OMEGA

Correct Development of Real Time Systems

Title : Action specification in OMEGA

Author(s) : VERIMAG, OFFIS, CAU

Editor : Verimag

Date : 04/03/2004

Identifier : IST/33522/M2.2.1

Document Version : 3-a4

Status : Under work

Confidentiality : Protected

Abstract : This document is composed of two parts:

Part 1 gives the rationale behind the decision of the OMEGA consortium to define an OMEGA Action Language (OMAL). It discusses the various choices possible for representing actions in UML models. This issue exists because the actions do not have a common representation in various UML tools.

Part 2 defines the OMEGA Action Language: its principles, lexical and syntactic rules, examples.

Annex 1 lists some issues that have been raised by users/implementers of OMAL. The issues are either solved or to be solved in a future version of OMAL.

Ileana Ober
VERIMAG
Ileana.ober@imag.fr

Document history

Revision	Date	Author	Comments
1	05/12/2002	Ileana Ober	Preliminary version presented in Amsterdam (dec. 2002)
2	15/12/2002	Ileana Ober	Version including general remarks from the Amsterdam meeting
3	03/2003	Ileana Ober	Update after feed-back from OFFIS Added section *Plugging-in the action specification into UML models* and changes in the syntax
3-a1	02/2004	Iulian Ober	A list of issues to be addressed in the definition of the AL is added as Annex 1
3-a2	02/2004	IO, AV	Comments from VMG and OFFIS
3-a3	02/2004	IO	Started to modify spec according to comments
3-a4	03/2004	Everybody	Corrected during Paris meeting

Table of contents

1 Rationale for defining an OMEGA Action Language[1]

1.1 Why do we need to interchange actions?

UML offers the possibility to describe both the static structure of a system, and its behaviour. One of the building blocks of dynamic behaviour description is the action. An action can describe the body of an operation, or it specifies the actual behaviour on a transition or within a state (at entry or exit).

A UML model interchange that excludes *actions* means to exchange models including only explicit control flow, losing thus not only all data transformations, but also all signal emissions (which are part of a SendAction), all actual operation calls (part of the CallAction) and object creation and destruction.

If we need to export a complete UML model, this export shall take into account the actions (as they contain all the details related to behaviour). Therefore, in order to have truly interchangeable models we need to be able to somehow interchange actions. Presently, in all tools exporting UML models in XMI format actions are non structured, uninterpreted strings. As long as the XMI models do not provide an *abstract syntax* for actions we must agree on a *concrete syntax* for actions in order to be able to exchange models between different tools.

1.2 The choice of a programming language

One choice is to use a programming language (the target language of code generation) for representing actions. This is the choice taken by most of the UML tools.

The data structure (i.e. classes, attributes, and operations) are taken from the UML model, and the programming language is only used for the action part. However, not all UML primitives can be directly represented in a programming language. Typically, signal emission and association navigations cannot be natively represented in programming language constructs.

Tools overpass this difficulty by defining libraries for signal passing, association manipulation, and collections (needed for multiple associations).

None of these libraries is standard, moreover they very much depend on the implementation of the rest of the tool. For instance, for signal emission the same library function is employed as those used for all kinds of notifications within the simulator or code generation.

In conclusion, because the programming languages are not enough powerful to natively express all the kinds of actions needed in a UML specification, and because the libraries used for the parts not covered are, proprietary and tool dependent, the use of a programming language for expressing actions is incompatible with the goal of interchanging actions.

1.3 Current status in UML and UML tools (XMI)

The latest version of the UML standard is called **UML 1.4 with Action Semantics**. The Action Semantics part contains a description of what can be specified in actions at abstract semantics level (i.e. adding new metaclasses to the UML meta-model and connecting them properly with the rest of the UML meta-model). There are constructs to represent basic action primitives: send action, call action, etc. As well as constructs for expressing control and algorithmic. UML tools are supposed to offer concrete syntaxes based on this abstract semantics, and possibly exchange them via the XMI format at abstract syntax level.

[1] This part of the document was written before the OMEGA consortium took the decision to define the AL. Some parts presented as *plans* have been concretized in the meantime.

Unfortunately, *there is no UML tool* (that we are aware of) *supporting and exporting XMI for the action semantics part*. The most common scenario for treating actions in UML tools (including in those present in the project) consists in exploiting the fact that UML allows to have uninterpreted actions and to use some specific syntax for the actions, without respecting the standard abstract syntax. Therefore, instead of offering the abstract syntax tree corresponding to some actions, existing UML tools use plain text, that they interpret as they like.

The entire actions part may change as the UML suffers a major revision. Although there are high chances that the new action semantics is close to the UML 1.4 one, no decision has been taken yet.

Argo, Rational Rose – uses text strings to represent actions. No syntax is proposed, actions are used nowhere.

Rhapsody (as far as we were able to infer from the models provided in the project) uses text strings for actions (i.e. the abstract tree is not deployed). However, in the simulator and code generator a C++ syntax is assumed for actions. For association navigation, signal emission and collection handling some internal library functions are to be used.

Telelogic – uses a SDL-like syntax for actions, but as they do not export XMI, the goal of model interchange is far from being achieved.

1.4 Alternative solutions

1. A possible solution consists in defining an abstract syntax[2] and represent it in the XML obtained by tools. As no tool produces XMI for this part, we would have to do a first transformation on the XMI generated by commercial UML tools. This transformation would consist in replacing the parts of the XMI corresponding to uninterpreted actions. This would be the actual format for interchange within OMEGA.

   The advantages of this approach:

   - real interchange for actions
   - as the actions expressed in the XMI produced by tools are represented as uninterpreted strings that need to be parsed the parsing is done only once

   The drawbacks:

   - we would still have to define a concrete syntax, since it is not acceptable for a UML modeller to write the actions in what would be a quite complicated form of XML.
   - the resulting interchange format is not XMI anymore, with all the consequences resulting from this fact: impossibility to use readily available public libraries able to work on XMI
   - defining the abstract syntax and effectively integrating it into the standard UML abstract syntax (specifically in the UML.dtd) would take a non negligible effort which otherwise could be used for more fundamental issues.

2. A second solution consists in defining a concrete syntax (which can be very much like some programming language, to avoid the definition of yet another language). In this case the interchange format is still XMI, but where we have agreed on for a common syntax for actions.

   The advantages of this approach:

[2] we could also take the one existing in UML 1.4, but it is rather big and there is no tool support for it

- real interchange for actions
- we can still use XMI tools for the resulting exchange format
- XMI models can be fed back to UML tools

The drawbacks

- the parsing of the actions must be done by each user of the interchange format
- we have to agree on a common syntax

In the first six months of the project, we have well evaluated the pros and cons of the different solutions, and finally come to the agreement that the best compromise is to adopt the last solution, that is to agree on a common action syntax.

The remainder of the document contains an updated version of the action syntax elaborated during the first project period.

2 The OMEGA Action Language (OMAL)

2.1 Principles

OMAL is an imperative language conforming to the UML1.4 Action Semantics. The language contains only a minimal set of features, to simplify definition and compilation. For homogeneity and ease of use, we use a subset of OCL as expression language for OMAL.

2.2 Restrictions[3]

Notes concerning the current version of OMAL:

- We distinguish syntactically between expressions containing calls or object creation (*call_expression, create_expression*) and expressions containing only navigation and operators of predefined types (*simple_expression*).
  It is allowed to make an operation call (for model or collection operations) on the result of a navigation, but it is not allowed to further navigate from the result of a call (otherwise than by storing the result of the call explicitly in an attribute).
  This is in order to simplify compilation: expressions with nested calls / object creation may require temporary variables for storing intermediate results during evaluation, which are avoided in our setting.
- We only use a limited subset of OCL, especially in what concerns collection expressions. Currently, we suppose all collections are of type *Sequence*, and we consider the following operations:
  - getAt(i : Integer) – get the i'th member of the collection
  - setAt(i : Integer, object : OclAny) – set the i'th member of the collection to point to object
  - isEmpty(), notEmpty(), and size() – with the usual OCL meaning

  Another restriction concerns navigation with collections: we distinguish syntactically between expressions containing calls or object creation (*call_expression, create_expression*) and expressions containing only navigation and operators of predefined types (*simple_expression*).
  This is in order to simplify compilation: expressions with nested calls / object creation may require temporary variables for storing intermediate results during evaluation, which are avoided in our setting.

[3] These restrictions might be too strong for end-users in an industrial setting, and an industrialized version of OMAL should alleviate them.

2.3 OMAL Syntax

This section contains the lexical and syntactic rules defining OMAL.

2.3.1 Lexical tokens

2.3.1.1 Character set

The action language uses a character set corresponding to the 7 bit ASCII character set. Character encoding (ASCII, UNICODE or other encodings) is left open to tools as long as the character set remains as specified above.
In the following, we denote characters and character sequences between apostrophes (‘’). For designating characters we use the usual symbols or the C escape sequences (for non-printable characters).

2.3.1.2 White spaces

White spaces are defined as: space (‘ ‘), tab (‘\t’) or new line (‘\n’ or ‘\r’).
White spaces may appear between any two tokens in an action specification and are ignored.

2.3.1.3 Comments

There are two forms of comments:
‘/*text*/’ where text is any character sequence not containing the subsequence ‘*/’.
‘//text’ where text is any character sequence containing exactly one new line (‘\n’ or ‘\r’) at the end of the sequence.
‘--text’ where text is any character sequence containing exactly one new line (‘\n’ or ‘\r’) at the end of the sequence.
Comments may appear between any two tokens in an action specification and are ignored.

2.3.1.4 Identifiers[4]

An *identifier* is an unlimited-length sequence of *letters* and *digits*, the first of which must be a letter. Identifiers are denoted in the grammar by the token IDENTIFIER, defined by the following regular expression:

IDENTIFIER	::=	NONDIGIT (NONDIGIT + DIGIT)*
NONDIGIT	::=	'_' + 'A' + … + 'Z' + 'a' + … + 'z'
DIGIT	::=	'0' + '1' + … + '9'

2.3.1.5 Keywords

The following character sequences are reserved for use as *keywords* and cannot be used as identifiers.

BEGIN ::= ‘begin’

CHOOSE ::= ‘choose’

COBEGIN ::= ‘cobegin’

COEND ::= ‘coend’

DO ::= ‘do’

ELSE ::= ‘else’

END ::= ‘end’

ENDCHOOSE ::= ‘endchoose’

ENDIF ::= ‘endif’

[4] From this point on, each lexical definition must be read as follows:
- at the left side of ‘::=’ we write the token name by which the token is identified in the grammar
- at the right side of ‘::=’ we write the character sequence / regular expression defining the token.

ENDWHILE ::= 'endwhile'

IF ::= 'if'

INFORMAL ::= 'informal'

NEW ::= 'new'

REPLY ::= 'reply'

RETURN ::= 'return'

SELF ::= 'self'

THEN ::= 'then'

WHILE ::= 'while'

The following character sequences are reserved for use as *keywords* in future versions of OMAL and cannot be used as identifiers.

LEADSTO ::= 'leadsto'

2.3.1.6 Literals

The following character sequences are reserved for use as literals denoting values of predefined types.

- Boolean literals:

  FALSE ::= 'false'

  TRUE ::= 'true'

- Time literals:

  NOW ::= 'now'

- Integer literals:

  INTEGER_LIT ::= DIGIT (DIGIT)*

- Real literals:

  REAL_LIT ::= DIGIT (DIGIT)* '.' DIGIT (DIGIT)*

- Object reference literals:

  NULL ::= 'null'

- String literals:

  STRING_LIT ::= '"text"'

Where text is any character sequence containing neither new lines ('\n' or '\r') nor '"'.

2.3.1.7 Symbols and other tokens

The following character sequences are tokens to which we do not give names. They are used as such in the grammar:

':' ':=' ',' '::' '.' '(' ')' ';' '!' 'and' 'or' 'not' '='
'<>' '->' '::' '+' '-' '*' '/' '%'

2.3.2 Grammar

2.3.2.1 Meta-language

For defining the grammar of OMAL, we use a simple form of EBNF.

- Non-terminals are identified by strings in slanted style, such as *action*.
- Tokens are identified either through their name (in uppercase, like INTEGER_LIT) or through the precise character sequence defining them (between '', like ':=').
- On the right-hand side of a production :
  - The empty token/nonterminal sequence is denoted by e.
  - Round parentheses ( ) are used to group token/nonterminal sequences

- Optional parts are written between [ ]
- Repetitive parts formed of 0 or plus occurrences of a are written as (a)*
- Alternative between a and ß are written as a | ß

2.3.2.2 Production rules[5]

Actions

```
action                ::=  [ IDENTIFIER ':' ][6]
                           ( elementary_action | control_action | composite_action |
                                 nondet_choice_action | interleaving _action)

elementary_action     ::=  e
                      |    assignment_action
                      |    call_action
                      |    send_action
                      |    return_action
                      |    reply_action
                      |    informal_action

assignment_action     ::=  navigation_expression ':=' expression

call_action           ::=  call_expression

send_action           ::=  navigation_expression '!' signal_name
                           '(' [ simple_expression ( ',' simple_expression )* ] ')'

return_action         ::=  RETURN simple_expression

reply_action          ::=  REPLY operation_name '(' simple_expression ')'

informal_action       ::=  INFORMAL STRING_LIT

control_action        ::=  IF expression THEN action ( ';' action )* ENDIF
                      |    IF expression THEN action ( ';' action )*
                           ELSE action ( ';' action )* ENDIF
                      |    WHILE expression DO action ( ';' action )* ENDWHILE

composite_action      ::=  BEGIN action ( ';' action )* END

nondet_choice_action  ::=  CHOOSE action ( '|' action )* ENDCHOOSE

interleaving_action   ::=  COBEGIN action ( '|' action )* COEND
```

Expressions

```
expression            ::=  call_expression
                      |    create_expression
                      |    simple_expression

call_expression       ::=  navigation_expression '.'
                           [classifer_name '::' ] operation_name
```

[5] The grammar in this form is ambiguous and is neither LL(8) nor LR(8). Tool providers may transform it in the form that is suitable for their compiler technology, provided the language does not get changed.
[6] This is to give names to actions. Named actions are needed in the timed part

```
                                '('  [ simple_expression ( ',' simple_expression )* ]  ')'

create_expression       ::=     NEW classifier_name  [ '::' operation_name ]
                                '('  [ simple_expression ( ',' simple_expression )* ]  ')'

simple_expression       ::=     [ simple_expression 'or' ] and_expression

and_expression          ::=     [ and_expression 'and' ] relational_expression

relational_expression   ::=     [ relational_expression ('<' | '<=' | '=' | '>=' | '>' | '<>' ) ]
                                add_expression

add_expression          ::=     [add_expression ('+' | '-' ) ] mult_expression

mult_expression         ::=     [mult_expression ('*' | '/') ] unary_expression

unary_expression        ::=     [ ( '-' | 'not' ) ] primary_expression

primary_expression      ::=     literal
                        |       navigation_expression
                        |       '(' simple_expression ')'

literal                 ::=     FALSE | TRUE | NOW | INTEGER_LIT | REAL_LIT | NULL

navigation_expression   ::=     (
                                        SELF |
                                        parameter_name |
                                        attribute_name |
                                        association_end_name |
                                        reception_name
                                )
                                ( '.' (attribute_name | association_end_name) )*
                                [
                                        '->' collection_oper_name
                                        '('  [ simple_expression ( ',' simple_expression )* ]  ')'
                                ]

signal_name             ::=     IDENTIFIER
operation_name          ::=     IDENTIFIER
classifier_name         ::=     IDENTIFIER
parameter_name          ::=     IDENTIFIER
attribute_name          ::=     IDENTIFIER
association_end_name    ::=     IDENTIFIER
reception_name          ::=     IDENTIFIER
collection_oper_name    ::=     IDENTIFIER
```

2.4 Some informal notes on static semantics

OMAL is statically type checked. Type conformance is based on strict equality for predefined types (no default conversion is defined). Type conformance for object types is defined in the usual way based on the UML class hierarchy defined by the model in which OMAL actions are used.

2.5 Examples

This section contains some action examples:

- An example of composite action (from class EAP from the OMEGA EADS case study) with assignment, operation call, signal sending:

```
begin
 current_is_ok:=EVBO.Close();
 Cyclics!Anomaly()
end
```

- An example from VERIMAG's BitCounter example:

```
begin
  if (next <> null) then
    self.result :=next.get();
    result:=2*result
  endif;
  if (value) then
    result:=result+1
  endif;
  return result
end
```

2.6 Plugging the actions specification into UML models

Although in the context of the UML metamodel it is quite clear where the actions come into the picture of a UML model, things get more complicated if the concern is tool interoperability. This is basically because various tools, use different nodes in the abstract tree for storing the actions.

Basically there are two places actions may occur in a model specification: in the body of a method (when describing the behaviour corresponding to the operation) and on a state machine transition. We have looked at the two commercial UML tools considered in OMEGA to see how exactly actions can be specified in a UML model and how they are saved in the generated XMI file.

2.6.1 Methods

Rational Rose does not generate Method objects, but only Operations. In OMEGA models, the method body has to be placed in the *Operation->Semantics* text field. This text field is exported as the *Operation.specification* item in XMI.

Rhapsody seems does generate Method objects. In OMEGA models, the method body has to be placed in the *Operation->Implementation* text field. This text field is exported as the *Method.body.body* item in XMI. (Note: *Method.body* yields a *ProcedureExpression* object, *Method.body.body* yields a *String* containing the text of the *Operation->Implementation* field).

Both Rose and Rhapsody provide a field called *documentation* for *Operations*. This could provide the possibility to distinguish concrete and abstract action specifications which may be useful in the context of validation. However, for the moment we have not explored this possibility yet.

2.6.2 Transitions

In general, in OMEGA UML models we consider that

- Transition effect is a single action, as defined by the OMAL non-terminal action.
- Transition guard is a simple expression, as defined by the OMAL non-terminal simple_expression.

In **Rational Rose:**

- The effect may be defined in the *Action text field* (NOTE : a single text line), which can be found in XMI in the following location: *Transition.effect.action->getFirst().name*.
- The guard may be defined in the *Guard text box*, which will put it in XMI in *Transition.guard.expression.body.*

In **Rhapsody** :

- The effect may be defined in the *Action text field*, which can be found in XMI in the following location: *Transition.effect.script.body*.
- The guard may be defined in the *Guard text box*, which will put it in XMI in *Transition.guard.expression.body.*