
```

11.1.2.2.2 Semantics

All call action invocations are synchronous as per [1].

The send signal action invocation is asynchronous, as per [1]. It adds the signal to the target object's event queue, said queue being dispatched to the target object according to [1] and complying with local semantic variations.

The invoked behavior name must be visible to the UAL fragment.

When more than one behavior is visible with a matching name, the behavior with a matching signature is invoked.

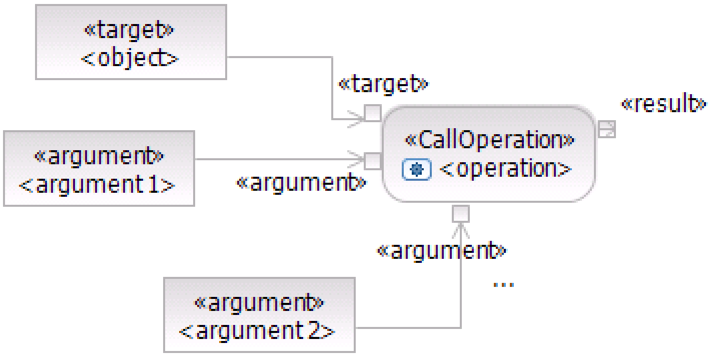
Multiple behaviors with the same name and same signature shall be a translator error.

If no behavior with the same name and matching signature is visible, a translator error occurs.

11.1.2.2.3 Mapping

An invocation action maps to a call operation action for the operation specified.

```
<object>.<operation>(<argument 1>, argument 2, …)
```



11.1.2.3 Send Signal Action

As specified in [2], a Send Signal Action constructs an instance of a Signal, and transmits it to a target object. The target object may be an object reference or it may be a port reference, to which a target object is connected.

In the fUML virtual machine, the signal instance is actually sent by calling the *sendSignal* operation on the target object (see [1] clause 8.5.3)

The sender receives no reply, as per clause 11.3.45 of [2].

An alternate syntax allows the sending of an instance of a previously constructed Signal (forwarding pattern.)

11.1.2.3.1 Message Direction

Experience with protocols in some real time tools has led to a distinction between sending signals outward through a port and sending signals inward through a port. The former use case would see the signal forwarded across another connection to a target while the latter would be delegated to the owning classifier or behaviour.

This has evolved to using “`send`” to denote an outward link traversal through a port and “`receive`” to denote injection directly into a port or object.

To understand the distinction, one must consider a pair of connected ports. Suppose `Alice` and `Bob` own ports `a` and `b` respectively but each has access to both ports. `Bob` can therefore send a signal to `Alice` in two ways:

```
b.Hello(“This is Bob”).send();
a.Hello(“This is Bob”).receive();
```

The difference between these two amounts to a link traversal in the former case and a direct injection in the latter case.

11.1.2.3.2 Multiplicity

Ports and objects can have multiplicity, which implies instances of the (owning) classifier. In such a case, it is necessary to provide for the selection of a specific instance without making the syntax cumbersome when there is only one instance.

When sending to a specific instance of a port, the only syntax change is to use sendAt with an index rather than send. When multiplicity is `[1..1]` then the send all syntax `“send()”` has exactly the same effect as `“sendAt(0)”`. Similar provisions apply to the `reply` and `invoke` variants.

The `sendAt` form is immune to target multiplicity changes and maps exactly as if it had been written as: `<target>.get(<index>).send()`. But the latter form requires the generation of multiple types when `“get”` is not needed, i.e. when multiplicity is `[1..1]`.

11.1.2.3.3 Synchronicity

Protocols sometimes require synchronous messaging, facilities for which have been excluded from the foundational UML. The action language provides a way to differentiate between synchronous and asynchronous messaging with different mappings. Asynchronous messaging is denoted by the `“send”` syntax while synchronous messaging is denoted by the `“invoke”` syntax.

11.1.2.3.4 Notation

There are four notations for each operation:

- construct a signal and send to all target instances;
- construct a signal and send to one target instance;
- send an existing signal to all target instances; and
- send an existing signal to one target instance.

Send a signal outward through a port:

```
<target>.<signalClassName>(<arguments>).send();
<target>.<signalClassName>(<arguments>).sendAt(<index>);
<target>.<signalClassName>(<signal>).send();
<target>.<signalClassName>(<signal>).sendAt(<index>);
```

To inject a signal into a target:

```
<target>.<signalClassName>(<arguments>).receive();
<target>.<signalClassName>(<arguments>).receiveAt(<index>);
<target>.<signalClassName>(<signal>).receive();
<target>.<signalClassName>(<signal>).receiveAt(<index>);
```

Send a signal outward through a port and wait for a response before continuing:

```
<target>.<signalClassName>(<arguments>).invoke();
<target>.<signalClassName>(<arguments>).invokeAt(<index>);
<target>.<signalClassName>(<signal>).invoke();
<target>.<signalClassName>(<signal>).invokeAt(<index>);
```

In order to write self-documenting code that is immune to protocol changes between synchronous and asynchronous communication, a reply command is introduced to map to the necessary algorithm to turn a signal around and send it back to the sender. Since it is always a response to a specific event being handled, there is no "`At`" form for the reply operations.

```
<target>.<signalClassName>(<arguments>).reply();
<target>.<signalClassName>(<signal>).reply();
```

11.1.2.3.5 Disambiguation

The notation for sending an existing signal could be shortened from:

```
<target>.<signalClassName>(<signal>).send();
```

to:

```
<target>.send(<signal>);
```

But this notation has a problem with specialized signals in this case where a protocol with (at least) three outgoing signals: B, E1 and E2 where E1 and E2 both extend B and a port p that uses that protocol.

```
B s1 = new B(...);
B s2 = new E1(...);
B s3 = new E2(...);
p.send(s1);
p.send(s2);
p.send(s3);
```

The disambiguation that is inherent in the more verbose form is difficult in this form, since s1, s2 and s3 all have the same type.

11.1.2.3.6 Examples

```
// send
fridge.doorControl.Opened().receive();
fridge.tempControl.Reading(temperature).receive();

// inject
fridge.controlsPort.DoorOpened().send();
fridge.controlsPort.TemperatureReading(temperature).send();

// send existing signal
fridge.controlsPort.DoorStateChange(currentstate).send();
fridge.tempControl.TemperatureReading(currenttemp).send();

// send to specific instance
fridge.controlsPort.LightOn().send(freezerLightIndex);
fridge.lights.LightStateChange(currLightState).receive(freezerLightIndex);

// send directly to the object instance
fridge.DoorOpened().receive();
fridge.TemperatureReading(currenttemp).receive();
```

11.1.2.3.7 Semantics

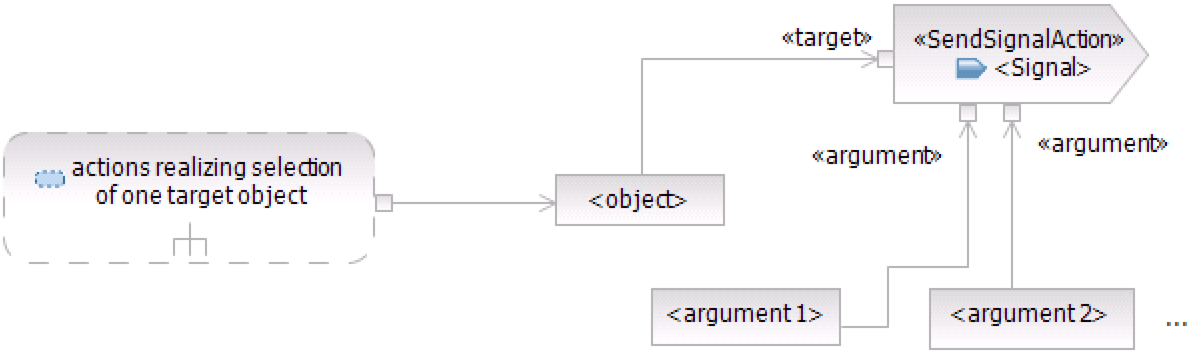
The target object must have a reception defined for the Signal type.

The Signal must have a defined default constructor and all arguments must be physically present. For optional parameters (i.e. multiplicity lower bound is 0), the argument may be the `null` literal to denote a missing parameter.

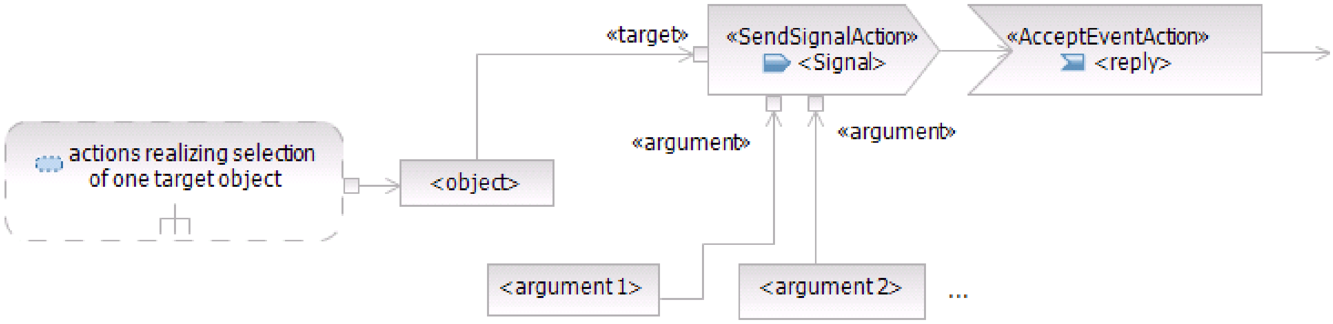
Multiple default constructors with the same number of parameters with `null` arguments in some positions may defeat disambiguation. In this case, the translator shall flag an error.

11.1.2.3.8 Mapping

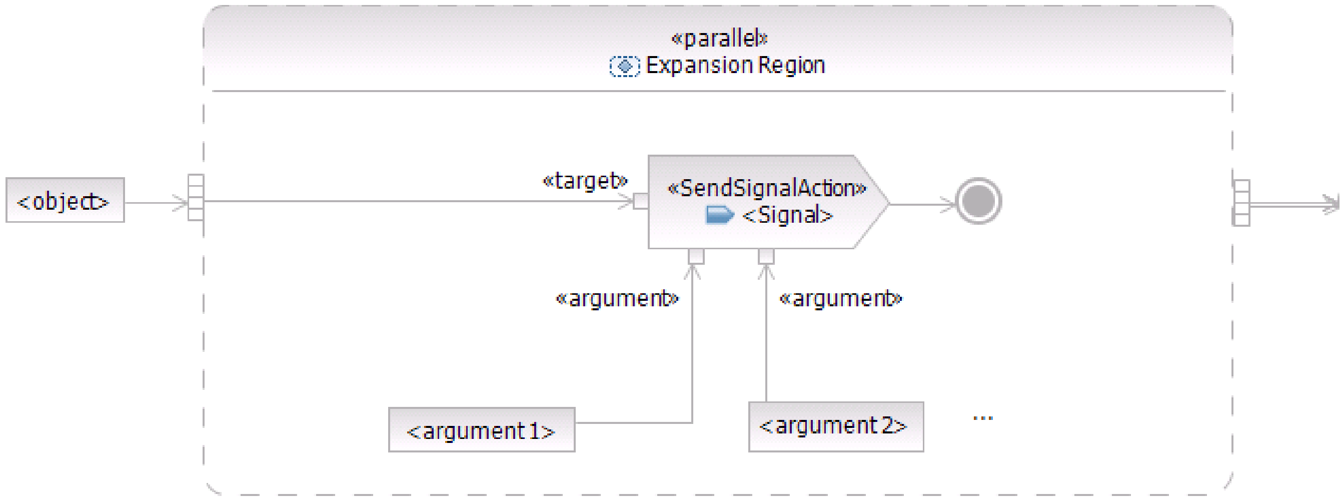
SendAt, receiveAt and replyAt



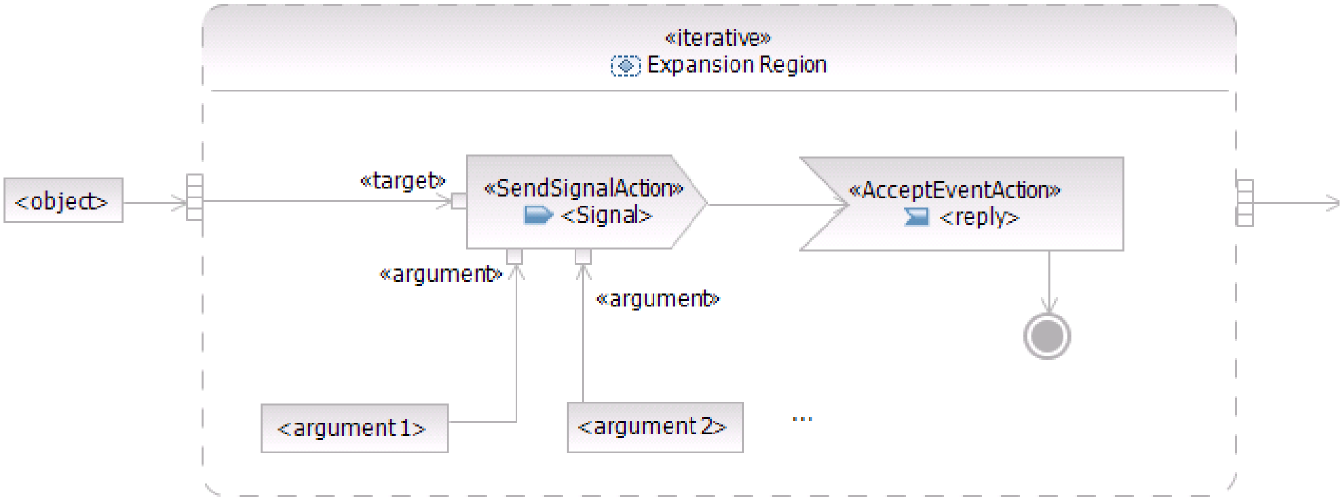
InvokeAt



Send, receive and reply

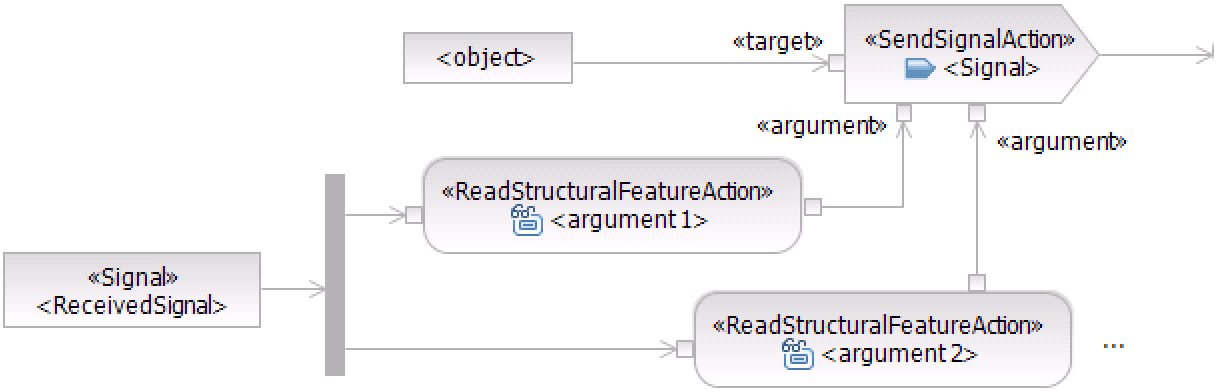


Invoke



Send Existing Instance

fUML excludes the Send Object Action, therefore the mapping for sending an existing Signal instance maps to fUML by reading the arguments connect the results to the input pins of the Send Signal Action. This pattern applies to all Send Signal Action mappings.



11.1.3 Receive Signal Action (Transition)

This action exists for State Machine transitions and is covered here for completeness of translation. As mentioned earlier, the state chart itself will be translated to an Activity for an FVM. However, the UAL fragments are subject to the same standardized translation as any UAL fragment and so appear in this specification.

11.1.3.1 Transition

A Transition or Receive Signal Action specifies the reception of a signal, which may trigger execution of actions.

In a state-oriented state chart diagram, a transition line is used to denote the entire transition, while in a transition-oriented state chart diagram a Receive Signal Action symbol is used to denote the Receive Signal Action specifically.

A transition line shall according to the UML standard support the following syntax:

```
<triggers> [ <guard expression> ] / <actions>
```

Here `<triggers>` is in the simple case just a reference to a signal, but in general more than one signal can be referenced.

Also, for each such reference to a signal having formal parameters it is possible to declare assignments of actual signal parameters to attributes visible from the scope of the transition. It is also possible to define variables in-place; that is, variables that are local to that particular transition. The latter possibility is only useful when exactly one signal is referenced in the `<triggers>` section.

A special case is to specify that any signal can trigger a transition. This is denoted in the syntax by using a '*' as trigger. In that case it is not possible to declare assignments of signal parameters; that has to be done using UAL code instead.

Examples of this syntax include:

```
mySignal / myContext.port1.send(mySignal); // forwarding pattern
SigWithPar (classAtt) [classAtt > 0] / classAtt++;
* / System.out.println ("An unexpected signal was received!");
```

Since there are three separate blocks, the UAL fragment parser will split these sections apart and process them independently.

The `<triggers>` will map to one or more signal receptions.

The `[guard]` will map to a Boolean expression or expression statement, see section 10.5 Expression Statements and section 11.1.1 Operators for mappings.

And the `<actions>` body will map to a code block, expressed as a structured activity. See section 10 Blocks and Statements for mappings.

11.1.3.2 Timer Events

A special timer trigger syntax exists in the UML for Time Events.

A transition may be triggered by a TimeEvent specifying either an absolute point in time, or a time relative to the point in time when the source state was entered. The UML syntax uses the keywords 'after' and 'at' to denote relative and absolute time values respectively. However, the exact syntax of the time expression itself is not standardized, so UAL uses the Java 1.5 conformant expressions that are accepted by the Timer.set() methods.

Examples of this syntax would be:

```
after 5 / foo; // Trigger transition after 5 time units (5s by default)
at (new GregorianCalender(2009, 1, 1)).getTime() /; // Trigger Jan 1 2009
```

Additional mappings for time events include an activity containing actions to set the appropriate timers and a signal reception to catch the event.

11.1.3.3 Call Events

Specifying an operation name instead of a signal name in the triggers section indicates a desire to receive an event whenever that operation is called from another object. The call event is generated after the operation has finished processing the invocation.

Again, the mapping includes actions to set the event trigger and a signal reception for the call event.

11.2 Intermediate Actions

The IntermediateActions package includes the majority of fUML's allowable actions. Of the actions defined in the UML 2 IntermediateActions package, only the broadcast signal and send object actions are excluded in fUML (see [1], clause 7.5.3).

11.2.1 Create Object Action

A create object action instantiates a classifier and yields an object instance that is typed by that Classifier.

A UML constructor is an operation with the stereotype <<create>> that returns an instance of the owning class. The rules for the order in which class attribute initializations are evaluated compared to constructor invocation are the same in UAL.

The foundational UML does not provide initialization during creation of an object; however this surface syntax does provide the familiar constructor syntax. Simultaneous creation and initialization are allowed in surface languages according to the UML Superstructure [2], clause 11.1:

"The specification defines the create action to only create the object, and requires further actions to initialize attribute values and create objects for mandatory associations. A surface language could choose to define a creation operation with initialization as a single unit as a shorthand for several

actions."

Thus, if initial attributes are specified, a fUML translator will generate a *coordinated set of actions* to provide the appropriate effect at execution time.

The create object action can only be used with a class in fUML.

11.2.1.1 Notation

ClassInstanceCreationExpression:
    new *TypeArguments*_{*opt*} *ClassOrInterfaceType* (*ArgumentList*_{*opt*})
ArgumentList:
    Expression
    ArgumentList , *Expression*

11.2.1.2 Example

```
new ActivityNodeActivationGroup()
new Person(name, address, SSN)
```

11.2.1.3 Semantics

A class name must be a visible non-local name that resolves to a class.

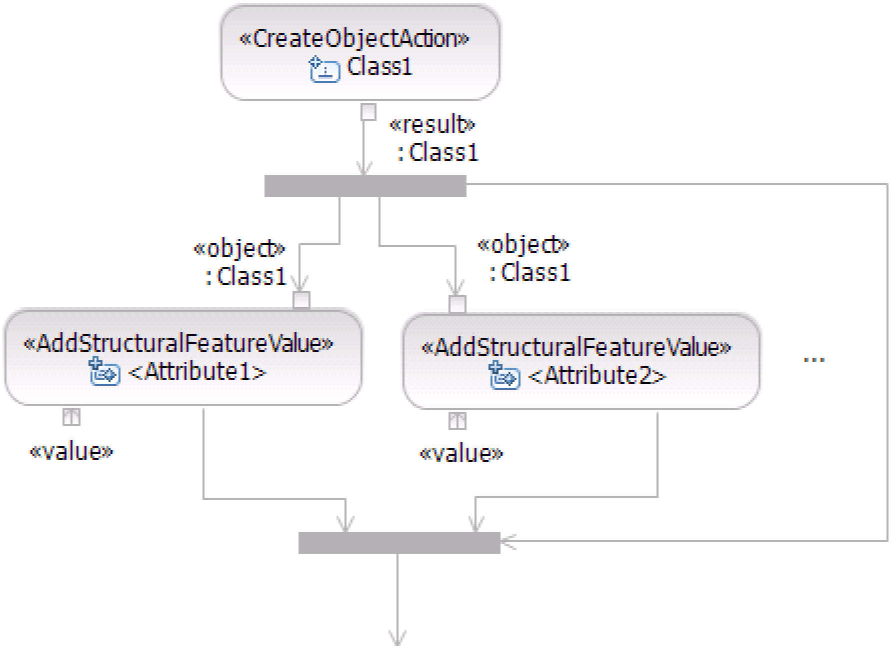
UAL requires that the constructor operation has the same name as the classifier. **(True?)**

11.2.1.4 Mapping

A create object action expression maps to a create object action for the class with the given name.

A coordinated set of actions is generated by translators when there are arguments to the constructor form and / or mandatory associations exist.

```
Class1 c1 = new Class1(Attribute1, Attribute2, …);
```



11.2.2 Destroy Object Action

A Destroy Object Action destroys an object explicitly. Although not explicitly stated in [1], it is stipulated by UAL that the classifier behavior of a destroyed object stops executing immediately. Allowing a classifier behavior to continue executing after the termination of its context object shall have undefined behaviour and target platforms should guard against this scenario.

11.2.2.1 Notation

Method invocation notation with no arguments, see clause 11.1.2.1.

```
<object>.destroy();
<object>.destroy(boolean DestroyLinks, Boolean DestroyOwnedObjects);
```

Alternate form has two Boolean parameters – DestroyLinks, DestroyOwnedObjects. These set the action's internal isDestroyLinks and isDestroyOwnedObjects features before initiating destruction of the object.

11.2.2.2 Examples

```
sensor.destroy();
anExecutingObject.destroy(); // also stops execution
sensor.destroy(true, false);
person.destroy(true, true);
```

11.2.2.3 Semantics

Invocation signature 1 accepts no arguments. Default behavior is to destroy owned objects and all links.

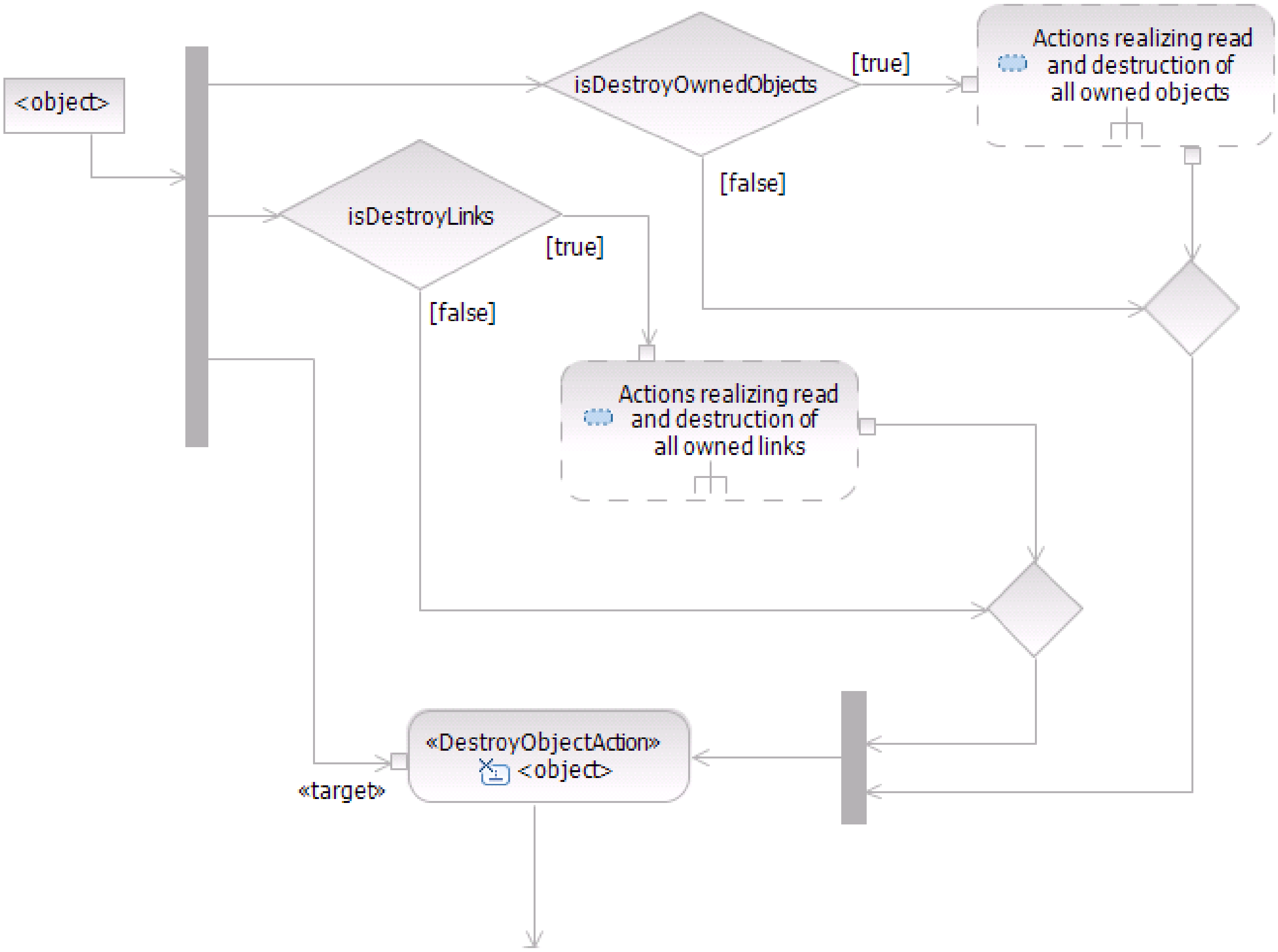
The UML Superstructure [2] defines a terminate pseudo state within state machines – fUML translators shall treat these states as equivalent to invoking a Destroy Object Action.

Destroying object behaviour while executing terminates its execution in the fUML virtual machine.

A destroy object action expression requires one incoming object flow whose type must be a class and allows no outgoing object flows.

11.2.2.4 Mapping

The following mapping starts with the object with its isDestroy… parameters set according to the inputs.


<object>
isDestroyOwnedObjects
[true]
Actions realizing read and destruction of all owned objects
[false]
isDestroyLinks
[true]
[false]
Actions realizing read and destruction of all owned links
«DestroyObjectAction»
<object>
«target»

A destroy object expression maps to a destroy object action.

With the non-parameterized form, the destroy object action sets isDestroyLinks and isDestroyOwnedObjects to true.

The parameterized form sets isDestroyLinks and isDestroyOwnedObjects to match the value of the equivalent parameters.

11.2.3 Read Self Action

From [2]:

Every action is ultimately a part of some behavior, which is in turn optionally attached in some way to the specification of a classifier (for example, as the body of a method or as part of a state machine). When the behavior executes, it does so in the context of some specific host instance of that classifier. This action produces this host instance, if any, on its output pin. The type of the output pin is the classifier to which the behavior is associated in the user model.

The read self action is used to reference attributes and operations of the host classifier. It is especially useful when an attribute is “shadowed” by a local variable declaration and thus must be qualified by the classifier’s identity.

11.2.3.1 Notation

```
this
```

11.2.3.2 Semantics

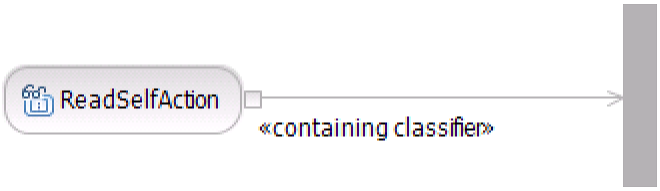
The statement must appear in a UAL fragment that is contained in a behavior that has a host classifier.

If the statement appears in a UAL fragment that itself is contained in a behavior that is acting as the body of a method, then that method must not be static.

The type of the result pin is the host classifier.

11.2.3.3 Mapping

A read self action expression maps to a read self action.



11.2.4 Value Specification Action

See 9.12, Literals.

11.2.5 Read Structural Feature Action

A read structural feature action retrieves the structural feature’s values. They are retrieved in order if the structural feature is ordered.

11.2.5.1 Examples

```
this.node
members.get(index).name
fridge.currentTemperature
```

11.2.5.2 Semantics

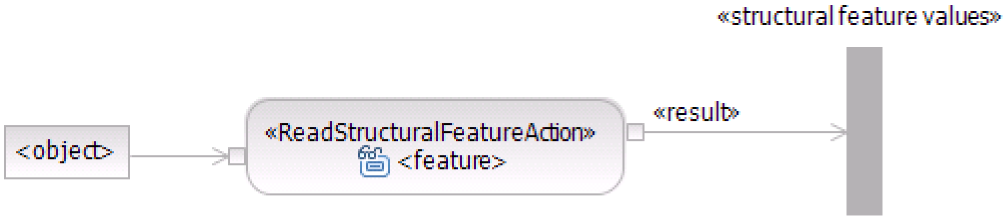
Structural features of the host classifier are visible within the host’s behaviors and all contained UAL fragments.

For “shadowed” structural features – those whose names match a local variable declaration – the host classifier can be referenced using “this” as its qualified name.

11.2.5.3 Mapping

This action is trivially mapped to the action of the same name. The input is the object whose structural feature is being read. The output is the values of the structural feature in the same order (if the feature is ordered) and with matching or greater multiplicity.

“This” as the object name maps to a read self action, whose output pin becomes the input pin of the read structural feature action. The read self action replaces the <object> reference on the mapping diagram.



11.2.6 Add Structural Feature Value Action

Structural features are modeled in UAL as ordered or unordered sets of values (ordered being modeled as a list.) They are **not** modeled as a set or list object in the Java sense, but rather simply use set or list syntax for manipulation.

If multiplicity is [1..1], then the assignment statement can be used in UAL.

Adding a structural feature value is the equivalent of adding a value to a set. If the set is ordered, then a 0-based insertion point must be provided and the value will be inserted in the set at exactly that point. If the structural feature us unordered then there is no insertion point and the value is simply added to the set. For an unordered feature, adding the same value has no effect.

11.2.6.1 Examples

```
this.houses.addAll(seller.houses)
this.houses.add(houseForSale)
this.houses.add(1, newestHouseForSale)  // first position
person.firstName = "Joe"
person.lastName = "Blow"
```

11.2.6.2 Semantics

Values of a structural feature may be ordered or unordered, even if the multiplicity maximum is 1.

Adding values to ordered structural features requires an insertion point (1 is the beginning) for a new value using the insertAt input pin. The notation is:

```
attribute.add(<index>, <value>)
```

The insertion point is omitted for unordered structural features. The notation is:

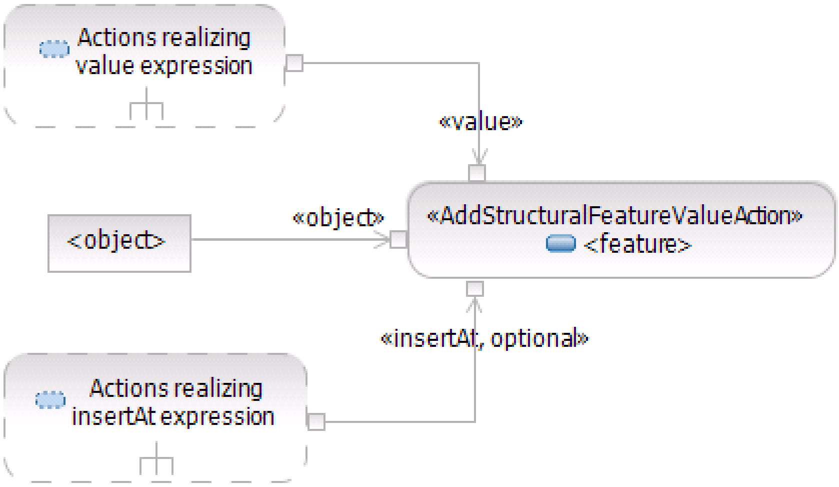
```
attribute.add(<value>)
```

Reinserting an existing value at a new position in an ordered unique structural feature moves the value to that position (this works because structural feature values are sets).

The insertion point is ignored when replacing all values.

11.2.6.3 Mapping

The insertAt calculation increments the resulting index by 1 as its last step in order that it matches fUML’s defined range where 1 indicates the first position.


Actions realizing value expression
«value»
<object>
«object»
«AddStructuralFeatureValueAction»
<feature>
«insertAt, optional»
Actions realizing insertAt expression

11.2.7 Remove Structural Feature Value Action

Structural features are modeled in UAL as ordered or unordered sets of values. They are **not** modeled as a set object in the Java sense, but rather simply use set syntax for manipulation.

Removing a structural feature value is the equivalent of removing a value from a set. If the set is ordered and non-unique, then a 1-based insertion point must be provided and the value will be removed at exactly that point if the value matches the element at that position.

If the structural feature is unordered and unique then no insertion point is required and the value is simply removed from the set.

There are three variations of the syntax:

- remove – removes one matching element or one element in a specified position (if necessary.)
- removeAll –removes all elements matching an input collection of elements
- clear – removes all elements.

11.2.7.1 Examples

```
this.houses.remove(soldHouse)  // remove a specific house
this.houses.remove(2)  // remove the 2nd house
scheduler.scheduledEvents.remove(event)
scheduler.scheduledEvents.clear()  // remove all events
this.eventPool.remove(3)  // remove the 3rd event
this.eventPool.removeAll(eventsOnTuesday)  // remove all listed events
this.name = null  // see clear structural feature action
```

11.2.7.2 Semantics

Removing a value succeeds even when it violates the minimum multiplicity.

Removing a value that does not exist has no effect.

If the feature is an association end, the semantics are the same as for destroying links, the participants of which are the object owning the structural feature and the value being removed.

Values of a structural feature may be duplicates in non-unique structural features.

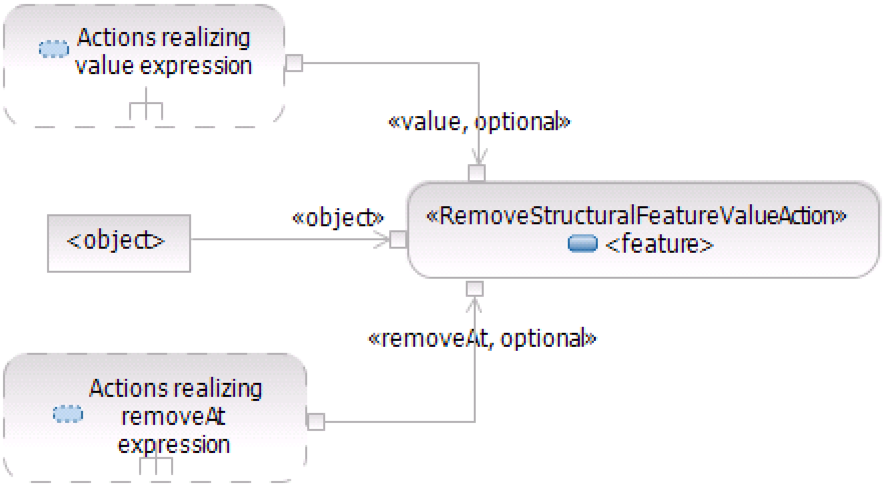
The remove format with index is required if with ordered non-unique structural features unless all instances of the element are to be removed.

The use of `null` does not imply a null reference, but rather an empty set, and may be used when multiplicity has a zero lower bound.

11.2.7.3 Mapping

With an ordered non-unique structural feature, use of the remove(<object>) format removes all instances by default; mapping to fUML shall set the *isRemoveDuplicates* attribute of the action to true. Else it shall be set to false.

The removeAt calculation increments the resulting index by 1 as its last step in order that it matches fUML's defined range where 1 indicates the first position.


Actions realizing value expression
«value, optional»
<object>
«object»
«RemoveStructuralFeatureValueAction»
<feature>
«removeAt, optional»
Actions realizing removeAt expression

11.2.8 Clear Structural Feature Action

Removes all values from a structural feature.

11.2.8.1 Examples

```
      this.houses.clear()  // removes all elements
      this.houses = null  // removes all elements if multiplicity 0..*
```

11.2.8.2 Semantics

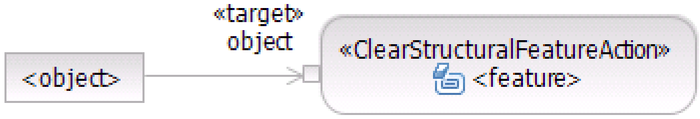
All values are removed even when that violates the minimum multiplicity of the structural feature. This is just as if the minimum were zero.

If the feature is an association end, the semantics are the same as for ClearAssociationAction on the object owning the structural feature.

The use of `null` does not imply a null reference, but rather an empty set, and may be used when multiplicity has a zero lower bound.

11.2.8.3 Mapping

Assignment to `null` when multiplicity lower bound is 0 maps to a `clear structural feature action`.



11.2.9 Read Link Action

ReadLinkAction is a link action that navigates across associations to retrieve objects on one end.

Associations are modeled in UAL as a collection of links with participating objects. A link can be referenced by its association name or role name, such that an ordered or unordered list of target objects is referenced.

In fUML, associations always own all their ends (see fUML Specification, clause 7.2.2), meaning that a read link action is always needed to read the ends of a link.

11.2.9.1 Notation

```
      <object>.<association>   // all links
      <object>.<role>   // all links
      <object>.<association or role>.get(<index>)   // one link
```

11.2.9.2 Semantics

This action navigates an association towards one end, which is the end that does not have an input pin to take its object (the “open” end).

The objects put on the result output pin are the ones participating in the association at the open end, conforming to the specified qualifiers, in order if the end is ordered.

The semantics are undefined for reading a link that violates the navigability or visibility of the open end.

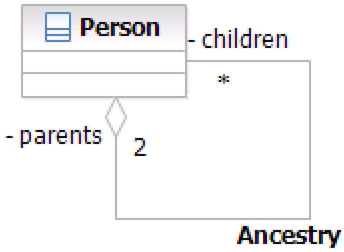
The type and ordering of the result output pin are the same as the type and ordering of the open association end.

The multiplicity of the open association end must be compatible with the multiplicity of the result output pin.

The open end must be navigable.

11.2.9.3 Disambiguation

It is possible for a classifier to have an association to itself, for example a person can be both a parent and a child.



To get all of Fred's children, one cannot use the association syntax:

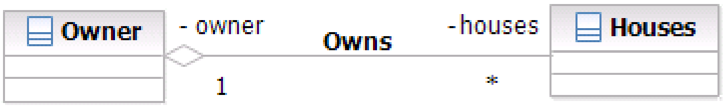
```
        fred.Ancestry
```

The translator shall, in fact, flag an error when this form is used in this ambiguous context. The solution is to use the role form of the syntax.

```
        fred.children
```

11.2.9.4 Examples

Suppose that an association `Owns` has ends `owner` and `houses`.



All houses owned by `jack`:

```
        jack.Owns
        jack.houses
```

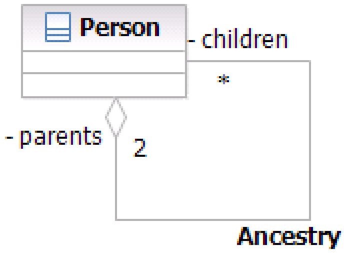
First house owned by `jack`:

```
        jack.Owns.get(0)
        jack.houses.get(0)
```

Does jack own the big red house?

```
        jack.Owns.contains(bigRedHouse)
        jack.houses.contains(bigRedHouse)
```

And using the more ambiguous example:



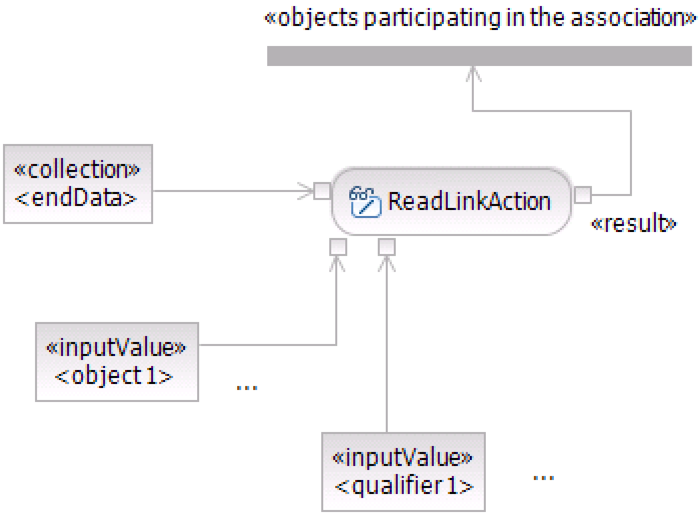
Find Andy's mother:

```
        Person mom = andy.parents.get(0);
        If (!mom.isFemale()) mom = andy.parents.get(1));
```

Find all of Andy's female children:

```
Set females;
for (Person p : andy.parents) p.isFemale() ? females.add(p);
```

11.2.9.5 Mapping



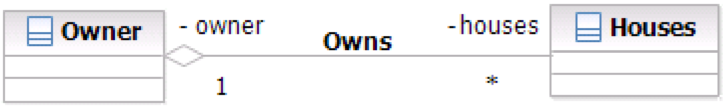
11.2.10 Create Link Action

By modeling associations as ordered or unordered sets is that we have the very familiar collection manipulation methods available.

So:

```
<object>.<assoc>.add(object>)
```

With our previous example:



… the statement:

```
jack.Owns.add(someHouse)
```

creates a new link of association Owns between the owner jack and the house someHouse.

If an end is ordered, a specific position index may be provided for the associated value. For example, if the houses end were ordered, then the following expression would make someHouse the first house in the list of houses for jack:

```
jack.Owns.add(1, someHouse)
```

Add the new listings to inventory:

```
inventory.Lists.addAll(newlyListedHouses)
```

Add the new listings at the beginning of the inventory so they are seen first:

```
inventory.Lists.addAll(1, newlyListedHouses)
```

11.2.10.1 Semantics

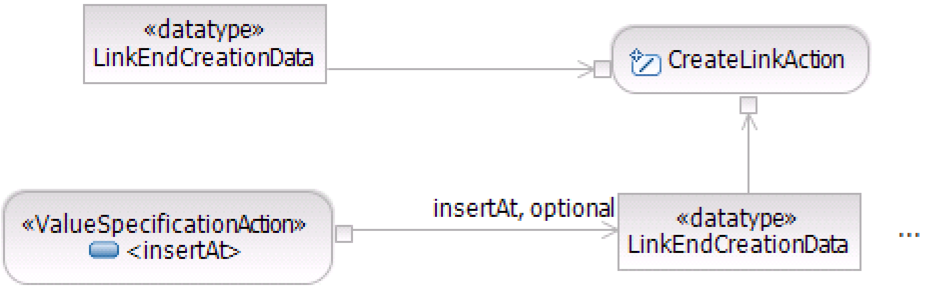
CreateLinkAction has no output pin.

An index of zero is a translator error.

11.2.10.2 Disambiguation

See clause 11.2.9.3.

11.2.10.3 Mapping



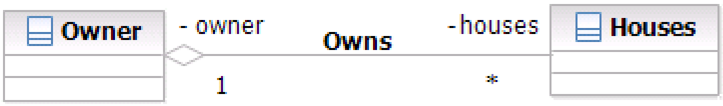
11.2.11 Destroy Link Action Expression

By modeling associations as ordered or unordered sets we have the very familiar collection manipulation methods available.

So:

```
<object>.<assoc>.remove(object>)
```

With our previous example:



… the statement:

```
jack.Owns.remove(someHouse)
```

… destroys the association `Owns` between the owner `jack` and the house `someHouse`.

If an end is ordered, a specific position index may be provided for the associated value. For example, if the houses end were ordered, then the following expression would remove the first house in the list of houses for jack:

```
jack.Owns.remove(1)
```

Remove the new listings from inventory:

```
inventory.Lists.removeAll(newlyListedHouses)
```

11.2.12 Semantics

An index of zero is a translator error.

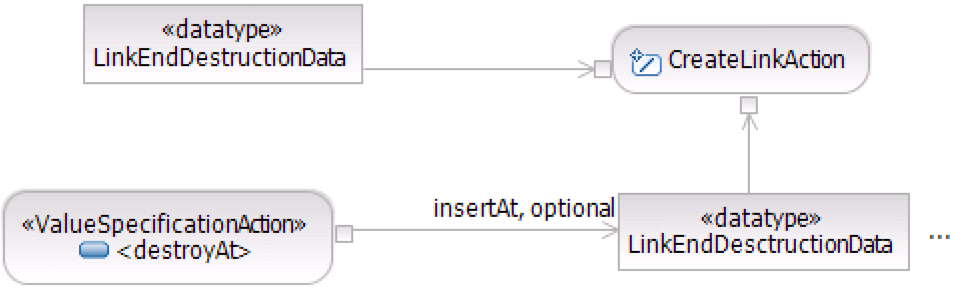
An index greater than the number of elements has undefined behavior.

Removing an element that is not in the set has no effect.

11.2.12.1 Disambiguation

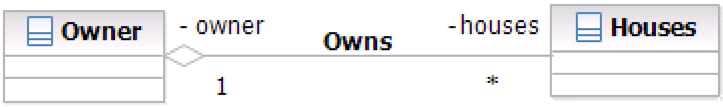
See clause 11.2.9.3.

11.2.13 Mapping



11.2.14 Clear Association Action

Destroys all links of an association in which a particular object participates.



If jack owns several houses, they can all be removed using:

```
jack.Owns.clear()
jack.houses.clear()
```

If the association is bidirectional, then the owner can be cleared in one of two ways:

```
someHouse.owner.clear()
someHouse.owner = null
```

As a reminder, setting a structural feature or association end to null is a synonym for the clear() operation.

11.2.14.1 Semantics

Clear removes all links, regardless of multiplicity. Therefore, it is possible to violate the multiplicity with this action.

11.2.14.2 Disambiguation

See clause 11.2.9.3.

11.3 Complete Actions

11.3.1 Read Extent Action Expression

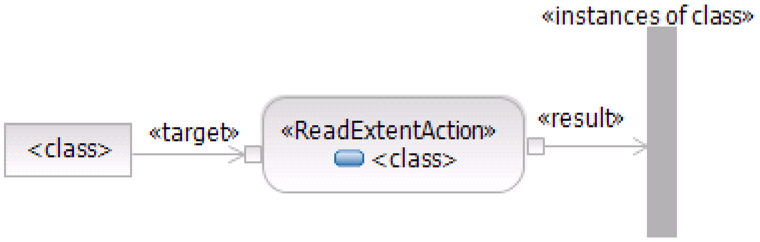
ReadExtentAction is an action that retrieves the current instances of a classifier.

All classifiers have extents in the fUML virtual machine. That is, all instances of each classifier are tracked at each locus, so it is possible to retrieve that list for purposes of searching. For example, to get all houses on the current locus, each one being of class House:

```
House.allINstances()
```

11.3.1.1 Mapping

```
<class>.allInstances()
```



11.3.2 Read Is Classified Object Action

ReadIsClassifiedObjectAction an action that determines whether a runtime object is classified by a given classifier.

Given a House named jacksHouse, checking if this specific house is a House looks like:

```
jacksHouse instanceof House
```

11.3.2.1 Examples

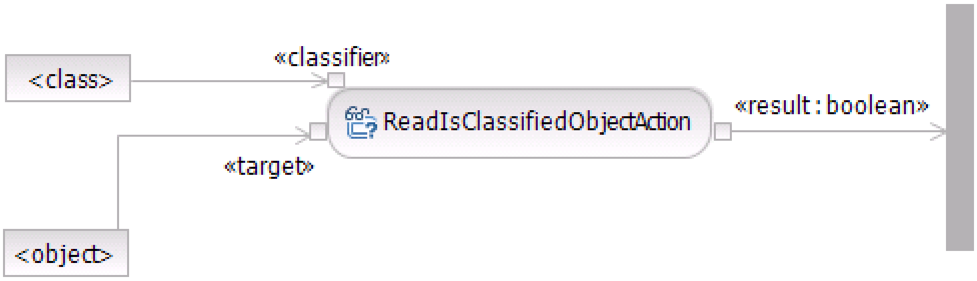
```
instanceof ActionActivation
instanceof SignalArrival
```

11.3.2.2 Semantics

isDirect is always false, in other words with this syntax the test is positive for the actual classification or any super classification.

This syntax makes no distinction between behavioral inheritance and interface inheritance.

11.3.2.3 Mapping



12 ANNEX A – Collection Class Library (Normative)

The collection classes define the operations and attributes that can be associated with generic collection types, specifically ordered and unordered sets, known herein as List and Set. The translator shall be responsible to correctly typing the collection in fUML or in the appropriate target language construct.

12.1 Collection

An abstract class (fUML excludes interfaces) that all collection classes specialize. A group of objects. No assumptions are made about the order of the collection (if any), or whether it may contain duplicate elements. Is iterable.

ElementType is denoted as `E` for brevity. Object is denoted as `o` for brevity when used as an argument.

12.1.1 Operations

```
add(E o)  // adds the element o to the collection
addAll(Collection c)  // adds all elements in c to the collection
clear()  // removes all elements
contains(Object o)  // true if collection contains o
containsAll(Collection c)  // true if collection contains all elements in c
equals(Object o)  // collections contain the same elements
isEmpty()  // collection has no elements
remove(Object o)  // removes o form collection
removeAll(Collection c)  // removes all elements in c from collection
retainAll(Collection c)  // removes all elements not in c from collection
size()  // number of elements in collection
```

12.2 Set

The familiar set abstraction. No duplicate elements permitted. Is not ordered. Extends the `Collection` interface.

12.2.1 Operations

No additional operations, however all add operations check for duplicates and do not add elements if already present in set.

12.3 List

Ordered collection, also known as a *sequence*. Duplicates are generally permitted. Allows positional access. Extends the `Collection` interface.

12.3.1 Operations

```
add(int index, E o)  // add o at position index
addAll(int index, Collection c)  // add elements in c at position index
get(int index)  // return the element at position index
indexOf(Object o)  // return the zero based position of o or -1 if not found
lastIndexOf(Object o)  // return the zero based position of the last
                       // occurrence of o or -1 if not found
remove(int index)  // remove the element at position index
set(int index, E o)  // replaces the element at position index with o
subList(int from, int to)  // returns a view of the portion of the list
```

```
                           // between from (inclusive, zero based) and
                           // to (exclusive, zero based)
```

12.4 Map

A mapping from keys to values. Each key can map to at most one value. A value can be mapped by many keys.

12.4.1 Operations

```
containsKey(Object key)  // true if map contains mapping for key
containsValue(Object value)  // true if map contains a mapping to value
get(Object key)  // returns the value to which key maps
keyset()  // returns a view on the set of keys in this map
put(K key, V value)  // associates key with (possibly new) value
putAll(Map m)  // copies mappings from m to collection
remove(Object key)  // removes mapping for key
values()  // returns a Collection view of the values in this map
```

12.5 tbd – queue, sortedSet, sortedMap

These additional specializations will be completed in the next draft.