

SDL Mapping for the UML Action Semantics

Morgan Bjorkander*, Ileana Ober*, and Thomas Weigert[†]

*Telelogic AB, Malmoe, Sweden

[†]Motorola, Schaumburg, Illinois, U.S.A.

August 15, 2000

Abstract. This chapter describes a mapping from the SDL language to an Action Semantics model. The purpose is to demonstrate that the action semantics is capable of representing a complete, complex, and practical action language.

1. Overview of SDL

The specification language SDL is widely used to describe the behavior of real-time and telecommunication systems and applications. SDL has been standardized by the International Telecommunications Union (ITU) in the Z.100 series of Recommendations.

While the UML aims to be applicable to a wide range of application domains, SDL has focused on the modeling of reactive, state/event driven systems. In order to subsume the possible variances of application domains, UML does not define all language concepts (such as its concurrency semantics) to the level of detail necessary to allow unambiguous interpretation. SDL, on the other hand, gives precise, formal semantics for all its concepts.

SDL has its strength in the modeling of concurrent active objects, the modeling of the hierarchical structure of active objects, and the modeling of their connection by means of well-defined interfaces. To support the demands of the real-time and telecom application domains, SDL provides the following additional concepts:

- A view of system behavior as induced by the collaboration of *strongly encapsulated active objects* (agents) interacting via *bidirectional interfaces* (gates) connected by channels.
- A complete *action language* that makes SDL independent of implementation languages. In line with the rest of SDL, behavior is specified in an imperative style and may be mixed with graphical SDL.
- *Object oriented data* based on single inheritance and with both polymorphic references (objects) and values, even in the same inheritance hierarchy. Type safety is preserved in the presence of covariance through multiple dispatch.
- An *exception handling* mechanism for behavior specified either through state machines or constructs of the action language that makes SDL suitable as a design/high-level implementation language.
- Composite states that are defined by *separate state diagrams* (for scalability); *entry/exit points* are used instead of state boundary crossing (for encapsulation), any composite state can be of a *state type* (for reuse), and state types can be *parameterized* (for even more reuse).
- Object-orientation applied to active objects including inheritance of behavior specified through state machines and inheritance of the (hierarchical) structure and connection of active objects.
- *Virtual types* that allow the redefinition of inherited types.
- *Constraints* on redefinitions in subclasses and on actual parameters in parameterization that afford strong error checking at modeling time.

In Recommendation Z.109, the ITU established SDL as a UML profile. With its adoption in 1999, the SDL UML profile has become the first standardized real-time UML profile. The SDL UML profile allows users to treat an SDL model as a specialization of the generic UML model thus giving more specific meaning to UML concepts geared towards representing entities in the real-time and telecom application domains (blocks, processes, states, gates, channels, etc.).

2. Formal Definition of SDL

The formal definition of SDL defines how to transform a system specification into the abstract syntax and defines how to interpret a specification given in terms of the abstract syntax. The formal definition is presented in Annex F to Recommendation Z.100.

In order to define the formal semantics of SDL, the language definition is decomposed into several parts:

- the grammars,
- the well-formedness conditions,
- the transformation rules, and
- the dynamic semantics.

The *grammars* define the set of syntactically correct SDL specifications. In Z.100, a concrete textual, a concrete graphical, and an abstract grammar are defined. The abstract grammar is obtained from the concrete grammars by removing irrelevant details such as separators and lexical rules.

The *well-formedness conditions* define which grammatically correct specifications are also correct with respect to context information, such as which names are allowed at a given place, or which kind of values can be assigned to a variable.

Furthermore, some language constructs appearing in the concrete grammars are replaced by other language elements in the abstract grammar using *transformation rules* to keep the set of core concepts small. These transformations are described in the model paragraphs of Z.100, and are formally expressed as rewrite rules.

The *dynamic semantics* is given only to syntactically correct SDL specifications that satisfy the well-formedness conditions. The dynamic semantics defines the set of computations associated with a specification.

2.1 Grammar

The *grammars* of SDL are represented using Backus-Naur-Form (BNF) with extensions to capture the graphical language constructs. However, the grammar in the main text of Z.100 is designed to be a presentation grammar; some restrictions that guarantee uniqueness of the semantics have been captured in the grammar using annotations that stress the semantic aspect of a symbol instead. The translation from the concrete textual SDL grammar to the abstract grammar of Z.100 consists of two steps. The first step from the concrete textual SDL grammar to the abstract grammar is derived from the correspondence between the two grammars, and removes irrelevant details such as separators and lexical rules. The second step, translating the concrete grammar to the abstract grammar, is formally captured by a set of transformation rules (see Z.100, Annex F.2).

2.2 Well-formedness Conditions

The *well-formedness conditions* define additional constraints that a specification has to satisfy. An SDL specification is *valid* if and only if it satisfies the syntactic rules and the static conditions of SDL.

The well-formedness conditions encompass:

- *Scope/visibility rules*: The definition of an entity introduces an identifier that may be used as the reference to the entity. Only visible identifiers must be used. The scope/visibility rules are applied to determine whether the corresponding definition of an identifier is visible or not.
- *Disambiguation rules*: Sometimes a name might refer to several identifiers. Rules are applied to find out the correct one.
- *Static semantic constraints*: These are rules applicable to specific entities. For example, there must exist at least one block definition or a process definition in a system definition. Other rules refer to the correctness of the concrete syntax, and have no counterpart in the abstract syntax. For example, the name at the beginning and at the end of a definition have to match.
- *Data type consistency rules*: These rules ensure that no operation is applied to operands that do not match its argument types. The sort of an actual parameter must be compatible with that of the corresponding formal parameter; the sort of an expression must be compatible with that of the variable to which the expression is assigned.

All but the last of these constraints can be defined and checked independent of the definition of the dynamic semantics (see Z.100, Annex F.2). Due to polymorphism, data type consistency may require dynamic type checks.

2.3 Transformation Rules

Some constructs of SDL are considered to be “derived concrete syntax” (i.e., a shorthand notation) for other equivalent constructs of the concrete syntax. For example, omitting an input for a signal is derived concrete syntax for an input for that signal followed by a null transition back to the same state.

The properties of a shorthand notation are derived from the way it is modelled in terms of (or transformed to) the primitive concepts. In order to ensure easy and unambiguous use of the shorthand notations, and to reduce side effects when several shorthand notations are combined, these concepts are transformed in a specified order.

The result of the transformation of a fragment of text in derived concrete syntax is either another fragment of text in derived concrete syntax, or a fragment of text in concrete syntax. The result of the transformation may also be empty. In the latter case, the original text is removed from the specification.

Annex F.2 of Z.100 prescribes the transformation of SDL specifications by a sequence of *transformation steps*. Each transformation step consists of a set of single transformations as stated in the Model sections, and determines how to handle one special class of shorthand notations. A single transformation is realized by the application of a rewrite rule to the concrete specification, which essentially means to replace parts of the specification by other parts as defined by the rule. The result of one step is used as input for the next step.

2.4 Dynamic Semantics

The *dynamic semantics* (see Z.100, Annex F.3) consists of the following parts (see Figure 63):

- The *SDL Abstract Machine (SAM)* is defined using the mathematical framework of evolving algebras or Abstract State Machines (ASM). The definition of the SAM is divided into three parts, corresponding to the abstract syntax: basic signal flow concepts (signals, timers, exceptions, gates, channels) and agents (processes, blocks) are represented by specialized ASM agents; behavior primitives constitute the abstract machine instructions of the SAM.
- *SDL Abstract Machine Programs* define the set of computations. These programs consist of an initialization phase and an execution phase. SAM programs have predefined parts that are the same for all SDL specifications, and variable parts that are generated from the abstract syntax representation of a given SDL specification.
- The *compilation function* maps behaviour representations into the SDL Abstract Machine primitives. This function constitutes an abstract compiler taking the AST of the state machines and actions as input and transforming it into the abstract machine instructions. Result of the compilation are sets of behavior primitives modeling the actions of the SDL agents.
- The *initialization* defines the static structural properties of the system. The initial state of a system is reached by creating the SDL system agent, and by activating this agent in the pre-initial state. The initialization recursively unfolds the static structure of the system, creating further SDL agents as specified. The same process will be utilized in the subsequent execution phase, whenever SDL agents are created dynamically. (In other words, the initialization is merely the instantiation of the SDL system agent.)

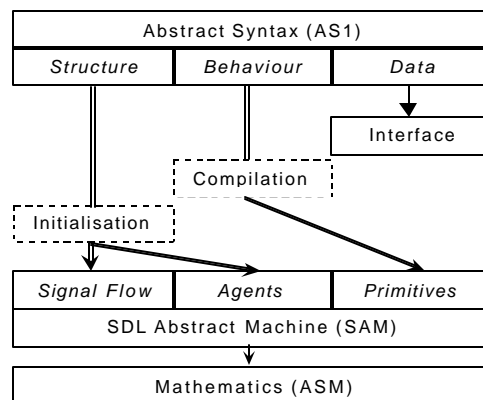


Figure 63. Overview of the SDL Dynamic Semantics Specification in Z.100

The formal semantics is defined starting from the abstract syntax of SDL. From this abstract syntax, a behavior model is derived. The SAM is expressed in terms of the ASM model, which has a mathematical definition (using axiomatic

semantics). The compilation defines an abstract compiler mapping the behavior parts of SDL to abstract code (denotational semantics). Finally, the initialization describes an interpretation of the abstract syntax to build the initial system structure (operational semantics).

The *dynamic semantics* associates with each SDL specification a particular multi-agent, real-time ASM. Intuitively, an ASM consists of a set of autonomous agents. The behaviour of these agents is determined by ASM programs, each consisting of a transition rule, which defines the set of possible computations (called “runs” in the context of ASM). The agents cooperatively perform concurrent machine runs. Each agent has its own partial view on a global state, which is defined by a set of static and dynamic functions and domains. By having non-empty intersections of partial views, interaction among agents can be modelled. An introduction to the ASM model is given in Z.100, Annex F.1.

3. Strategy and Conventions

In this chapter, we shall present the dynamic semantics of a subset of SDL through the UML action semantics. Rather than associating an ASM with an SDL specification, we shall show how an SDL specification can be translated into a UML model that utilizes the UML action semantics. The constructs of SDL are, therefore, given their meaning by the translation to UML.

We shall focus on the subset of SDL that allows the specification of actions in the UML sense. In terms of Z.100, this subset is characterized by the productions of the concrete SDL grammar representing a transition (see Z.100, §11.12), a procedure containing only a start transition (see Z.100, §9.4), or an operation (see Z.100, §12.1.8), respectively.

In the following sections, we describe a compiler (a translation procedure) which, when given a specification of the action subset of SDL will produce a UML model. This translation procedure will operate on the abstract syntax of SDL (see Section 2.1) after the transformation rules (see Section 2.3) have been applied and removed any derived concrete syntax. (In order to appreciate the transformation from the SDL concrete syntax to the abstract syntax, this chapter should be read in conjunction with Z.100.)

The resultant UML model will have a behavioral interpretation according to the UML action semantics equivalent to that of the original SDL model. Note that this is not completely true, as there are aspects of SDL which have no counterpart in the UML action semantics and can, therefore, not be expressed in UML. (The following sections call out where the UML semantics falls short of being able to express the action subset of SDL.)

For each node of the SDL abstract syntax, the subsequent presentation will adhere to the following organization and structure:

- The fragment of the abstract syntax of SDL derived from this node is shown in UML syntax as a metamodel. As the abstract syntax of SDL in Z.100 is given in extended BNF, we have made small adjustments to allow presentation as a metamodel. We have also omitted certain nodes of the SDL abstract syntax that are not relevant for the specification of actions.
- An informal description of the dynamic semantics of this node is given in terms of the SDL abstract syntax, closely following the text in Z.100.
- A UML model is given which defines the dynamic semantics of this node.

Taken together, the abstract syntax/model pairs of the following sections define a recursive descent compiler which will traverse the abstract syntax tree derived from a given SDL specification (involving only the action subset of SDL) and at each node of the abstract syntax tree emit a fragment of a UML model. After the abstract syntax tree has been traversed completely, we shall have obtained a complete and valid UML model. The dynamic semantics of this UML model also gives the dynamic semantics of the original SDL specification (modulo those aspects of SDL which cannot be expressed in UML).

The UML models given for each node of the SDL abstract syntax are only partial models as they depend on the further translation of contained nodes of the SDL abstract syntax. At these places, the translation procedure will recursively descend to the contained nodes. In the diagrams describing these models, we shall use the convention that an instance of a subsystem bearing the name of a node of the abstract syntax tree will represent the UML model derived from the application of the translation procedure to that node. In other words, subsystem instances will denote the recursive application of the compiler to a node of the SDL abstract syntax matching the name of the subsystem.

The UML model fragments giving the semantics of the SDL constructs tend to be large. To enhance readability of the diagrams and focus on the essentials, we have introduced a number of derived associations and classes:

Instances of the abstract metaclass *Action* and its subclasses are never connected directly to another action. They either connect via an associated *ControlFlow*, or via an associated *OutputPin/DataFlow/InputPin* chain. These intermediate classes are suppressed. We shall show an associated *ControlFlow* as a derived association between two actions with the role name taken from the association end connected to the action. We shall show an associated *OutputPin/DataFlow/InputPin* chain as a derived association between two actions with the role names taken from the association ends connected to the *InputPin* and *OutputPin*, respectively. We omit the special notation indicating a derived model element.

The test and body actions of a clause often are *GroupActions* merely for the purpose of containing a number of actions. In those situations, we have omitted this *GroupAction* and show the embedded actions as being part of the clause instead.

At times we place model elements that are not owned by an object within its attribute compartment, where this results in significant space savings and no confusion is likely to arise.

Finally, we shall rely on a constrained variant of the metaclass *Clause*, indicated by the stereotype «else», which represents a *Clause* with a test that always evaluates to true.

We illustrate the translation procedure by a simple, but rather contrived example. Consider the following fragment of SDL. Let *y* be a variable holding an object for which an *init* method with two integer parameters is defined.

```
Lbl: { dcl x Integer;
      if (y = 3) {
          y.init(x, 0);
          break Lbl; }
    }
```

The intuitive meaning of this SDL fragment is as expected: A variable *x* is declared but not initialized. Then, if *y* equals 3, the *init* method is invoked on *y*, followed by a *break*, which causes flow of control to continue after the statement with the matching label.

The abstract syntax tree corresponding to this fragment derives from *Block-node* and is shown in Figure 64. In the following, we shall indicate elements of the SDL abstract syntax by formatting them in italic font.

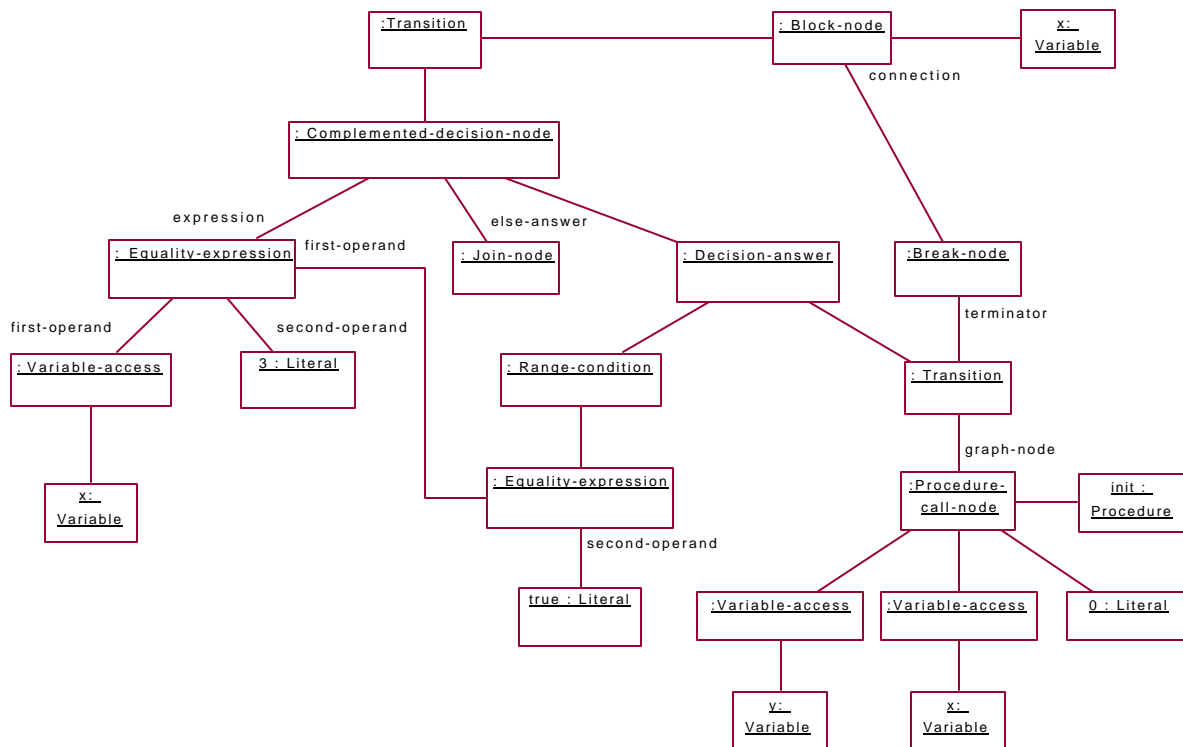


Figure 64. Abstract Syntax Tree for Example Specification Fragment

We illustrate the translation procedure by showing how it gradually generates the corresponding UML model. The translation begins at the *Block-node*. Figure 65 shows that the result of applying the translation of *Block-node* (a *Compound-node*) as described in Figure 93 is a *GroupAction*. Since the variable x declared locally in the scope of the *Block-node* is not initialized, no model elements corresponding to the variable initialization is produced. The variable y referenced in this fragment is declared outside the scope of the *Block-node*.

The next step (see Figure 66) translates the *Transition*, which is a *Complemented-decision-node*, as described in Figure 90 into a *ConditionalAction* containing two clauses: one corresponding to the consequent branch of the conditional, and one corresponding to the alternative branch of the conditional.

In the next step (see Figure 67) the content of the first clause is translated. The *Range-condition* forming the test of the clause and the *Expression* are *Equality-expressions* and are translated as shown in Figure 98. The body of the clause is represented by a *Transition* and is translated as shown in Figure 73.

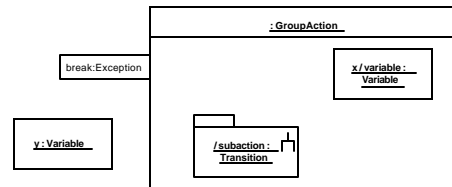


Figure 65. Translation of Example Specification (step 1)

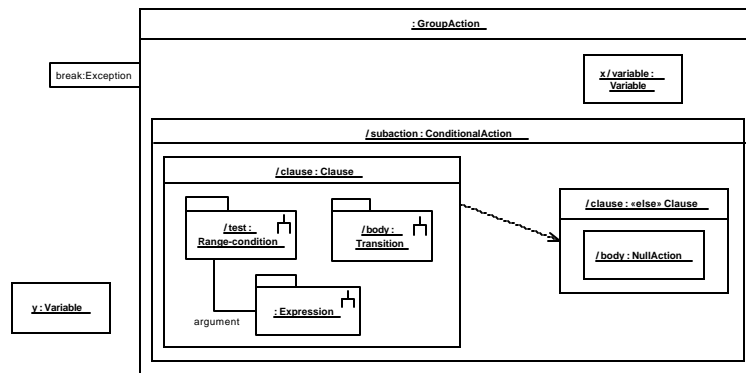


Figure 66. Translation of Example Specification (step 2)

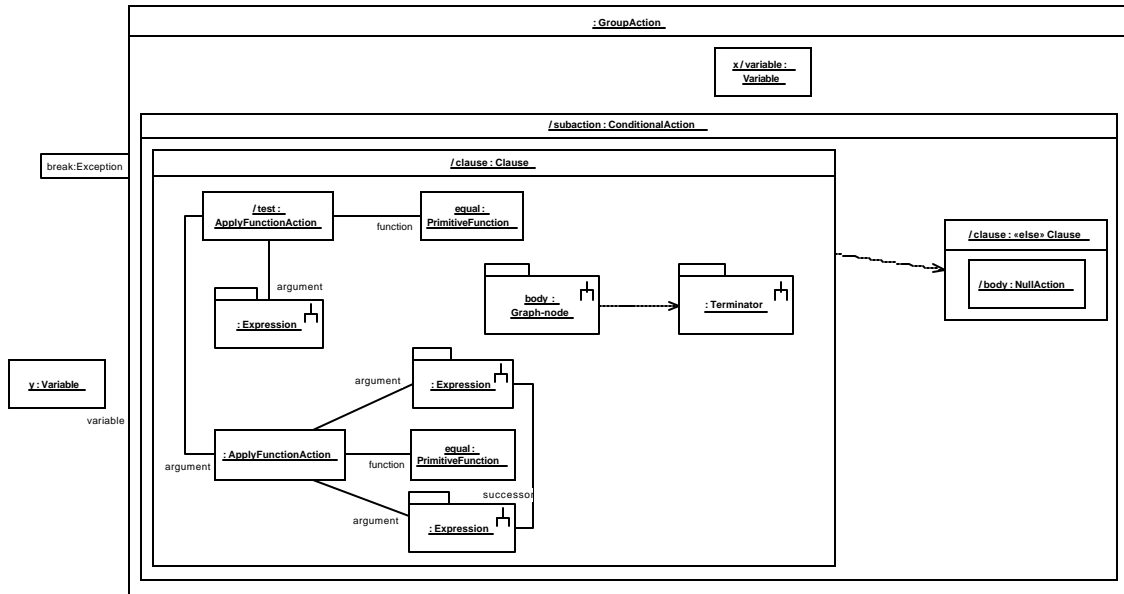


Figure 67. Translation of Example Specification (step 3)

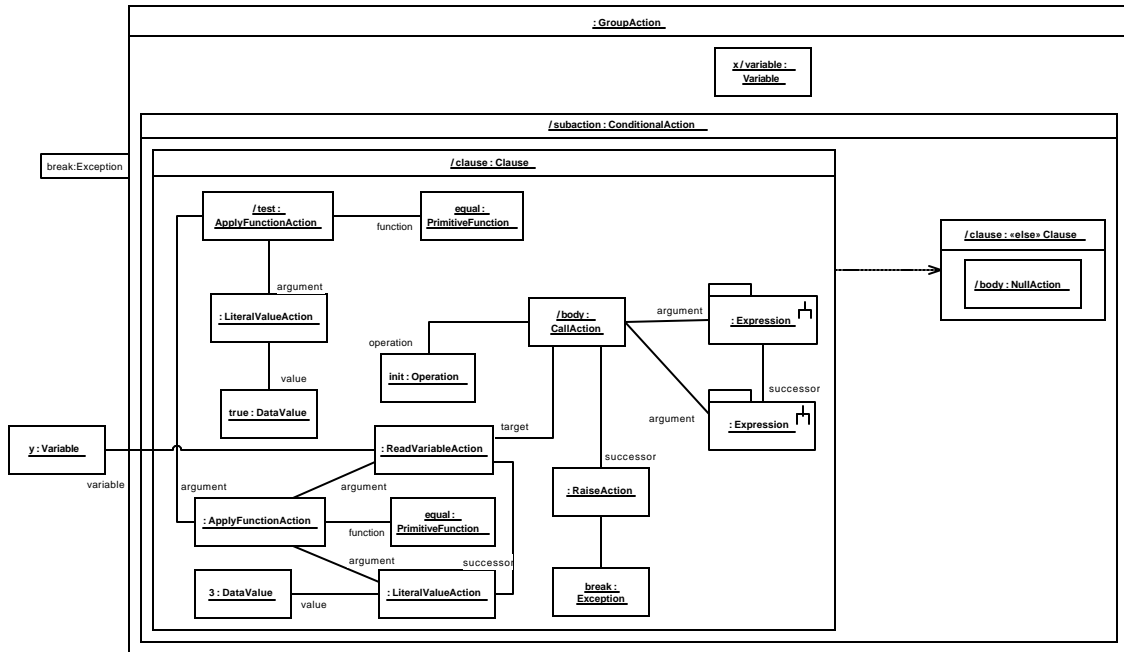


Figure 68. Translation of Example Specification (step 4)

4. Procedure

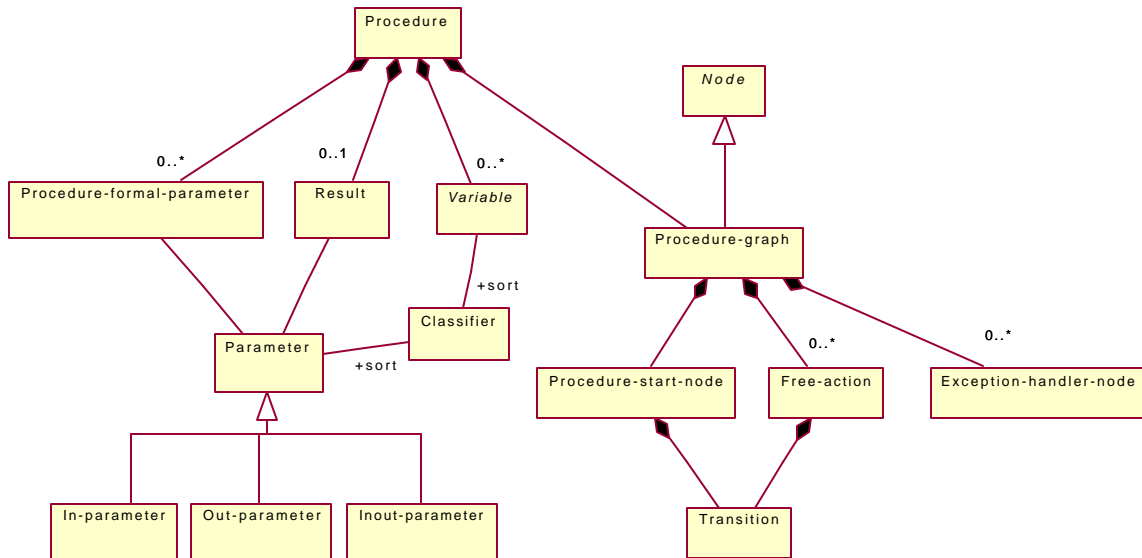


Figure 70. Abstract Syntax for Procedure

The interpretation of a *Call-node* (see Section 7) causes the creation of a procedure instance and the interpretation to commence with the interpretation of the *Procedure-graph* in the following way:

- A local variable is created for each *In-parameter*, having the *Sort* of the *In-parameter*. The variable is associated with the result of the *Expression* given by the corresponding actual parameter, if present, by interpreting an assignment between the variable and the expression.
- A local variable is created for each *Out-parameter*, having the *Sort* of the *Out-parameter*. The variable is not initialized.
- A local variable is created for each *Variable* in the *Procedure*.
- Each *Inout-parameter* denotes a variable defined in an outer scope that is given by the actual parameter expression (see Section 7).
- The *Transition* contained in the *Procedure-start-node* is interpreted.
- Before interpretation of a *Return-node* contained in the *Procedure-graph*, the *Out-parameters* are given the values of the corresponding local variable.

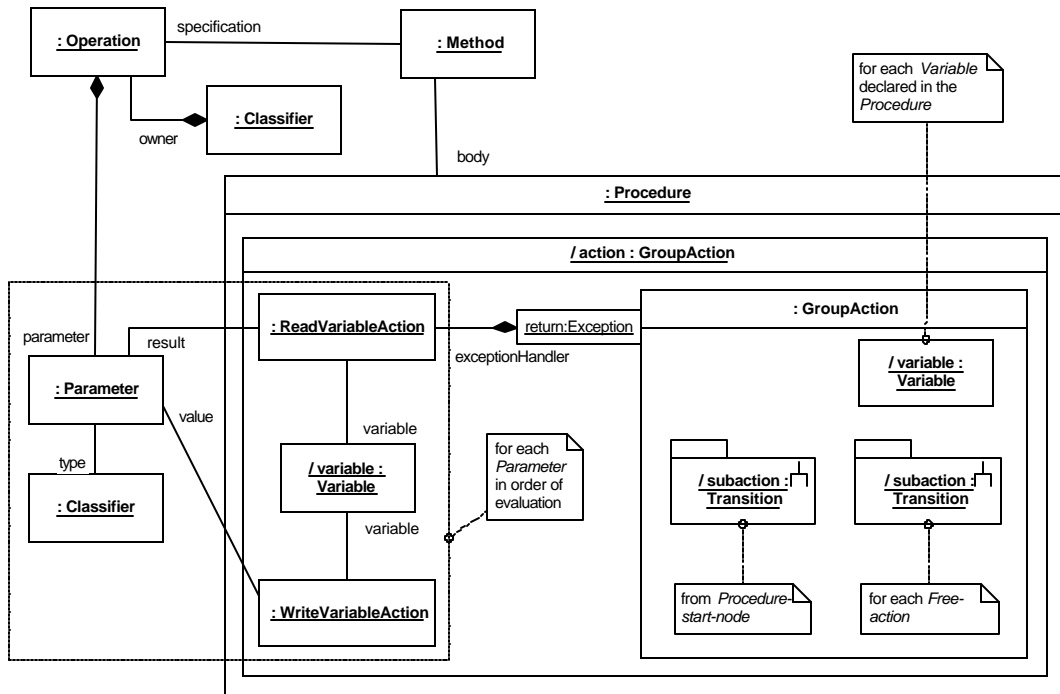


Figure 71. Action Semantics for Procedure

Note. The determination of the owner of the Operation representing a *Procedure* is beyond the scope of the UML Action Semantics.

5. Transition

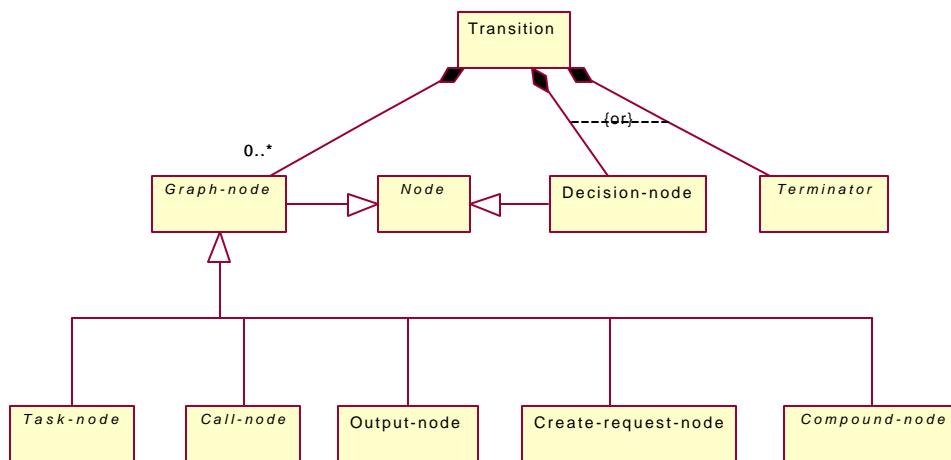


Figure 72. Abstract Syntax for Transition

A transition performs a sequence of actions. During a transition, the state of objects and variables may be manipulated and signals may be output.

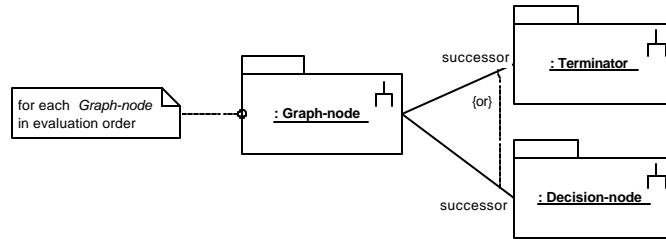


Figure 73. Action Semantics for Transition

A transition ends with a *Decision-node* or a *Terminator*, that is, with a return, with an exception being raised, or with the transfer of control to another transition. The interpretation of a *Terminator* is described in Section 6.

The interpretation of a *Graph-node* is described in Section 7. Note that in the figure above the mapping of only one *Graph-node* is shown. For each *Graph-node*, the resultant subsystem is inserted and connected by associations corresponding to the order of evaluation. The last subsystem has an association to the subsystem derived from the *Terminator* or *Decision-node*, respectively.

6. Terminator

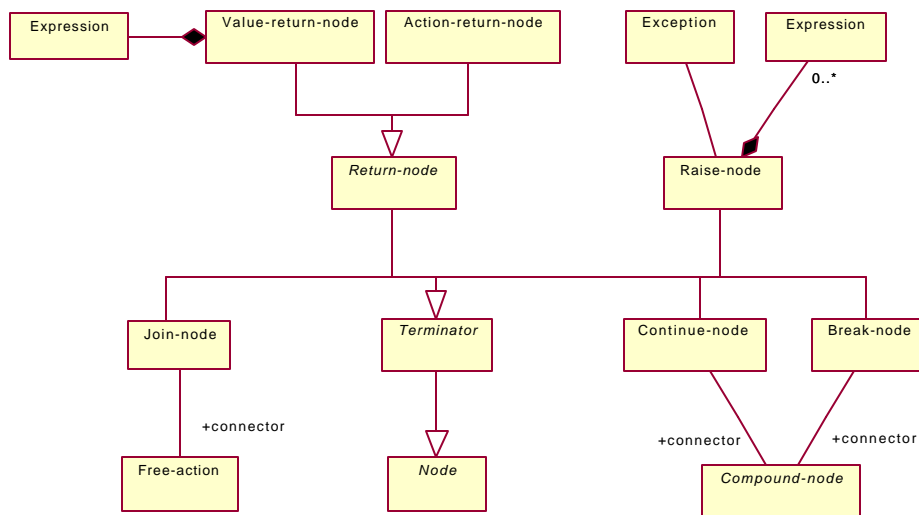


Figure 74. Abstract Syntax for Terminator

6.1 Join

When a *Join-node* is interpreted, interpretation continues with the *Transition* of the associated *Free-action*.

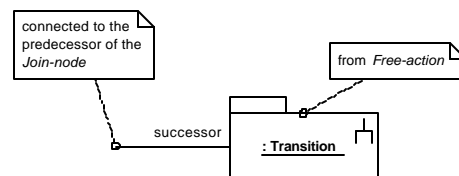


Figure 75. Action Semantics for Join-node

6.2 Return

A *Return-node* is interpreted in the following way:

- All variables created by the interpretation of the *Procedure-graph* (see Section 4) will cease to exist.
- The interpretation of the *Procedure-graph* is completed and the procedure instance ceases to exist.
- If a *Value-return-node* is interpreted, the result of *Expression* is returned to the calling context.
- Hereafter, interpretation of the calling context continues at the node following the call.

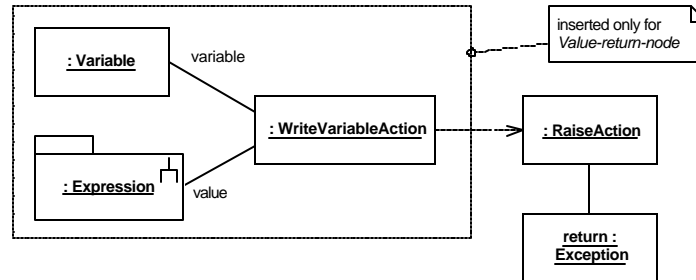


Figure 76. Action Semantics for Return-node

6.3 Raise

Interpretation of a *Raise-node* creates an exception instance (see Section 10 for the interpretation of an exception instance). The data items that are conveyed by the exception instance are the results of interpretation of the *Expressions*.

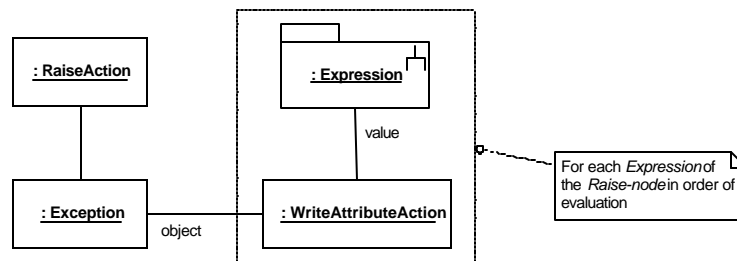


Figure 77. Action Semantics for Raise-node

6.4 Break and Continue

Interpretation of a *Break-node* or *Continue-node* transfers control to the associated *Compound-node*.

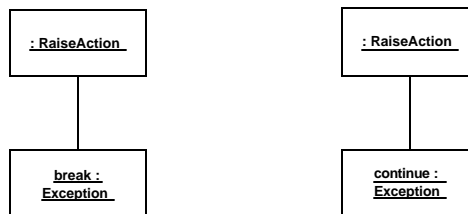


Figure 78. Action Semantics for Break-node and Continue-node

7. Action

7.1 Task

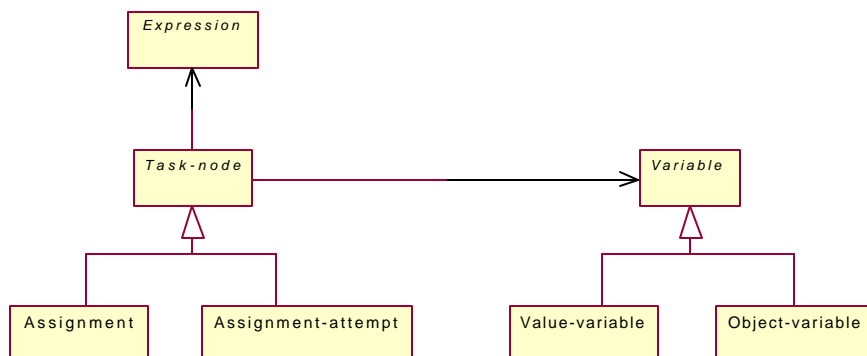


Figure 79. Abstract Syntax for Task-node

A *Task-node* is represented by a *WriteVariableAction*. If the *Task-node* was an *Assignment*, the mapping is as follows:

- a) If the *Variable* is a *Value-variable* (i.e., it has a value sort), then the result of the *Expression* is copied onto the *Variable* by interpreting the *copy* method defined by the data type definition that introduced the sort of the *Variable*, given *Expression* as actual parameter. If the *Expression* is *Null*, the predefined exception *InvalidReference* is raised.

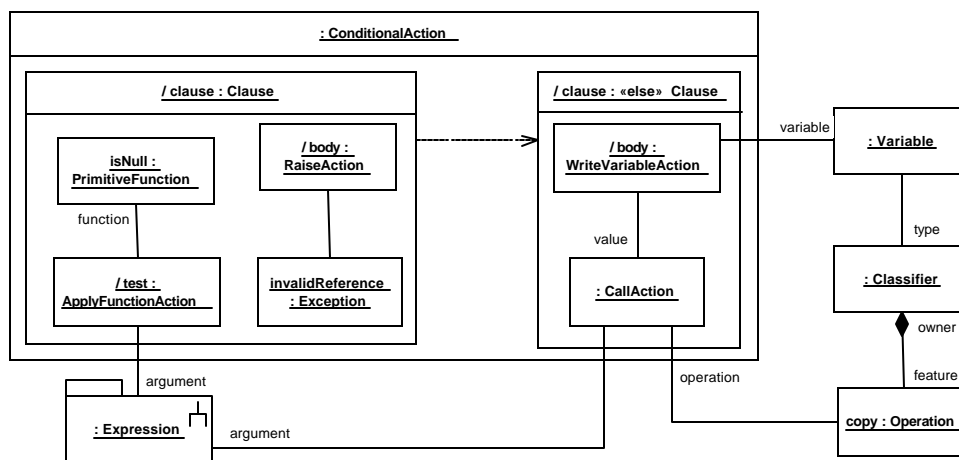


Figure 80. Action Semantics for Assignment involving a Value-variable

- b) If the *Variable* is an *Object-variable* (i.e., it has an object sort) and the result of the *Expression* is an object, the *Variable* is associated with the object that is the result of the *Expression*. If the result of the *Expression* is a value, a clone of the result of *Expression* is constructed by interpreting the *clone* operator defined by the data type definition that introduced the sort of the *Variable*, given *Expression* as actual parameter. The *Variable* is associated with a reference to the cloned value.

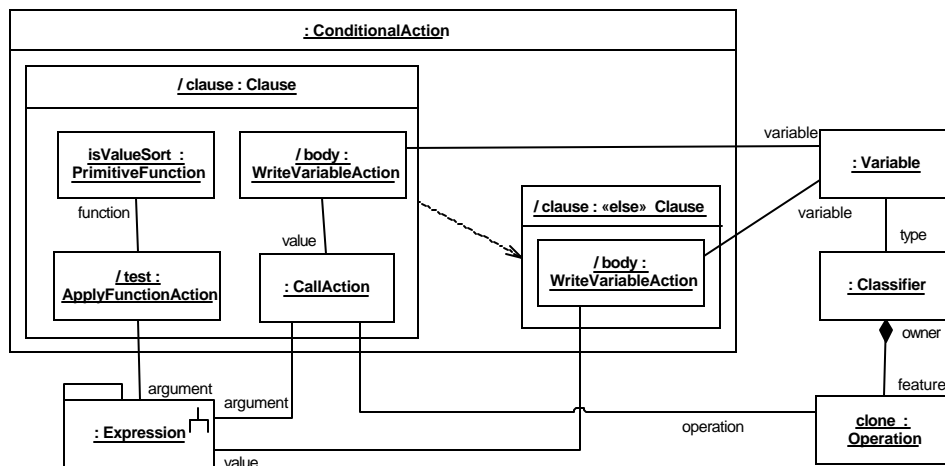


Figure 81. Action Semantics for Assignment involving an Object-variable

If the *Task-node* was an *Assignment-attempt*, if the dynamic sort of the *Expression* is sort compatible (see Z.100, §12.1.3) to the sort of the *Variable*, an Assignment involving the *Variable* and the *Expression* is interpreted. Otherwise, the *Variable* is associated with Null.

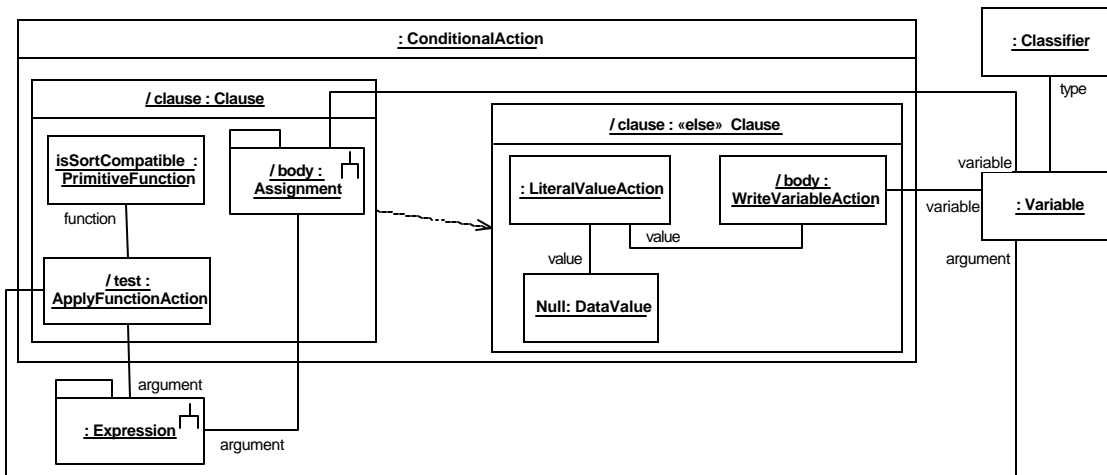


Figure 82. Action Semantics for Assignment-attempt

7.2 Create request

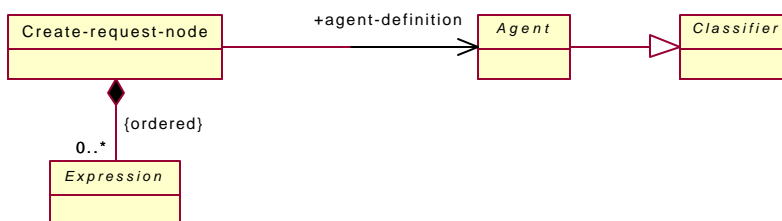


Figure 83. Abstract Syntax for Create-request-node

The interpretation of a *Create-request-node* causes the creation of an agent instance either inside the agent that performs the create or in the agent that contains the agent that performs the create. The *Parent-expression* of the created agent (see Z.100, §9) has the data value returned by *Self-expression* of the creating agent. The *Self-expression* of the created agent (Z.100, §9) and *Offspring-expression* of the creating agent (Z.100, §9) return the identify of the created agent. When an agent instance is created the actual parameter expressions are interpreted in the order given, and assigned (as defined in Z.100, §12.3.3) to the corresponding formal parameters. If an attempt is made to create more agent instances than specified by the maximum number of instances in the agent definition, then no new instance is created, the *Offspring-expression* of the creating agent has the result Null and interpretation continues.

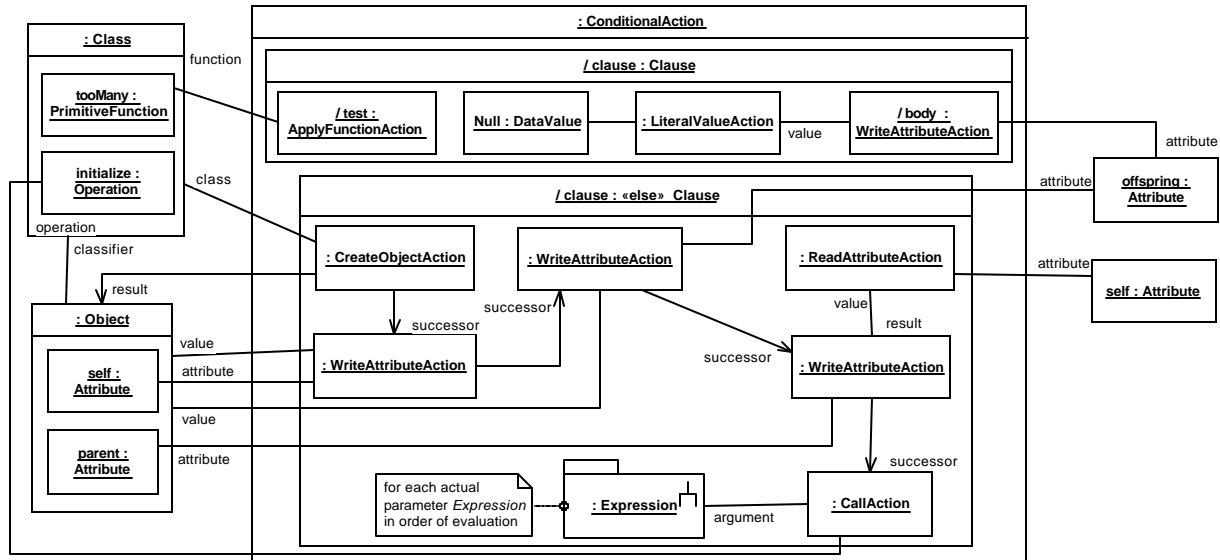


Figure 84. Action Semantics for Create-request-node

Note. The semantics of the interpretation of an agent is beyond the scope of the UML action semantics.

7.3 Call

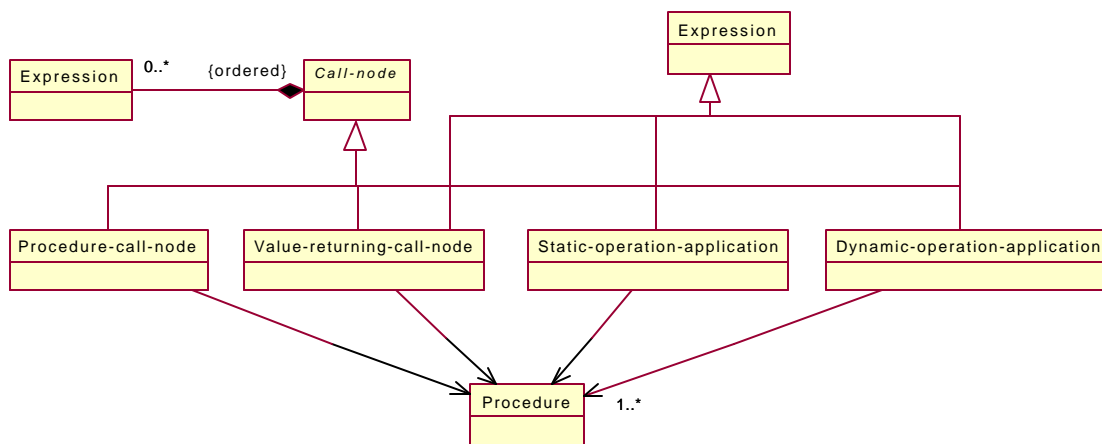


Figure 85. Abstract Syntax for Call-node

The interpretation of a *Call-node* interprets the actual parameter *Expressions* in the order given and then transfers the interpretation to the *Procedure* referenced, and the procedure graph of this *Procedure* is interpreted (see Section 4).

The interpretation of the transition containing a *Procedure-Call-node* continues when the interpretation of the called procedure is finished.

The interpretation of the transition containing a *Value-returning-call-node*, a *Static-operation-application*, and a *Dynamic-operation-application* continues when the interpretation of the called procedure is finished and the result of the called procedure is returned.

In the case of a *Dynamic-operation-application* a unique *Procedure* is selected by applying the algorithm described in Z.100, §12.2.7 to the set of *Procedures* referenced. If the result of an actual parameter corresponding to a virtual argument was Null, the predefined exception *InvalidReference* is raised.

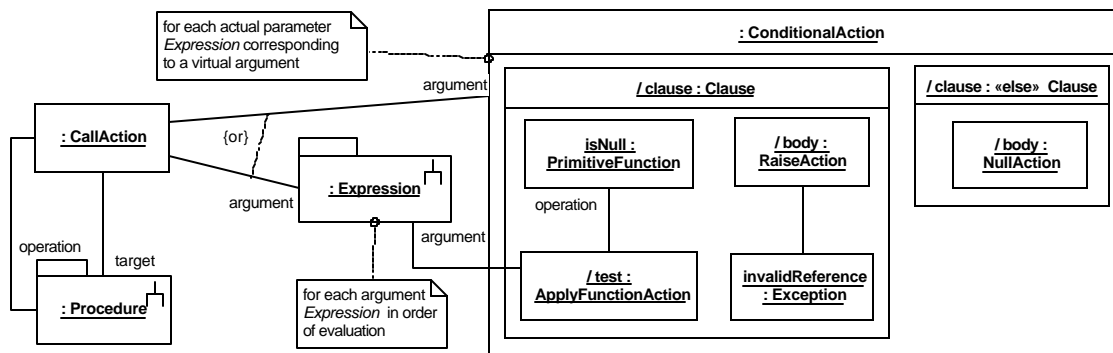


Figure 86. Action Semantics for Call-node

Note that the target of the `CallAction` will be an instance of a `Classifier`, once the translation for `Procedure` is applied.

Note. The semantics of the selection of the dynamically called `Procedure` in a `Dynamic-operation-application` is beyond the scope of the UML Action Semantics.

7.4 Output

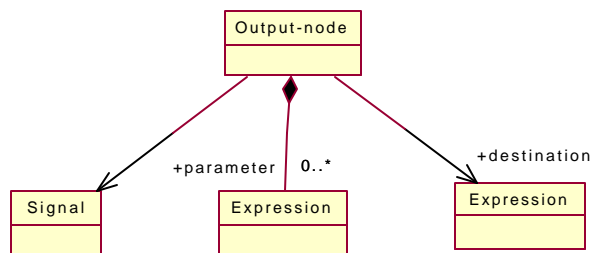


Figure 87. Abstract Syntax for Output-node

The receiver (i.e., the target object) is determined from the destination `Expression` as described in Z.100, §11.13.4 and the compatibility check for the dynamic sort of the `pid` expression (see Z.100, §12.1.6) is performed for the signal instance. If the destination `Expression` evaluates to Null, the signal instance will be discarded.

If both the sender and the receiver are contained in the same process instance, then the data items conveyed by the signal instance are the results of the actual parameters of the output. If the sender and the receiver are contained in different process instances then the data values conveyed by the signal instance are newly created replicates of the results of the actual parameters of the output and share no references with the results of the actual parameters of the output. When there are cycles of references in the result of the actual parameters, the conveyed data items will also

8. Decision

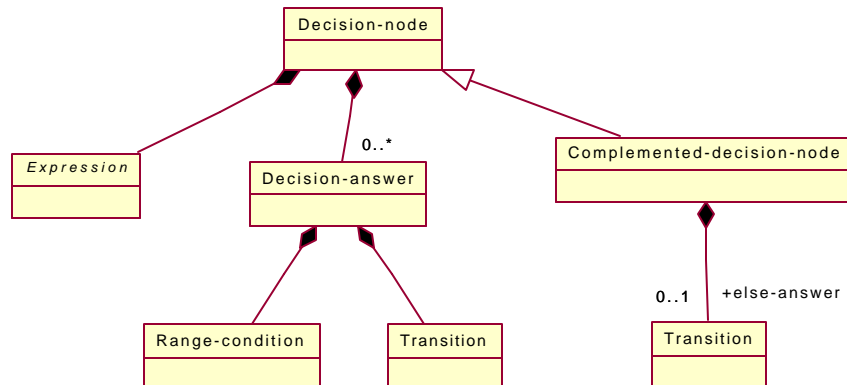


Figure 89. Abstract Syntax for Decision-node

A decision transfers the interpretation to *Transition* of the *Decision-answer* whose *Range-condition* contains the result given by the interpretation of the *Expression*.

In a *Complemented-decision-node*, one of the answers may be the complement of the others. This is achieved by specifying the *else-answer*, which indicates the set of activities to be performed when the result of the *Expression* is not covered by the results specified in any *Decision-answer*.

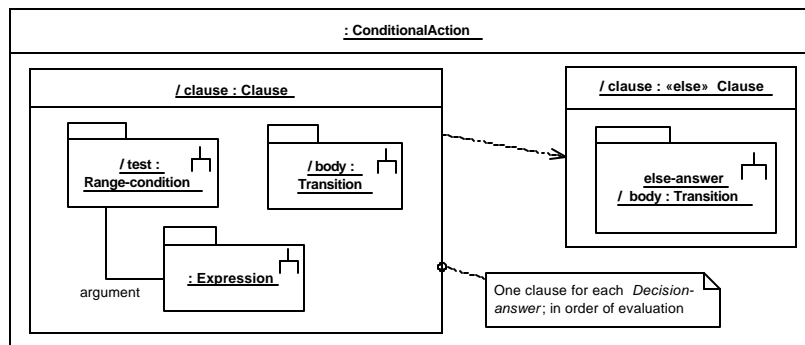


Figure 90. Action Semantics for Complemented-decision-node

Whenever the *else-answer* is not specified, and the result from the evaluation of the question expression does not match one of the answers, then the predefined exception *NoMatchingAnswer* is raised.

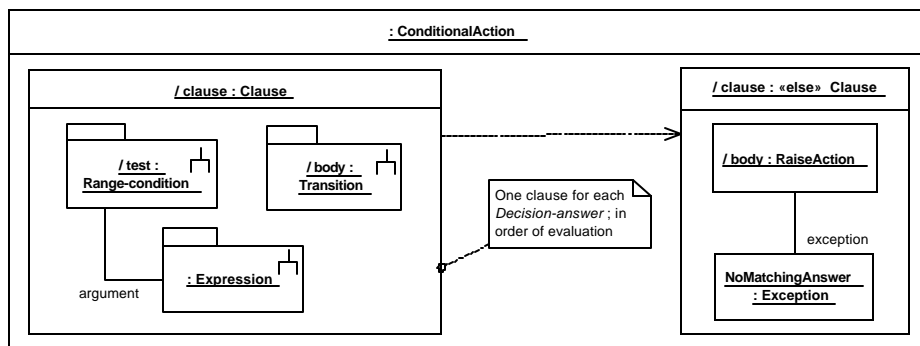


Figure 91. Action Semantics for Decision-node

9. Compound Statement

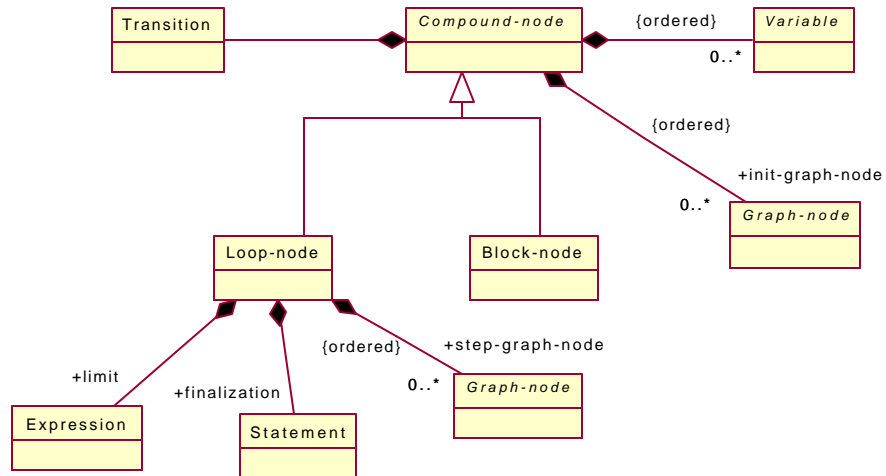


Figure 92. Abstract Syntax for Compound-node

A *Compound-node* creates its own scope. The interpretation of a *Compound-node* proceeds as follows:

- a) A local variable is created for each *Variable*.
- b) The list of *init-graph-nodes* is interpreted.
- c) The *Transition* is interpreted.
- d) When the interpretation of the *Compound-node* terminates, all variables created by the interpretation of the *Compound-node* will cease to exist. Interpretation of a *Compound-node* terminates:
 - i) when a *Break-node* is interpreted; or
 - ii) when a *Continue-node* not associated with this *Compound-node* is interpreted; or
 - iii) when a *Return-node* is interpreted; or
 - iv) when an exception instance is created that is not handled within the *Transition* of the *Compound-node*.
- e) Hereafter, interpretation continues as follows:
 - i) If the interpretation of the *Compound-node* terminated due to the interpretation of a *Break-node* associated with this *Compound-node*, then interpretation continues at the node following the *Compound-node*; otherwise
 - ii) if the interpretation of the *Compound-node* terminated due to the interpretation of a *Break-node*, *Continue-node*, or *Return-node*, then the interpretation continues with interpretation of the *Break-node*, *Continue-node*, or *Return-node*, respectively, at the point of invocation of the *Compound-node* (see Section 6); otherwise

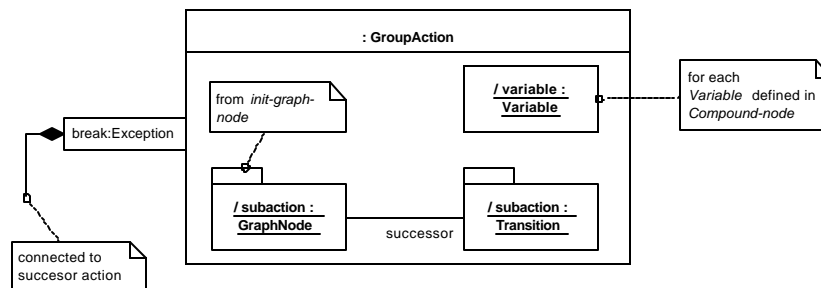


Figure 93. Action Semantics for Block-node

- iii) if the interpretation of the *Compound-node* terminated due to the creation of an exception instance the interpretation continues as described in Chapter 10, "Exception".

The interpretation of a *Loop-node* inserts the following step after step (c) above: When a *Continue-node* associated with the *Loop-node* is interpreted, the list of *Step-graph-node*s is interpreted and further interpretation continues at step (c).

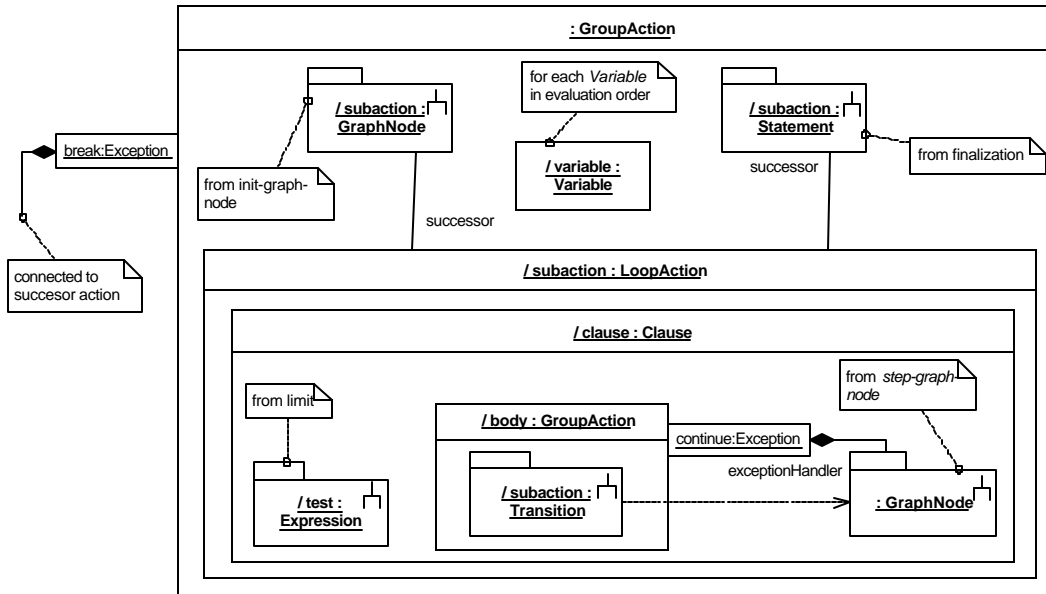


Figure 94. Action Semantics for Loop-node

10. Exception

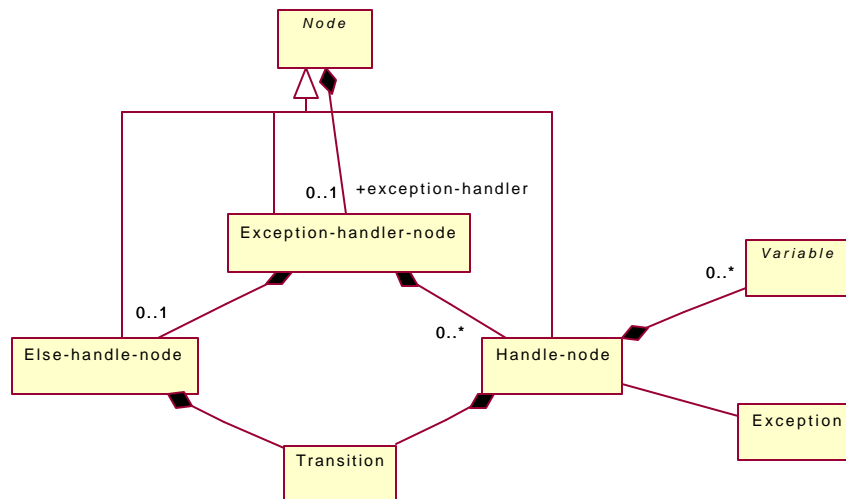


Figure 95. Abstract Syntax for Exception-handler-node

An exception handler represents a particular condition in which a procedure may handle an exception instance that it has created. Handling an exception instance results in the interpretation of a *Transition*.

If the *Exception-handler-node* has no *Handle-node* associated with the exception instance, the exception instance is caught by the *Else-handle-node*. If there is no *Else-handle-node*, the exception instance is not handled in that exception handler.

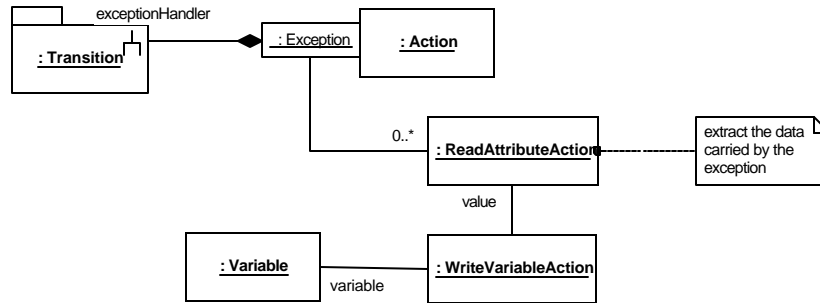


Figure 96. Action Semantics for Exception-handler-node

11. Passive Expression

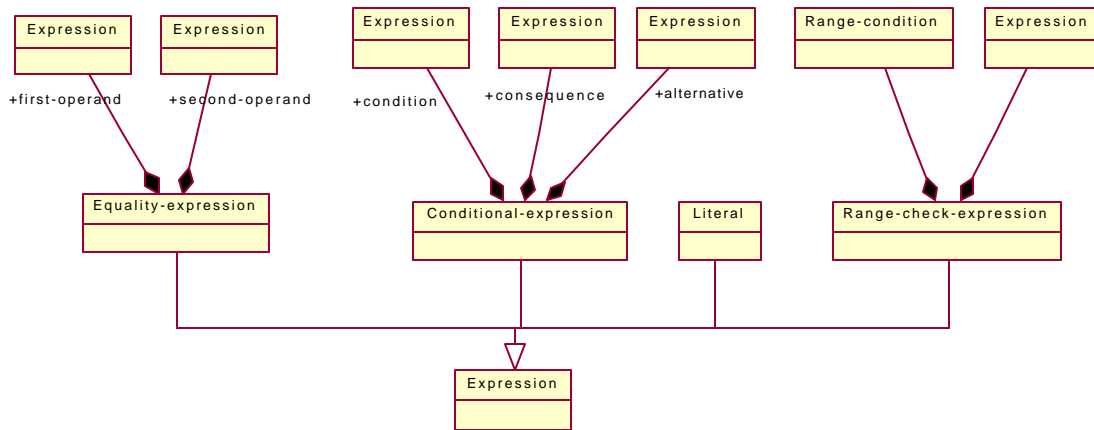


Figure 97. Abstract Syntax for Passive Expressions

11.1 Equality

Interpretation of the *Equality-expression* proceeds by interpretation of its *First-operand* and its *Second-operand*. If, after interpretation, both operands are objects, then the *Equality-expression* denotes reference equality. It returns the predefined Boolean value true if and only if both operands are either Null or reference the same value. If, after interpretation, either one of the operands is a value, the *Equality-expression* denotes equality of values.

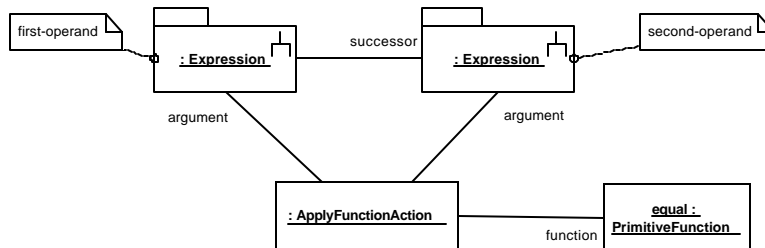


Figure 98. Action Semantics for Equality-expression

11.2 Conditional

A conditional expression represents an *Expression* that is interpreted as either the *Consequence-expression* or the *Alternative-expression*. If the *Boolean-expression* returns the predefined Boolean value true then the *Alternative-expression* is not interpreted. If the *Boolean-expression* returns the predefined Boolean value false then the *Consequence-expression* is not interpreted. The result of the conditional expression is the result of interpreting the *Consequence-expression* or the *Alternative-expression*.

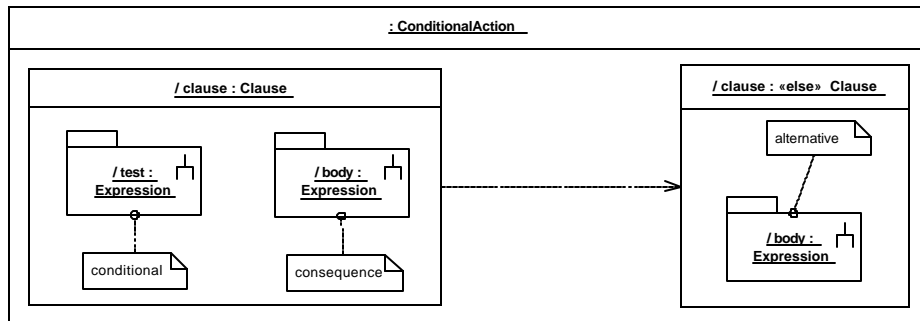


Figure 99. Action Semantics for Conditional-expression

11.3 Literal

A *Literal* returns a unique data item.

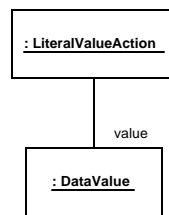


Figure 100. Action Semantics for Literal

11.4 Range check

A *Range-Check-Expression* has the result true if the result of the *Expression* fulfils the *Range-condition* as defined in Z100, §12.1.9.5, otherwise it has the result false.

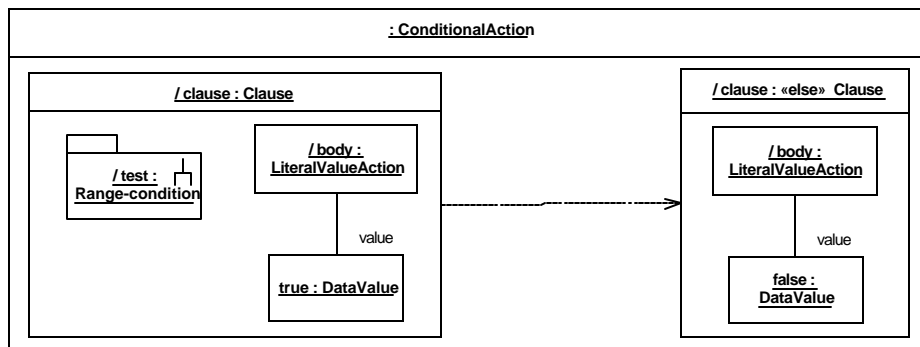


Figure 101. Action Semantics for Range-check-expression

12. Active expression

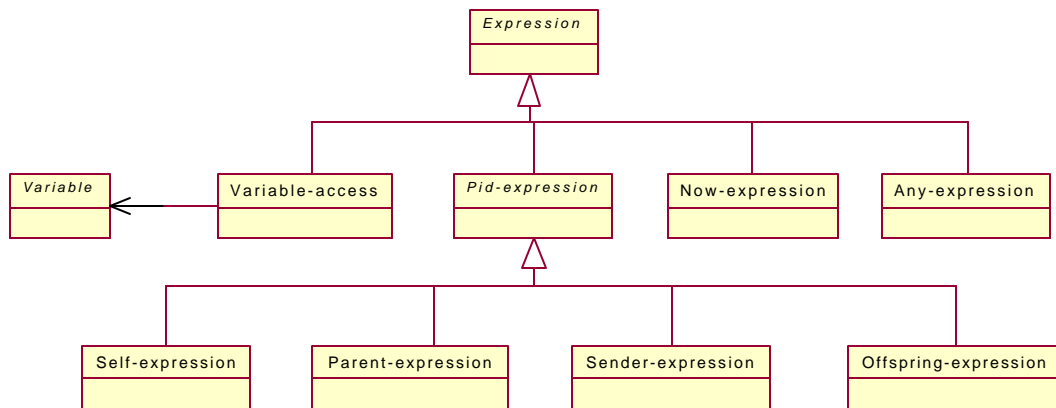


Figure 102. Abstract Syntax for Active Expressions

12.1 Variable access

A variable access is interpreted as giving the data item associated with the identified variable.

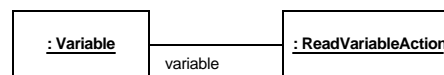


Figure 103. Action Semantics for Variable-access

12.2 Now

The *Now-expression* represents an *Expression* requesting the current value of the system clock giving the time. The origin and unit of time are system dependent. Whether two occurrences of a *Now-expression* in the same transition give the same value is system dependent.

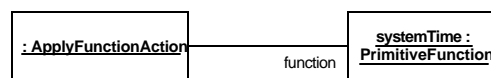


Figure 104. Action Semantics for Now-expression

12.3 Self, Parent, and Offspring

A *Self-expression* accesses the implicit anonymous variable self. Correspondingly, other implicit anonymous variables are accessed by a *Parent-expression*, *Sender-expression*, and *Offspring-expression* (see Section 7). Note that in the UML action semantics, these implicit variables are represented as attributes of the containing process.

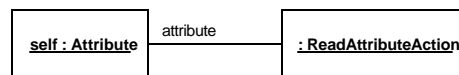


Figure 105. Action Semantics for Self-expression

12.4 Any

An *Any-expression* returns an unspecified element of the associated sort, if that sort is a value sort. If that sort is an object sort, the *Any-expression* returns Null.

Note. The semantics of the *Any-expression* is beyond the scope of the UML action semantic.
