DDL: An Object Oriented Design Description Language

Sergio E. R. de Carvalho*

email:sergio@inf.puc-rio.br

PUC-RioInf.MCC29/97

Abstract

Object-oriented designs are usually expressed graphically, via static and dynamic diagrams. When translated into executable code, diagram characteristics are lost in low-level programming language details. To prevent discontinuities, designs should be expressed in languages textually describing the same discourse. This report presents one such language, DDL, being used intermedially in a CASE tool allowing the automatic generation of C++ code. This graphical-textual match furthers project documentation, correctness verification, and traceability, for example.

Keywords: object-oriented language, design description, design-program correspondence

Resumo

Projetos orientados a objetos são tipicamente representados graficamente, via visões estáticas e dinâmicas. Quando traduzidos para linguagens de programação, aspectos são perdidos nos muitos detalhes que estas linguagens impõem. Para evitar descontinuidades, diagramas de projeto devem ser representados em linguagens expressando o mesmo discurso. Este relatório descreve uma destas linguagens, DDL, sendo usada de forma intermediária em uma ferramenta CASE que permite a geração automática de código C++. A correspondência entre as linguagens gráfica e textual permite uma melhor documentação de projetos, verificação de correção, e recuperação de êrros.

Palavras chave: linguagem orientada a objetos, descrição de projetos, correspondência projeto-programa

* This work was supported by CNPq, and partially sponsored by the project ARTS (formal Approach to Real-Time Systems development) being carried out by The Laboratory of Formal Methods, Department of Informatics, Pontifícia Universidade Católica do Rio de Janeiro, Brazil, and its research associates, under contract to Equitel S.A. Equipamentos e Sistemas de Telecomunicações.

1. Introduction

The construction of a software system typically evolves from the definition of its objectives, usually represented in natural language, to its implementation in a programming language. Many intermediate system representations may exist between these two extremes. During design, for example, object-oriented systems may be represented graphically, via some CASE tool, and then textually, prior to their transformation to executable code.

Ideally, graphical and textual design languages express the same discourse. This facilitates automatic phase to phase transformations with verification of consistency, traceability, and reuse, for example. Discontinuities are also minimized. When this is not the case, considerable effort must be spent on error-prone activities until executable code is obtained.

This document describes DDL, a textual Design Description Language for object-oriented systems. Using DDL, system designers can express high-level code corresponding to their designs; or, alternatively, CASE tools can generate DDL code corresponding to graphical system representations. Programmers can read this code and create corresponding code in an object-oriented language of their preference; or, alternatively, transformation systems can do this job, fully automating the software construction process.

DDL has mechanisms for the representation of classes and their relationships, for object life-cycles, scenarios and object communication in general. It also contains features not usually found in object-oriented systems today, as several semantics for object behavior, the notion of object discipline, and the definition of constant objects.

DDL is a pure, second generation object-oriented language. A DDL program contains code for all classes defined in a project, expressing their contents and relationships. DDL is currently inserted in a CASE tool, where method independent designs can take place. DDL can also be used directly as a programming language, since it contains (with the exception of expressions) syntax and semantics for object-oriented language constructs.

The definition of DDL took into account simplicity and familiarity. The language is small, and most of its object-oriented constructs can be explained in terms of knowledge already in possession of most conventional programmers today. For example:

- classes in DDL are just like types in conventional languages, compile-time entities modeling objects as types model variables;
- the methodical behavior of objects is represented by class contained procedures, which although bound to receiving objects at execution time, have the same semantics as those found in Pascal or C;
- the data structure related to objects at execution-time is that of records in conventionallanguages.

2. Overview

This section presents an overview of the language, introducing programs, classes and built-in class relationships, and the behavior of objects. Section 3 presents the syntax and semantics of DDL.

Program. A DDL program is a collection of possibly nested classes, declared in the program or brought in from libraries. There is no external referencing environment (no main program, external data or functions).

Class collections can be constructed for execution or for storage in libraries. An executable program must have at least one outermost located class containing a procedure called "Main". One of these classes, indicated via the execution environment, is the main class in the program. Execution begins with the application of "Main" to an automatically created object of the main class. On the other hand, a class library does not need (but may contain) one or more classes containing "Main" procedures.

Class. Classes model objects, indicating to the compiler their structure and behavior. A class declaration may optionally contain, as sections:

- *inheritance*: simple, creating a super-type/sub-type relationship, used by the compiler to generate robust code;
- *nesting*: a class A may be contained in a class B, having full visibility of its features, and being protected from accesses external to B;
- *object structure*: a record of attribute declarations, each containing the name of an attribute and its modeling class;
- *common structure*: another record, statically allocated, unique and visible to all class objects, through class operations;
- *state component*: a special object component, modeling the state dependent behavior of class objects;
- *object behavior*: declarations of the operations applicable to class objects, plus the declarations of messages and exceptions that class objects may send or propagate.

Classes can be concrete, abstract, generic. Abstract classes allow the declaration but not the creation of objects, since at least one of its operations is not fully defined. These classes are useful in inheritance hierarchies, promoting polymorphism. Concrete classes may inherit abstract classes (but not vice-versa), implementing incomplete operations, and thus allowing the creation of objects. Generic or parameterized classes are class models, which can be instantiated with argument classes.

Inheritance. DDL inheritance is simple, relating a superclass to a subclass, and promoting the reuse of superclass sections. Every subclass modeled object is also a superclass object. Operations may be redefined in subclasses, as long as their interfaces (name, parameters, result) are the same. Polymorphic superclass objects can assume, at execution-time, the structure and behavior of subclass objects.

Object structure. Corresponding to each object there exists, at execution-time, a block of data for its attributes. At object creation (automatic for stack allocated objects, explicit for polymorphic objects) an initial value is assigned to each object. This value is the ordered collection of its component values, always available, since primitive class objects, such as INTEGER, have pre-defined values.

Object components are automatically protected from external accesses, being visible only to class operations. This protection can be explicitly relaxed by programmers.

Common structure. Each class may contain a statically allocated collection of objects, visible to class (and subclasses thereof) operations. Direct access to this structure is denied to class users, but may be explicitly given by programmer's option.

Object behavior. Objects in DDL can satisfy requests in several different ways: procedurally, answering asynchronous, handshake and future messages, cooperatively, and exceptionally. Although exception handling is always private, these services are typically public, but may be rendered private by programmer's option.

The procedural or methodical behavior is that found in other object-oriented languages: the client's execution is suspended until the procedure terminates. The procedure is late bound to polymorphic objects, and compile-time bound to automatic objects (objects which may not change classes at execution-time).

Objects in DDL may receive asynchronous messages, messages with handshake protocols, or messages with future synchronism. Asynchronous message receiving does not block the client; handshake message receiving blocks the client until the reception is acknowledged; future message clients may self-block to receive parameters or otherwise synchronize with the receiver, which remains blocked at termination until the client reaches the future synchronization point.

Coroutines can also be declared in classes, offering a cooperative behavior for objects. Unlike procedures, coroutines yield execution control without losing their referencing environments, and save their own resume addresses. Iterators are similar to coroutines, but can only be used in loops, to compute, outside the loop's environment, the next value for the loop control object.

Exceptional conditions may arise during program execution. To handle this situation, DDL allows the definition of exception handlers at the operation and class levels. Operation handlers are similar to those in Ada: exceptions may be declared, raised and locally handled. Class level handlers specify another handling scope, being automatically applied to the receiving object if no local handler is defined for some raised exception. Exception handlers are always private.

Objects. Objects may be declared as components of other objects, in the common class area, locally in operations, as operation parameters, and as operation results. They can be collected in containers, such as arrays and lists (objects of instantiated genericclasses).

At declaration, always mandatory, programmers may assign the polymorphic discipline to objects. Such objects may change classes at execution-time (always in the hierarchy rooted by their declaration classes). These objects are heap allocated, and their life cycles are programmer controlled. Automatic objects, on the other hand, are stack allocated, just as Pascal or C variables, and cannot change classes at execution-time.

Certain automatic relationships between objects are created at execution-time: creator-creature, and parent-son. The creator is an object that, servicing a request, creates another object, its creature. The parent is an object that, in its structure declaration, contains the declaration of its son. These relationships are represented in DDL by the generic designators CREATOR and PARENT, respectively. Using these identifiers programmers can increase class decoupling, defining classes that relate to the outside world sending messages to their creators or parents.

3. Syntax and semantics

The description that follows is typical in the presentation of programming languages: syntax is formally given in a BNF-like format, and semantics is described in natural language. In the syntactic description, the following conventions were adopted:

- non-terminal symbols are represented in small letters, with underscores if needed (as for example in object_declaration);
- reserved words are represented in capital letters, with underscores if needed (as for example in LOOP);
- other terminal symbols are enclosed in single quotes (as for example ':=');
- the symbol **->** means "is defined by";
- the symbol **|** indicates an alternative description;
- the symbol **opt** indicates an optional definition;
- the symbol **repeat** indicates a repeated definition, without separators;
- the symbol **list** indicates a repeated definition, separated by commas;
- the symbols **[** and **]** are used to give scope to the **opt**, **repeat**, and **list** modifiers;
- the symbol + indicates at least one instance of the following non-terminal.

The reserved words in DDL (and their abbreviations) are the following:

- object behavior semantics:
  - ASYNCHRONOUS (ASYNC) -- asynchronous message semantics
  - COROUTINE (COR) -- cooperative behavior
  - EXCEPTION_HANDLER (EHANDLER) -- exception handler
  - FUTURE -- future message semantics
  - HANDSHAKE (HANDS) -- synchronous message semantics
  - ITERATOR (ITER) -- cooperative behavior for iterations
  - MESSAGE_HANDLER (MHANDLER) -- message handling
  - PROCEDURE (PROC) -- procedural behavior

- built-in basic classes:
  - BOOLEAN (BOOL)
  - BYTE
  - CHARACTER (CHAR)
  - INTEGER (INT)
  - REAL
  - STRING

- class and class section definitions:

CLASS	-- class definition
COMMON	-- class structure definition
CONSTANT (CONST)	-- constant class objects
CONTAINS	-- class nesting
END	-- end of definition
EXCEPTION	-- exceptions that may be caused by class objects
INHERITS	-- superclass definition
MESSAGE	-- messages that may be sent by class objects
OBJECT	-- object structure definition
PUBLIC	-- visibility specification
PRIVATE	-- visibility specification
STATE	-- state behavior object component

- pre-defined class operations:

CLONE	-- shallow object copy
COERCE	-- object reincarnation in another family class
CREATE	-- object creation
DISPOSE	-- object access elimination
IS_CLONE	-- shallow copy test
IS_NULL	-- initialization test
IS_SAME	-- same access test
PROMOTE	-- object specialization in subclass
WAIT	-- future synchronism

- statements:

CASE	-- case selection
ELSE	-- alternative
END	-- statement end
EXCEPT	-- local exception handler
EXIT	-- loop termination
FROM	-- message sender identification
IF	-- if selection
LOOP	-- repetition
QUIT	-- message handler terminator
RAISE	-- exception generator
REPEAT	-- loop iteration terminator
RETURN	-- procedure terminator
RETURNS	-- return object
THEN	-- if clause
WHEN	-- CASE, EXCEPT clause
YIELD	-- coroutine control flow
YIELDS	-- object yielded by coroutine

- parameter passing:

IN	-- from argument to parameter
OUT	-- from parameter to argument
REFERENCE (REF)	-- access to argument

- array size:
  GET_LIMIT -- returns number of elements in a dimension
  SET_LIMIT -- defines the number of elements in a dimension

- object discipline:
  POLYMORPHIC (POLY) -- heap allocation, class change at execution time

- pronouns:
  CREATOR -- the creating object
  CURRENT -- the current object in the navigation of a data structure
  PARENT -- the object where a component is defined
  SELF -- the receiving object

- initial value of polymorphic objects:
  NULL

A project description is a collection of possibly nested classes. There is no global referencing environment, nor external functions. To be executable, this collection needs at least one non-nested class containing a procedure called Main. This is the main class in the program; execution starts with the application of Main to an automatically created object of the main class. A class collection can also model a resource, being compiled into a library.

program **->**
    [repeat class_declaration **]**

A class declaration needs a class specifier (a name, unless the class is generic), and optionally may contain inheritance and nesting clauses, and a class body.

class_declaration **->**
    CLASS class_specifier ';'
        [opt inheritance_clause ';' **]**
        [opt contains_clause ';' **]**
        [opt class_body **]**
    END CLASS

class_specifier **->**
    class_name **[opt** class_parameters **]**

class_name **->**
    identifier

class_parameters **->**
    '<' **[list** class_parameter+ **]** '>'

Generic parameters may be restricted as in T IN Student, *which validates as arguments any subclass of* Student.

class-parameter **->**
    identifier **[opt** argument_ family **]**

argument_family **->**
    IN class_name

Concrete and abstract generic classes may be inherited. A subclass of a generic class may introduce more generic parameters, beyond those inherited. Inherited parameters have the same identifier as in the superclass, as in
    CLASS ListofTwo <T1> INHERITS List <T2>;.
To create an instance of ListofTwo, two arguments are necessary, as for example in
    ListofTwo <Student, Course> lsc;.

inheritance_clause **->**
    INHERITS class_specifier

Classes of any nature may be statically nested. To improve the readability of programs, the nesting class specifies all nested classes, which are however defined outside the scope of the nesting class. A nested class is only visible within its nest, being protected from external accesses, and has total visibility of the nesting environment.

contains_clause **->**
    CONTAINS **[list** class_specifier+ **]**

A class body is a collection of optional components, not necessarily defined in the order presented below. The common structure *is a statically created referencing environment, visible to all class objects (and objects of their subclasses) via class operations. The* object structure *defines object components. The* class state *component is responsible for the state dependent behavior of objects, being modeled by the root class of an automatically created class hierarchy. The* message section *lists all messages that class objects may send to their parents or creators. Class* exceptions *indicate possible abnormal results that class operations may send to client objects. Class* constants *are objects with immutable component values. Finally, class* operations *describe object behavior, with several possible semantics.*

class_body **->**
    [opt common_structure **]**
    [opt object_structure **]**
    [opt class_state **]**
    [opt class_messages **]**
    [opt class_exceptions **]**
    [opt class_constants **]**
    [opt object_behavior **]**

The environment created by the common structure cannot be directly accessed by class users, but is visible to subclass constructors. However, visibility (of the entire section or just of selected components) may be granted to users by programmer's choice.

common_structure **->**
    [opt PUBLIC **]** COMMON
        [repeat field_declaration+ **]**
    END COMMON

The internal object representation cannot be directly accessed by class users, but is visible to subclass constructors. However, visibility (of the entire section or just of selected components) may be granted to users by programmer's choice.

object_structure ->
    [opt PUBLIC **]** OBJECT
        [repeat field_declaration+ **]**
    END OBJECT

Some objects may have their behavior affected by their current state. This possibility is indicated by the declaration of a special component, modeled by the root class of a hierarchy modeling possible object states. This object may change classes (states) at execution-time, always within its state family.

class_state **->**
    STATE class_name POLYMORPHIC object_name ';'

Components may be declared in the common structure and in the object structure sections, where their user visibility may be defined.

field_declaration **->**
    [opt PUBLIC **]** object_declaration ';'

Objects may be declared in the common structure and object structure sections, locally in class operations, in parameter lists and as operation results. All objects must be declared, where they acquire a base class and a discipline.

object_declaration **->**
    modeling_class_name **[list** modeled_object+ **]**

modeling_class_name **->**
    primitive_class_name
    | class_instance

primitive_class_name **->**
    BOOLEAN
    | BYTE
    | CHARACTER
    | INTEGER
    | REAL
    | STRING

A class instance is an already defined class, residing in a library or created in a program. This class can be concrete, abstract, or a generic class instance.

class_instance **->**
class_name **[opt** class_arguments **]**

class-arguments **->**
‘<’ **[list** class_instance+ **]** ‘>’

Objects are declared with a discipline, which may be automatic (the default) or polymorphic. Automatic objects are automatically created, exist in the execution stack, and cannot change classes at execution-time. On the other hand, polymorphic objects must be explicitly created, are heap allocated and may change classes at execution-time. Automatic objects are always initialized, either automatically (every class has an initial value for its objects), or explicitly with the value of a class constant. Polymorphic objects, once created, also have an initial value, and their creation can be tested via the special function IS_NULL.

modeled_object **->**
[opt POLYMORPHIC **]** object_name **[opt** object_initialization **]**

object_name **->**
identifier

object_initialization **->**
‘:=’ initial_object_value

Literals must be used in the explicit initialization of primitive class objects. Class constants may be used in the initialization of objects modeled by programmer declared classes.

initial_object_value **->**
literal
| constant_name

Every object must be declared, and polymorphic objects must be created. Objects declared in the object structure section of a class have a parent*, and created objects have a* creator*. Parent and creator objects may receive messages from their sons and creatures. The message section of a class lists these messages.*

class_messages **->**
MESSAGE
[repeat message_declaration+ **]**
END MESSAGE

message_declaration **->**
message_name **[opt** message_parameters **]** ‘;’

message_name **->**
identifier

message_parameters **->**
‘(‘ **[list** message_parameter+ **]** ‘)’

message_parameter **->**
modeling_class_name **[opt** POLYMORPHIC **]**

Exceptions are implicitly or explicitly caused by unstable objects, and can be handled in the operation where raised (local handling) or by special operations in the class modeling the unstable object. In any case the operation terminates and the exception ceases to exist; to the client object it didn't even occur. If no local or class handlers are defined, the exception is propagated to the client, or raised in its environment. All exceptions that may propagated by a class must be declared in a special class section.

class_exceptions ->
EXCEPTION
[repeat exception_declaration+ **]**
END EXCEPTION

exception_declaration **->**
identifier ‘;’

Objects with immutable values may be declared in classes, and are implicitly visible to class users. Their identifiers become reserved words, and can be used in assignments and comparisons to variable class objects. Superclass constant objects may be used in the definition of subclass constants.

class_constants **->**
[opt PRIVATE **]** CONSTANT
[repeat constant_declaration+ **]**
END CONSTANT

constant_declaration **->**
[opt PRIVATE **]** constant_name ‘=’ constant_values ‘;’

constant_name **->**
identifier

constant_values **->**
[opt superclass_constant **]** ‘(‘ **[list** constant_field+ **]** ‘)’

superclass_constant **->**
identifier ‘,’

constant_field **->**
field_name ‘=’ initial_object_value

field_name **->**
identifier

Objects behave answering client or own requests. In DDL this behavior can be procedural, cooperative, exceptional or message related. The exceptional behavior is automatically applied to an unstable object, when an exception is raised and no local handler is found. Class clients know the exceptions that may be propagated and should provide their own handling.

object_behavior **->**
    [repeat class_operation+ **]**

class_operation **->**
    procedure_declaration
    |message_handler_declaration
    |exception_handler_declaration
    | coroutine_declaration
    | iterator_declaration

Procedures are like methods in the usual object-oriented languages, early bound to automatic objects and late bound to polymorphic objects. Procedural semantics is that of conventional languages, blocking the client until termination. Procedures may be private. Objects of any class may be procedure parameters or return objects.

procedure_declaration **->**
    procedure_signature **[opt** procedure_body **]**

procedure_signature **->**
    [opt PRIVATE **]** PROCEDURE procedure_name
    [opt procedure_parameters **]**
    [opt return_object_declaration **]** ';'

procedure_name **->**
    identifier

procedure_parameters **->**
    '(' **[list** procedure_parameter+ **]** ')'

Automatic parameters correspond to their arguments via assignment; polymorphic parameters via polymorphic transformation. The IN and OUT modes express the direction of the correspondence, and also variance (always from subclass to superclass). Both cause the local creation of an object. The REF mode allows references to automatic and polymorphic arguments; REF parameters are object identifiers.

procedure_parameter **->**
    procedure_parameter_mode modeling_class_name
    [opt object_discipline **]** parameter_name

procedure_parameter_mode **->**
    IN **|** OUT **|** REFERENCE

parameter_name **->**
    identifier

return_object_declaration **->**
    RETURNSmodeling_class_name
    [opt POLYMORPHIC **]** return_name

return_name **->**
    identifier

procedure_body **->**
    [opt local_declarations **]**
    BEGIN
    [repeat statement+ **]**
    [opt exception_handling **]**
    END PROCEDURE

local_declarations **->**
    [repeat local_declaration+ **]**

Local operation environments may contain the declaration of exceptions, those that can be raised and handled locally.

local_declaration **->**
    object_declaration
    | exception_declaration

exception_declaration **->**
    EXCEPTION **[list** exception_name+ **]** ';'

Objects may respond to synchronous (handshake and future) and asynchronous messages. Message handlers define this behavior.

message_handler_declaration **->**
    message_handler_signature
    [opt sending_objects **]**
    [opt message_handler_body **]**

message_handler_signature **->**
    [opt PRIVATE **]** mhandler_mode MESSAGE_HANDLER handler_name
    [opt message_parameters **]** ';'

In the asynchronous mode, the sender is not blocked; in the handshake mode, the sender and receiver wait on one another until the message can be received; in the future mode, the client may self-block to wait for server results.

mhandler_mode ->
    ASYNCHRONOUS
    | FUTURE
    | HANDSHAKE

handler_name ->
    identifier

Handshake and asynchronous messages may have only input parameters. Future handlers may also have output parameters.

message_parameters ->
    '(' **[list** message_parameter+ **]** ')'

message_parameter ->
    IN modeling_class_name **[opt** POLYMORPHIC **]** parameter_name
    | OUT modeling_class_name **[opt** POLYMORPHIC **]** parameter_name

Objects are automatically related to their CREATORs and PARENTs. A FROM clause allows the identification of the sender in the handler's signature. The absence of this clause identifies messages that can be received from any objects with visibility to the receiving object.

sending_objects ->
    FROM **[list** object_name+ **]**

message_handler_body ->
    [opt local_declarations **]**
    BEGIN
    [repeat statement+ **]**
    [opt exception_handling **]**
    END MESSAGE_HANDLER

Class level exception handlers are automatically applied to the receiving object, if no local handler is available in the raising operation. When applied, the raised exception disappears.

exception_handler_declaration ->
    EXCEPTION_HANDLER handler_name ';'
    [opt exception_handler_body **]**

exception_handler_body ->
    [opt local_declarations **]**
    BEGIN
    [repeat statement+ **]**
    [opt exception_handling **]**
    END EXCEPTION_HANDLER

Coroutines retain their execution environments when yielding control, and save their resume addresses to be used when control is yielded back.

coroutine_declaration **->**
    coroutine_signature
    [opt coroutine_body **]**

coroutine_signature **->**
    [opt PRIVATE **]** COROUTINE coroutine_name
    [opt coroutine_parameters **]**
    yield_object_declaration ‘;’

coroutine_name **->**
    identifier

coroutine_parameters **->**
    ‘(‘ **[list** coroutine_parameter+ **]** ‘)’

To avoid side-effects, coroutines can only have input parameters, passed when the coroutine is first activated.

coroutine_parameter **->**
    IN modeling_class_name **[opt** POLYMORPHIC **]** parameter_name

yield_object_declaration **->**
    YIELDS modeling_class_name **[opt** POLYMORPHIC **]** yield_name

yield_name **->**
    identifier

coroutine_body **->**
    [opt local_declarations **]**
    BEGIN
    [repeat statement+ **]**
    [opt exception_handling **]**
    END COROUTINE

Iterators are special coroutines, created to cooperate with loops, producing the next value of the loop control object.

iterator_declaration **->**
    iterator_signature
    [opt iterator_body **]**

iterator_signature **->**
    [opt PRIVATE **]** ITERATOR iterator_name
    [opt iterator_parameters **]**
    yield_object_declaration ‘;’

iterator_name **->**
    identifier

iterator_parameters **->**
    '(' **[list** iterator_parameter+ **]** ')'

To avoid side-effects, iterators can only have input parameters, received during its first application.

iterator_parameter **->**
    IN modeling_class_name **[opt** POLYMORPHIC **]** parameter_name

yield_object_declaration **->**
    YIELDS modeling_class_name **[opt** POLYMORPHIC **]** yield_name

yield_name **->**
    identifier

iterator_body **->**
    [opt local_declarations **]**
    BEGIN
    [repeat statement+ **]**
    [opt exception_handling **]**
    END ITERATOR

DDL statements are similar to those in conventional languages.

statement **->**
    assignment
    | repetition
    | selection
    | application
    | termination
    | exception

Value assignment can only be applied to automatic objects, copying values to the execution stack. The value being assigned can be that of a subclass object. The pronoun SELF designates the receiving object in operation statements.

assignment **->**
    assignment_target ':=' value ';'

assignment_target **->**
    object_name **[opt** field_specification **]**
    | SELF **[opt** field_specification **]**

field_specification **->**
    [repeat field_access+ **]**

field_access **->**
    '.' object_name

DDL does not have syntax for expressions. Multiple values are represented by constant or variable objects.

value **->**
    expression
    | multiple_value

multiple_value **->**
    constant_name
    | assignment_target

The loop statement can be associated to an iterator, which provides the next value for the loop control object.

repetition **->**
    LOOP **[opt** iterator_application **]**
    [repeat statement+ **]**
    END LOOP

DDL has selection statements controlled by conditions (IF) or values (CASE).

selection **->**
    if_statement
    | case_statement

if_statement **->**
    IF expression
    THEN **[repeat** statement+ **]**
    [opt elsif_options **]**
    [opt else_option **]**
    END IF

elsif_options **->**
    [repeat elsif_option+ **]**

elsif_option **->**
    ELSIF expression THEN **[repeat** statement+ **]**

else_option **->**
    ELSE **[repeat** statement+ **]**

case_statement **->**
    CASE expression
    [repeat when_option+ **]**
    [opt else_option **]**
    END CASE

when_option **->**
WHEN **[list** initial_object_value+ **]** THEN **[repeat** statement+ **]**

Application statements invoke an operation on a receiving object. This object may be referenced by name (if visible) or by pronoun (using the relationships automatically created). Superclass operations may be applied within subclass operations, if qualified with the superclass' name.

application **->**
application_target '<-' **[opt** modeling_class **]**
operation_name **[opt** arguments **]** ';'
| application_target '<-' predeclared_operation ';'

application_target **->**
object_name **[opt** field_specification **]**
| SELF **[opt** field_specification **]**
| PARENT
| CREATOR
| CURRENT

SELF cannot receive the pre-defined operations CREATE, DISPOSE, COERCE and PROMOTE; PARENT and CREATOR can only receive messages, and may not be used as arguments in operation applications.

modeling_class**->**
class_name ':'

operation_name **->**
identifier

arguments ->
'(' **[list** argument + **]** ')'

argument **->**
expression
|initial_object_value
| NULL

predeclared_operation **->**
[opt modeling_class **]** CREATE **[opt** '(' constant_name ')' **]**

Applicable to polymorphic objects. Creates a heap space for the receiving object. The optional constant is used to initialize the receiving object, with a value different than INITIAL. The modeling class at creation can be a subclass of the declaration class.

| DISPOSE

Applicable to polymorphic objects. Eliminates the association between identifier and object, without necessarily making the receiving object's space available. After the application, the receiving object IS_NULL.

| COERCE '(' object_name **[opt** field_specification **]** ')'
| modeling_class COERCE

Applicable to polymorphic objects within their families, forcing their modeling by some family class. If prefixed with a class name, associates the receiving object's identifier with a new object in the target class.

| PROMOTE '(' object_name **[opt** field_specification **]** ')'
| modeling_class PROMOTE

Applicable to polymorphic objects within their families, forcing their modeling by some descendant class. If prefixed with a class name, associates the receiving object's identifier with a new object in the target class.

| IS_SAME '(' object_name **[opt** field_specification **]** ')'

Applicable to polymorphic objects. Returns TRUE if the receiving object accesses the argument, otherwise returns FALSE.

| CLONE '(' object_name **[opt** field_specification **]** ')'

Applicable to polymorphic objects. Creates a shallow copy of the argument and associates it with the receiving object.

| IS_CLONE '(' object_name **[opt** field_specification **]** ')'

Applicable to polymorphic objects. Returns TRUE if the receiving object is a shallow copy of the argument, otherwise returns FALSE.

| IS_NULL

Applicable to polymorphic objects. Returns TRUE if the receiving object has not been created or has been disposed, otherwise returns FALSE.

| WAIT '(' object_name **[opt** field_specification **]** ')'

Applicable to objects that sent future messages to the argument, blocking these objects until synchronism occurs, with the passing of output parameters, if any.

EXIT statements terminate repetitions (and the associated iterators, if they exist); REPEAT statements initiate a new iteration (re-invoking the associated iterator, if it exists); QUIT statements terminate message handlers, and YIELD statements yield control to coroutine clients.

termination **->**
    EXIT **[opt** WHEN expression **]** ‘;’
    | REPEAT **[opt** WHEN expression **]** ‘;’
    | RETURN **[opt** WHEN expression **]** ‘;’
    | QUIT **[opt** WHEN expression **]** ‘;’
    | YIELD **[opt** WHEN expression **]** ‘;’

The RAISE statement raises an exception known in the scope.

raise **->**
    RAISE exception_name ‘;’

Local exception handling may be specified in operations.

exception_handling**->**
    EXCEPT **[repeat** exception_case+ **]**
    END EXCEPT

exception_case **->**
    WHEN **[list** exception_name+ **]** THEN **[repeat** statement+ **]**