D2.2.2 Tool set for system verification

Annex 1. OMEGA syntax for users

OMEGA

Correct Development of Real Time Systems

Title :	OMEGA syntax for users
Author(s) :	Marcel Kyas, Joost Jacob, Ileana Ober, Iulian Ober, Angelika Votintseva
Editor :	Verimag
Date :	19/01/2004
Identifier :	IST/33522/D2.2.2-A1
Document Version :	4
Status :	Under work
Confidentiality :	Restricted

Abstract : This document gathers all the notations defined in OMEGA. As OMEGA is defined based on UML **we do not describe** here all UML specific notations, instead we give the differences to the official standard (reference version UML 1.4).
The introduction briefly describes how various notations fit together.

Note : This document was updated during the 5th period to reflect changes in the OMEGA action language and the definition of UML observers.

Document history

Revision	Date	Author	Comments
1	08/07/2003	Ileana Ober	Preliminary version containing skeleton + actions + extensions described in XMI
2	16/07/2003	Ileana Ober Marcel Kyas Joost Jacob	Added time + OCL + Components
3	08/2003	Ileana Ober Marcel Kyas Joost Jacob Angelika Votintseva	Add kernel language section
4	03/2004	Iulian Ober	Updated OMEGA Action Language, added observers

OMEGA syntax for users

1 Introduction

This document is intended to offer a first entry point for using the OMEGA kernel language and the OMEGA extensions. It is not self contained as the semantics of the concepts newly added in OMEGA is NOT contained in this document. Each chapter is an extract, focusing only on syntactic aspects, from a more complete deliverable/milestone.

1.1 Ingredients for making a model OMEGA compliant

Compliance of a UML with the OMEGA model (and implicitly with the OMEGA tools) involves several conditions, which are described in and correspond to the subsequent sections of this document. The main lines are:

- Observing the kernel language restrictions/extensions (Section 2). These are restrictions/extensions over the UML concepts (meta-classes) that are allowed in OMEGA models.
  The syntactic restrictions are there mostly to ensure coherence and the possibility of efficient verification for safety-critical models.
  The syntactic extensions are designed to precise things relating to some UML concepts (e.g. specifically saying if an Operation is Triggered or Primitive, as these have a special semantics defined in OMEGA).
- Using the OMEGA action language for describing the actions of a system. Actions are executed either as the result of a statechart transition being taken by an object, or as the result of a *primitive* operation being called on an object. The action language is designed to bridge the absence of a *standard* concrete syntax for the UML actions.
- Using the OMEGA timed extensions, if the model requires them. These extensions are defined in terms of standard UML extension mechanisms (stereotypes, tagged values) and interpreted comments, so that they do not interfere with tools that are not aware of them.
- Using the component description syntax, if the model requires use of components.
- Observing the OMEGA restrictions/extensions for OCL expressions used in the model.
- Using compatible LCSs on the side of the OMEGA UML model.

1.2 A note on tool usage

In an ideal UML world, where all UML editors follow the UML standard to the letter, this document should be sufficient in order for a user to be able to design OMEGA compliant models.

However, the two tools considered in the project, I-Logix Rhapsody and Rational Rose, do not support all (neither the same) UML concepts (metaclasses and meta-attributes/associations). A simple example is the *isAbstract* meta-attribute of UML classes, which is supported directly in Rhapsody and via a stereotype (*Active*) by Rational Rose. Moreover, the link between the meta-information and the concrete means to edit it via the two graphical editors is not always apparent to the user.

As a result of this state of facts, in order for the users to be able to edit OMEGA compliant UML models with the two aforementioned editors, the OMEGA tool providers must provide additional guidelines as to where and in which format information must be input. Such guidelines are partially present in this document, but they are rather informal for the moment. They will be included in the final documentation of each OMEGA tool.

2 Kernel language

In this section we define the kernel language without time and component extensions. The Omega language is a subset of UML 1.4 with a few tagged values needed for the strict formal semantics.
First, for type usage we will allow only classes (used as references to objects), enumeration, and such predefined types as integer and boolean[1]. These types can be used in the constructor *array*, which must be bounded (static arrays) and corresponds to *MultiplicityRange* in the UML Data Type package. In the Subsection 2.1 we define the Omega kernel language specifying its divergence from the standard UML. Subsection 2.2 describes well-formedness rules as additional restrictions on the usage of the syntactic elements from the kernel model.

2.1 UML 1.4 versus Omega Kernel Language

Here we revise the core notions from UML w.r.t. the Omega kernel language, mentioning differences. Note that we do not support any stereotypes defined in UML 1.4, besides those explicitly specified in the extensions of the kernel model (for time and component specifications).

2.1.1 Structural Elements

2.1.1.1. **Abstraction** (inherits from Dependency relation): not supported.

2.1.1.2. **Artefact** is relevant only for the component extension, thus not considered here (besides model itself).

2.1.1.3. **Association**: standard constraint *xor* and tagged value *persistent* are not considered.

2.1.1.4. **AssociationClass**: not supported.

2.1.1.5. **AssociationEnd**: the values of attributes *aggregation*, *changeability*, *visibility*, *multiplicity*, and *isNavigable* are bound by the rules described in more details in Subsection 2.2.1. The value of attribute *targetScope* is considered only as default value *instance*. The value *package* of attributes *visibility* is not considered. Attribute *qualifier* can be only multiplicity (corresponding to the array index).

2.1.1.6. **Attribute**: no restrictions besides the visibility values: visibility *package* is not supported.

2.1.1.7. **BehaviouralFeature**: no special restriction.

2.1.1.8. **Binding** (inherits from Dependency relation): not supported, being a relation between a template and a model element, since templates are not considered.

2.1.1.9. **Class**: we introduce additional attributes *kind* with values from {*reactive, simple*} and *isCompRoot* with Boolean values in addition to the standard *isActive*, *isRoot* (w.r.t. inheritance) etc. These new attributes can be represented as tagged values.

2.1.1.10. **Classifier**: the tagged value of attribute *persistence* is considered only as *transitory*.

2.1.1.11. **Comment**: no specific restrictions, since comments have no semantics.

2.1.1.12. **Component**: not relevant here.

2.1.1.13. **Constraint**: considered in Section 5.

[1] Type string can be also used for uninterpreted actions or comments

2.1.1.14. **DataType**: only integer, Boolean, enumeration of literals and class identifiers are considered. To define data type corresponding to arrays in programming languages, *Multiplicity* attribute can be used with one *MultiplicityRange*. Strings can be used for comments or unintepreted actions.

2.1.1.15. **Dependency**: not supported.

2.1.1.16. **ElementOwnership**: not considered (being by default the whole model)

2.1.1.17. **ElementResidence**: is relevant only for the component model, thus, not considered here.

2.1.1.18. **Enumeration & EnumerationLiteral**: no restriction.

2.1.1.19. **Feature**: attribute *ownerScope* is considered with one value *instance*.

2.1.1.20. **Flow**: not supported.

2.1.1.21. **GeneralisableElement**: no restriction.

2.1.1.22. **Generalisation**: no *powertype* is considered.

2.1.1.23. **Interface**: is relevant only for the component model.

2.1.1.24. **Method**: the only legal actions in a method body are those specified in Section 3.

2.1.1.25. **ModelElement**: no templates and dependency relations are supported, so, no attributes related to templates and dependency.

2.1.1.26. **Namespase**: no restriction.

2.1.1.27. **Node**: not relevant here.

2.1.1.28. **Operation**: attribute *isAbstract* has only value *false* (abstract operations are not supported). We introduce additional attribute *isTriggered* of Boolean type, to distinguish operations those rise call events from those having associated method.

2.1.1.29. **Parameter**: attribute *kind* has value *inout*.

2.1.1.30. **Permission** (inherits from Dependency relation): not considered.

2.1.1.31. **Primitive**: UnlimitedInteger is not considered (can not be used for the verification).

2.1.1.32. **ProgrammingLanguageDataType**: not considered other than predefined (possibly with a range), enumeration, and class definitions.

2.1.1.33. **Relationship**: only association (three kinds: composition, aggregation and the rest called for convenience *neighbour*) and generalisation relation.

2.1.1.34. **Stereotype**: only those, described explicitly in the time- and component extensions of the Omega kernel language are allowed (no other, in particular, user defined).

2.1.1.35. **StructuralFeature**: attributes and association ends are considered here (ports are described in the specification of component model). The restrictions on the attribute values for association ends are mentioned above. For attributes, *changeability* is considered equal to *changeable*, *multiplicity* is considered equivalent to *range* to form arrays, *targetScope* is *instance*, *persistence* is *transitory*.

2.1.1.36. **TemplateArgument** and **TemplateParameter**: are not considered since templates are not supported.

2.1.1.37. **Usage** (inherits from Dependency relation): not supported.

2.1.1.38. **Expression**: only expressions mentioned in Section 3 are allowed (as listed in `<expression>`).

2.1.1.39. **Multiplicity**: can consist only of one **MultiplicityRange**. Three kinds of multiplicity are considered – fixed number (range [n,n] with n>0), unbounded multiplicity * (range [0,*]), or a range [n,m] with m>n≥0.

2.1.1.40. **Package**: attribute *importedElement* is not supported.

2.1.2 Behavioural Elements

2.1.2.1. **Action** and **ActionSequence**: are defined in Section 4.

2.1.2.2. **Argument**: described in Section **Erreur ! Source du renvoi introuvable.** as `<opt_simple_expression_list>`.

2.1.2.3. **AttributeLink**: no restrictions

2.1.2.4. **Exception**: syntactically not distinguished from signal event (should have higher priorities than other signals).

2.1.2.5. **Instance**: attribute *persistent* has only value *transitory*.

2.1.2.6. **Link** and **LinkEnd**: attribute *qualifierValue* is not supported other than index value (since *qualifier* is supported only for *multiplicity*, see item 5).

2.1.2.7. **Object**: no restriction.

2.1.2.8. **Reception**: attribute *isAbstract* is always *false* (abstract signals are not considered).

2.1.2.9. **Signal**: no restrictions.

2.1.2.10. **Stimulus**: no restrictions.

2.1.2.11. **SubsystemInstance**: is relevant only for the component model, not considered here.

2.1.2.12. **Actor**: no restrictions.

2.1.2.13. **CallEvent**: the corresponding triggered operation (rising the event) has attribute *concurrency* with value *guarded* or *sequential*.

2.1.2.14. **ChangeEvent**: not supported.

2.1.2.15. **CompositeState**: no restriction.

2.1.2.16. **Event**: no restriction.

2.1.2.17. **FinalState**: no restriction.

2.1.2.18. **Guard**: is a Boolean expression without side effects (possible syntax is described in Section 3).

2.1.2.19. **Pseudostate**: attribute *kind* has not value *choice* (a static variant of *choice* is a kind of *junction* pseudostate)

2.1.2.20. **SignalEvent**: no restriction.

2.1.2.21. **SimpleState**: no restriction.

2.1.2.22. **State**: attributes *doActivity* and *internalTransition* are not supported (only explicit transitions are allowed).

2.1.2.23. **StateMachine**: only one per class. A statemachine can be only attached to a class, port/interface or component.

2.1.2.24. **StateVertex**: no restrictions.

2.1.2.25. **StubState**: not considered.

2.1.2.26. **SubmachineState**: not considered (no special semantics, used only for convenience).

2.1.2.27. **SynchState**: not considered.

2.1.2.28. **TimeEvent**: no restrictions, defined in more details in Section 6.

2.1.2.29. **Transition**: no restriction.

2.2 Omega Wellformedness Rules

2.2.1 Classes and Associations

2.2.1.1. There is the root class for every (component) model – the maximal class under composition and aggregation relation, which is active[2].

2.2.1.2. If an association (composition, aggregation or neighbour) relation has more than two association ends, then one of them is called root-end ("starting point" of the association), and all others are called end-points and must be navigable and visible. The root-end can be navigable and visible only in all the corresponding end-points or in none of them.

2.2.1.3. The composition relation must define a directed acyclic graph.

2.2.1.4. For all composition relation, all its end-points *ac_id.cj* have the following properties:

[2] UML 1.4 defines a root only for the generalisation relation (attribute *isRoot*), but we also need such notion for the composition and aggregation for the semantic reason (and the definition of the initial configuration).

- The value of attribute *multiplicity* is a fixed number (n>0) or unbounded (denoted as *)
- Attribute *isNavigable* has value *true*
- Attribute *changeability* has value *frozen* if its multiplicity is fixed (n>0) or *addOnly* if its multiplicity is unbounded (*).
  Attribute *multiplicity* of the root-end has value 1 and *changeability* is *frozen*.

2.2.1.5. For all aggregation relation, all its end-points have the following properties:

- The value of attribute *multiplicity* is a fixed number (n>0), unbounded (*) or range [m,n], where m, n>0.
- Attribute *isNavigable* has value *true*
  Attribute *multiplicity* of the root-end has value 1 and *changeability* is *frozen*.

2.2.1.6. For other associations (neighbour):

- All end-points as well as the root-end have the value of attribute *multiplicity* as a fixed number (n>0), unbounded (*) or range [m,n], where m, n>0.

2.2.1.7. No sharing of weak components (end-points of aggregation relations) between several weak composites (root-ends of aggregation relations) in run-time.

2.2.2 Operations, Events and Attributes

2.2.2.1. Only the following kinds of stimuli are considered: signal event emissions, operation calls, timeouts, object creation and destruction (special kinds of operation calls). Signal events have public visibility.

2.2.2.2. There are two special kinds of primitive operations: constructor and destructor, which can be defined via a tagged value *isSpecial* of type enumeration (*constructor*, *destructor*, *none*), because the presence of these operations is restricted by the inter-object relations (see item 2.2.2.8).

2.2.2.3. No naming conflicts of operations, attributes, classes and associations names – e.g., in multiple inheritance.

2.2.2.4. There is no invocations of triggered operations in the bodies of primitive operations.

2.2.2.5. A dependency graph of operation calls is tree-like (without recursions).

2.2.2.6. Triggered operations are guarded or sequential.

2.2.2.7. Primitive operations are sequential or free of side effects (attribute *isQuery* is *true*).

2.2.2.8. For all classes c, operations for object creation and destruction (primitive operations with the corresponding tagged values) of c can be invoked from a class c' iff there is aggregation or composition association directed from c' (association root) to c.

2.2.2.9. Each object has (additional to standard) link *my_ac* to an active object, which is derived from the structure of inter-object relations (via composition). Thus, each active object with all its associated passive objects form a so called *activity group*. For the safety reasons, we restrict the model behaviour so, that an object may call an operation only from an object of the same activity group.

The syntax of the Omega actions is defined formally in Section 4. Here are some restrictions on their usage within method bodies.

2.2.2.10. No variable or constant declaration within operation bodies (all declarations should be specified at the level of class definition, i.e. as attributes with the desired visibility).

2.2.2.11. Restricted set of primitive actions and constructs (with syntax described in Section 3):

- Object creation
- Object destruction
- Assignment of an expression to an object attribute
- Assignments of attributes/navigation expressions to attributes/navigation expressions
- Operation call
- Setting return value
- Signal emission

We allow sequential composition, "if-then-else" conditional composition and FOR- and WHILE-loops.

2.2.2.12. For all navigation expression in the form a0.a1…..an (n ≥ 0):

- all references *a0,…, an-1* are association role names (can be default names *self*)

2.2.2.13. For all n ≥ 0 and assignments of a value to a navigation expression (e.g. a0.a1…..an := value) we require that an is a basic or navigation attribute, meaning that its type is a predefined type or it is an association end.

2.2.3 Statecharts

2.2.3.1. Every statechart must have a distinguished top state which is of mode OR.

2.2.3.2. The priorities of transitions rise from outmost to innermost source state, meaning that a transition originating from a substate has higher priority than a conflicting transition originating from any of its containing states.

2.2.3.3. If a class inherits from several classes, then only one of the generalised classes has statechart or generalised classes have equal statecharts. If a new statechart is specified in a specialised class, then it completely overwrites any statechart from its generalised class, i.e. the delegation of signal and call events (to the definition of their reception in another class) is not supported.

3 Action language

The syntax elements in this section are taken and adapted from milestone [M221].

3.1 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.

3.2 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.

3.3 OMAL Syntax

This section contains the lexical and syntactic rules defining OMAL.

3.3.1 Lexical tokens

3.3.1.1 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).

3.3.1.2 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.

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

3.3.1.3 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.

3.3.1.4 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'

3.3.1.5 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'
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'

3.3.1.6 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'

[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.

- 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 '"'.

3.3.1.7 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' '='
'<>' '->' '::' '+' '-' '*' '/' '%'

3.3.2 Grammar

3.3.2.1 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 ε.
  - Round parentheses ( ) are used to group token/nonterminal sequences
  - Optional parts are written between [ ]
  - Repetitive parts formed of 0 or plus occurrences of α are written as (α)*
  - Alternative between α and β are written as α | β

3.3.2.2 Production rules[5]

3.3.2.2.1 Actions

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

elementary_action ::= ε
| *assignment_action*
| *call_action*
| *send_action*
| *return_action*
| *reply_action*
| *informal_action*

assignment_action ::= *navigation_expression* ':=' *expression*

call_action ::= *call_expression*

[5] The grammar in this form is ambiguous and is neither LL(∞) nor LR(∞). 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

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

3.3.2.2.2 Expressions

expression ::= *call_expression*
| *create_expression*
| *simple_expression*

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

3.4 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.

3.5 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
```

3.6 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.

3.6.1 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.

3.6.2 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.*

4 Time extensions: Predefined data and object types

The syntax elements in this section are taken from deliverable [D113].

The following data types are defined in a UML package *OMEGAPredefined*. This package has to be included at the root level in every UML model that uses these types.

The definition of this package is available in Rational Rose and Rhapsody format. It contains several additional data types, which are ignored by the tools.

1. Timer

The predefined object type **Timer** gives simple means to observe and use time passage in behavior description. The following operations are currently defined on timers:

- **set (<duration>)** – where <duration> is an **integer** value. It sets the expiry time of the timer within <duration> time units.
- **reset() –** It puts the timer in an inactive state

When a timer is active and the expiry time is reached, a timeout event is generated. This event may be specified as trigger to a statemachine transition, using the syntax:

timeout(<timer variable>)

It is not specified whether the timeout event is sent as a signal or observed by other means, nor whether it is broadcast or sent to a uniquely defined object.

Note that, in this way, any object that has a navigation path to the timer variable may call all operation and may observe the timeout.

5 Time annotations syntax

The syntax elements in this section are taken from deliverable [D113].

2. Event types

An event type is a special kind of object type defined by the following components:

- a set of **attributes** (data or object references, as for any object type)
- an **update statement**, which is automatically executed by the timed analysis runtime at specific moments defined below. The purpose of this statement is to update the attributes of the event object with some observed data. The update statement is composed of the following:
    - an **event matching statement** which will identify the moments when the update statement is executed, by linking them to some other actions performed in the system. The event matching statement gives:
        - the event kind, which is one of the following: Invoke, InvokeReturn, Send, …
        - additional meta-data relative to the event (operation name, etc)
        - a rule for storing data related to the event (which depends on the event kind, and may be for example the sending object, receiving object, etc.) into event attributes.
    - a **condition** which establishes when the event is considered to be triggered (updated)
    - a **statement** which is executed is the condition is true, and which is used to store additional data from the model, for later use.

5.1.1 Event inheritance

For the convenience of writing event types, they can be inherited. The event inheritance has the semantics of inheriting attributes. The update statement definition is completely lost during inheritance.

5.1.2 Syntax

A Class with the stereotype <<TimedEvent>> defines an event type in the UML model.
The attributes of the event are given as the attributes of that class.
There may not be any operation defined in the event class.
There may not be any structural relationships (inheritance, associations) defined for the event class.
The update statement is attached as a note to the classifier. Its syntax is :

<EventUpdateStatement> ::=
match <EventMatchingStatement>
[**when** <SimpleExpression *(from action language)*>]
[**do <**Action *(from action language)*>]
< EventMatchingStatement > ::=
<InvokeStatement> | <InvokeReturnStatement> |<SendStatement>
| <ReceiveStatement> | <AcceptStatement> | <ReceivereturnStatement> | <AcceptreturnStatement> |
| <ReceivesignalStatement> | <AcceptsignalStatement>
| <StartStatement> | <EndStatement> | <StartendStatement> | <StarttransitionStatement>
| <EndtransitionStatement> | <StartendtransitionStatement>
| <StartStateStatement> | <EndStateStatement> | <StartendStateStatement> | <ChangeEventStatement>

.
<SimpleExpression *(from action language)*> and **<**Action *(from action language)*> are interpreted in the name context of the event class (i.e. may navigate via the event attributes).

5.1.3 Semantics

Variables having as type an event type may be declared either globally for the UML model or locally in a class. The syntax for declaring such variables is described in Section 4. If global, such a variable will point to a unique event object. If local to a class, such a variable points to a different event object for each instance of the class. Wherever such variables are allowed to be used (e.g. in the condition attached to a time constraint), they are used like references to normal objects (i.e. may be the starting point of navigations).
Declaring an event variable to be global or local may change the way it is updated:

- global event objects are updated whenever a matching behavior occurs anywhere in the system.
- local event objects are updated only when a matching action is performed or may be directly observed by the object to which they are attached. The meaning of "performed" or "directly observed" is defined for each event kind in the following.

3. Event matching statements

The following event matching statements may be used for specifying the update of an event object.

An event object may be specified in a local scope (a class) only when it matches an action that can be initiated or directly observed by the instances of the class. (Example: an **invoke** event may be specified in the class whose objects are invoking the operation, but not in the invoked operation or its enclosing class.) Thus, for each event kind, we specify which are the objects that initiate or directly observe the corresponding action.

Note: for the definition of the syntax, some frequent non-terminals are defined here:

<ModelElement> ::= <Identifier>
<QualifiedModelElement> ::= [<QualifiedModelElement> '::'] <ModelElement>
<NumericComparison> ::= **'<' | '<=' | '>=' | '>' | '=' | '<>'**
<NumericLiteral> ::= …

A. Interaction events

In this category we place the events relative to operation invocation or signal exchange. There are two kinds of interaction events:

1. Events *initiated* by the execution of an action, and which are *observable* by the same object that executed the action. This category comprises: **invoke**, **invokereturn**, **send**.
2. Events which are a *consequence* of a causal chain, which are *not observable* by the object that initiated the causal chain but which *are observable* by another object in the system. This category comprises: **receive, accept, receivereturn, acceptreturn, receivesignal, acceptsignal**.

The above distinction is necessary because, as mentioned before, an event specification placed in a local scope (*Class*) refers only to the events, of corresponding type/attributes, which are observed by the local *object* / *operation activation*.

▪ Invoke

An **invoke** event specification matches the instants when a call request for a specific operation is emitted.

5.1.3.1 Syntax

<InvokeStatement> ::=

invoke <Operation> (<NavigationExpressionList>)

[**by** <NavigationExpression *(from action language)*>]
[**on** <NavigationExpression *(from action language)*>]

where:

- <Operation> ::= <QualifiedModelElement> - designates an *Operation*.
- <NavigationExpressionList> ::=
  | (<NavigationExpression *(from action language)*> | **void**)
  (**','** (<NavigationExpression *(from action language)*> | **void**))*

5.1.3.2 Locality

In a local scope, an <InvokeStatement> matches invocations performed by the local object on other objects.

5.1.3.3 Static semantics

Type checks:

- The number and types of the expressions appearing in the parameter list after the <Operation> must match the number and types of the formal parameters of the designated *Operation*.
- **void** matches any type and designates no store location.

▪ InvokeReturn

An **invokereturn** event specification matches the instants when the return to an invocation of an operation is sent by the callee.

5.1.3.4 Syntax and semantics

<InvokeReturnStatement> ::=

invokereturn <Operation> (<NavigationExpressionList>)

[**by** <NavigationExpression *(from action language)*>]

[**to** <NavigationExpression *(from action language)*>]

where:

- <Operation> ::= <QualifiedModelElement> - designates an *Operation*.
- <NavigationExpressionList> is defined as in Section 5.1.3.1

Semantics:

- An **invokereturn** event specification in a class *C* matches the sending of a return to an invocation for one of the operations of the class. An **invokereturn** event specification in an operation *op* matches the handling of an invocation for *op*.

5.1.3.5 Locality

In a local scope, an <InvokeReturnStatement> matches the return from a local operation call initiated by an other object.

5.1.3.6 Static semantics

Type checks:

- The number and types of the expressions appearing in the parameter list after the <Operation> must match the number and types of the formal parameters of the designated *Operation*.
- **void** matches any type and designates no store location.

▪ Send

A **send** event specification matches the instants when a specific signal is emitted.

5.1.3.7 Syntax and semantics

<SendStatement> ::=

send <Signal> (<NavigationExpressionList>)

[**by** <NavigationExpression *(from action language)*>]

[**to** <NavigationExpression *(from action language)*>]

where:

- <Signal> ::= <QualifiedModelElement> - designates a *Signal*.
- <NavigationExpressionList> is defined as in Section 5.1.3.1

5.1.3.8 Locality

In a local scope, a <SendStatement> matches a signal sent out by the local object to some other objects.

5.1.3.9 Static semantics

Type checks:

- The number and types of the expressions appearing in the parameter list after the <Signal> must match the number and types of the formal parameters of the designated *Signal*.
- **void** matches any type and designates no store location.

▪ ReceiveReturn, AcceptReturn

A **receivereturn** event specification matches the instants when a return to a call request for a specific operation is received by the caller.
An **acceptreturn** event specification matches the instants when a return to a call request for a specific operation is handled by the caller.

5.1.3.10 Syntax and semantics

<ReceiveReturnStatement> ::=

receivereturn <Operation> (<NavigationExpressionList>)

[**by** <NavigationExpression *(from action language)*>]

[**to** <NavigationExpression *(from action language)*>]

<AcceptReturnStatement> ::=

acceptreturn <Operation> (<NavigationExpressionList>)

[**by** <NavigationExpression *(from action language)*>]

[**to** <NavigationExpression *(from action language)*>]

where:

- <Operation> ::= <QualifiedModelElement> - designates an *Operation*.
- <NavigationExpressionList> is defined as in Section 5.1.3.1

5.1.3.11 Locality

In a local scope, a <ReceiveReturnStatement> or <AcceptReturnStatement> matches the return from an operation call initiated by the local object on other objects.

5.1.3.12 Static semantics

Type checks:

- The number and types of the expressions appearing in the parameter list after the <Operation> must match the number and types of the formal parameters of the designated *Operation*.
- **void** matches any type and designates no store location.

▪ Receive, Accept

A **receive** event specification matches the instants when an invocation of an operation is received by the callee. An **accept** event specification matches the instants when an invocation of an operation is handled by the callee.

5.1.3.13 Syntax and semantics

<ReceiveStatement> ::=

receive<Operation> (<NavigationExpressionList>)

[**by** <NavigationExpression *(from action language)*>]

[**from** <NavigationExpression *(from action language)*>]

<AcceptStatement> ::=

accept <Operation> (<NavigationExpressionList>)

[**by** <NavigationExpression *(from action language)*>]

[**from** <NavigationExpression *(from action language)*>]

where:

- <Operation> ::= <QualifiedModelElement> - designates an *Operation*.
- <NavigationExpressionList> is defined as in Section 5.1.3.1

5.1.3.14 Locality

In a local scope, a <ReceiveStatement> or <AcceptStatement> matches the reception of an operation call to a local operation initiated by other object.

5.1.3.15 Static semantics

Type checks:

- The number and types of the expressions appearing in the parameter list after the <Operation> must match the number and types of the formal parameters of the designated *Operation*.
- **void** matches any type and designates no store location.

- ReceiveSignal, AcceptSignal

A **receivesignal** event specification matches the instants when a signal is received by its destination.
An **acceptsignal** event specification matches the instants a signal is handled by its destination.

5.1.3.16 Syntax and semantics

<ReceiveSignalStatement> ::=

receivesignal<Signal> (<NavigationExpressionList>)

[**by** <NavigationExpression *(from action language)*>]

[**from** <NavigationExpression *(from action language)*>]

<AcceptSignalStatement> ::=

acceptsignal <Signal> (<NavigationExpressionList>)

[**by** <NavigationExpression *(from action language)*>]

[**from** <NavigationExpression *(from action language)*>]

where:

- <Signal> ::= <QualifiedModelElement> - designates an *Signal*.
- <NavigationExpressionList> is defined as in Section 5.1.3.1

5.1.3.17 Locality

In a local scope, a <ReceiveSignalStatement> or <AcceptSignalStatement> matches the reception of a signal by the local context sent out by other object.

5.1.3.18 Static semantics

Type checks:

- The number and types of the expressions appearing in the parameter list after the <Signal> must match the number and types of the formal parameters of the designated *Signal*.
- **void** matches any type and designates no store location.

B. Action events

The action events must be specified either globally or locally in a *Class* or *Operation.* If local, they refer to an action executed by the *Class*' state machine or by the concerned operation.

5.1.4 Start, End, StartEnd

A **start** event specification matches the instants when a specific action starts executing.
A **end** event specification matches the instants when a specific action ends executing.
A **startend** event specification matches the instants when a specific atomic action starts executing.

Syntax and semantics<StartStatement> ::=

start <Class> '**@'** <label > [**by** <NavigationExpression *(from action language)*>]

<EndStatement> ::=

end <Class> '**@'** <label > [**by** <NavigationExpression *(from action language)*>]

<StartStatement> ::=
startend <Class> '**@'** <label > [**by** <NavigationExpression *(from action language)*>]

where:

- <Class> ::= <QualifiedModelElement> - designates the class containing the action
- <label> ::= <Identifier> - designates the action label
- <NavigationExpressionList> is defined as in Section 5.1.3.1

5.1.4.1 Locality

In a local scope, a <StartStatement>, <EndStatement> or <StartEndStatement> match the reach of an action (start or end) by the local context.

5.1.4.2 Static semantics

Type checks:

- The action should be contained in the object specified in the **by** clause

C. Transition events

5.1.5 StartTransition, EndTransition, StartEndTransition

A **starttransition** event specification matches the instants when a specific transition starts executing.
A **endtransition** event specification matches the instants when a specific transition ends executing.
A **startendtransition** event specification matches the instants when a specific atomic transition starts executing.

5.1.5.1 Syntax and semantics

<StartTransition Statement> ::=

starttransition <Class> '@' **<TransitionName> [** by **<NavigationExpression *(from action language)*>]**

<EndTransition Statement> ::=

endtransition <Class> '@' **< TransitionName> [** by **<NavigationExpression *(from action language)*>]**

<StartEndTransition Statement> ::=
startendtransition <Class> **'@'** < TransitionName> [**by** <NavigationExpression *(from action language)*>]

where:

- <TransitionName> ::= <Identifier> - designates a transition name.
- <NavigationExpressionList> is defined as in Section 5.1.3.1

5.1.5.2 Locality

In a local scope, a <StartTransition>, <EndTransition> or <StartEndTransition> match the start /end of a transition execution by the local context.

5.1.5.3 Static semantics

Type checks:

- The transition should be contained in the object state machine (or in some of its operation state machines) specified in the **by** clause

5.1.6 Enter, Exit

An **enter** event specification matches the instants when a specific state is entered.
An **exit** event specification matches the instants when a specific state is exit.

5.1.6.1 Syntax and semantics

<StartStateStatement> ::=

enter <Class> '@' **< StateName> [** by **<NavigationExpression *(from action language)*>]**

<EndStateStatement> ::=

exit <Class> '@' **< StateName> [** by **<NavigationExpression *(from action language)*>]**

<StartEndStateStatement> ::=
startendstate <Class> **'@'** < StateName> [**by** <NavigationExpression *(from action language)*>]

where:

- <StateName> ::= <Identifier> - designates a state name.
- <NavigationExpressionList> is defined as in Section 5.1.3.1

5.1.6.2 Locality

In a local scope, a <StartState>, <EndState> or <StartEndState> match the reach of a state entry/exit by the local context.

5.1.6.3 Static semantics

Type checks:

- The state should be contained in the object state machine (or in some of its operation state machines) specified in the **by** clause

D. Pre events

In this cathegory we place the « pre » events declarations. While all event definitions until now allow to identify the last occurrence of the event, the pre events allow us to identify events that happened in the past.

5.1.7 Syntax and semantics

<PreEventUpdateStatement> ::= (**pre)*** <EventUpdateStatement>

The number of pre keywords in the pre event declaration indicate how many events up to the last are we interested in: a single pre corresponds to the event occurrence previous to the one currently identified, a pre pre event declaration corresponds to a event occurrence second previous to the current one. Pre event declarations can only be identified when it's reference event occurs, this means that the actions that may exist in the event update statement are executed not when the event was detected as pre (i.e. not now), but at pre's reference occurrence time, i.e. when the event originally occurred (two event occurrences before).

4. Constraints

Timed constraints are considered as predicates that have to hold at certain points in time (usally, at the time of occurrence of a certain event).
Note: The scope of the event specifications used in a constraint is the scope of the constraint itself. Therefore, a constraint may be attached to a local element only if it refers only to locally observable events. Global constraints are attached to the <<TimeConstraint>> stereotyped class present in the model (and added to

5.1.8 Duration between events

A duration between two events, ev1 and ev2, measures the time passed between the last occurrence of the first events and the very next occurrence of ev2. If no ev2 occurred after the last occurrence of ev1, then we consider the previous occurrence of ev1, and so on.
If no ev1 occurred, then the duration between ev1 and ev2 is undefined (no matter if ev2 occurred or not).
The following components are important for a duration:

- the two events
- a filter condition (other than the filter conditions that may be present in the event type declarations) that applies only to this duration

Syntax and semantics

<BasicTimedConstraint> ::=
<BasicTimeDuration><NumericComparison> <NumericLiteral>

<BasicTimeDuration> ::=
<BasicDuration> |<PipelineDuration>

< BasicDuration > ::=
duration '('<Event attribute> **','** <Event attribute> **')' when** <DurationCondition>

<PipelineDuration>::= TBD

where:

- <Event attribute> ::= <Identifier> - designates an event attribute defined in the same scope as the constraint.
- <DurationCondition> ::= <SimpleExpression *(from action language)*> - interpreted in the scope enclosing the constraint

The two <Event attribute>s cannot be accessed by navigation.

- The <DurationCondition> serves at filtering the events on which the duration can be established

Semantics:

- <BasicTimedConstraint > constrains the duration between an event matching the second event specification, and the most recent event matching the first event specification.
- This constraint has to hold only upon the occurrence of the second event, if an event matching the first specification precedes it.

6 Property specification using UML observers

UML observers are the automata-based property specification mechanism used by the Verimag tool. From the syntactic point of view, observers re-use much of the existing concepts of UML and of the aforementioned time extensions, and introduce only some minimal extensions in the form of UML stereotypes.

Thus, an observer is specified in the UML model by a class stereotyped with <<Observer>>. The automaton defining the observer property is the UML state machine of that class.

The state machine may react either to simple (state-based) conditions in the model, or to events. The event types to which an observer may react are the same as the ones defined in the section 5. Thus, the transition triggers of an observers may include event matching statements, defined in section 5 by the production associated to the <EventMatchingStatement> non-terminal.

In order to obtain a property specification, the user has to classify some states of the observer as *error* or *success*. Syntactically, this is done by stereotyping the states with <<Error>> or with <<Success>>.

The model checking tools will search for executions of the system leading the observer to error or success states. This mechanism allows specifying arbitrarily complex *state- or event-based (and possibly timed) safety properties*.

An example of observer is shown in Figure 1 for the following property taken from the NLR case study: *if the DatabusController becomes not operational for over 10ms, then eventually within these 10m the MessageReceiver will be in ControllerError state.*

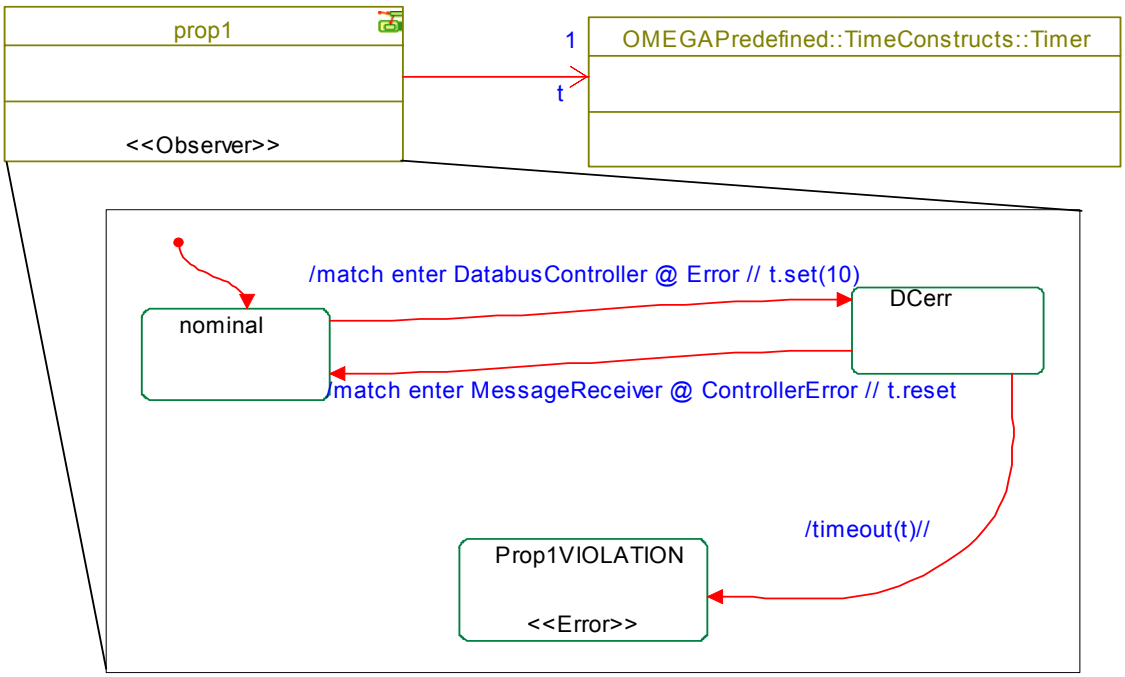

Figure 1. Observer for timed property of NLR case study

Further details on observers may be found in [OGO03].

7 Component based design syntax

The CASE tools Rhapsody and Rational Rose do not support components, yet. As a workaround, a component based design can be done within the Kernel Model Language by means of substitution: Class for Component, rolenames of realization or dependency relations for Ports. A user can draw classes and interfaces in Rhapsody and use a naming convention to make clear that this is a component diagram and not a class diagram.

Figure 2 displays a component diagram and how it can be represented as a class diagram. The association role names can be used to capture component port names and their role (required or provided).

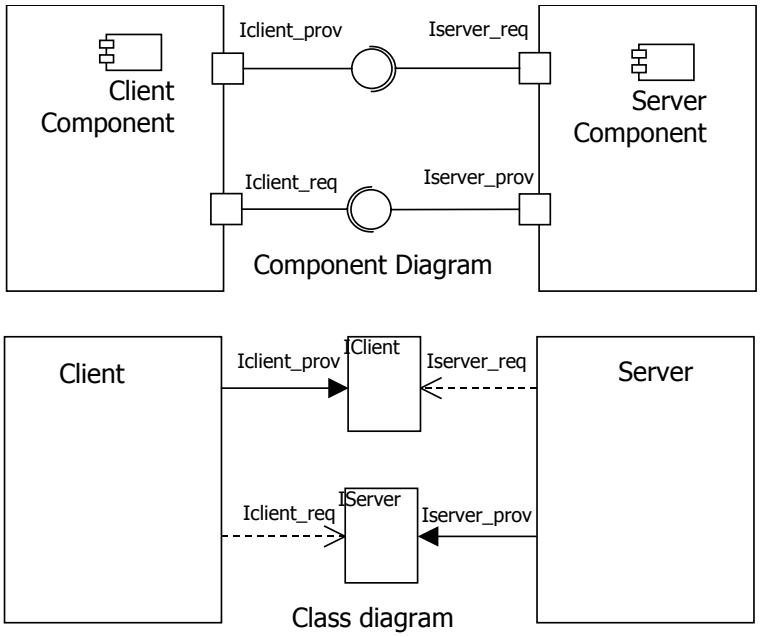

Figure 2 Component

Figure 3 displays how multiple interfaces can be associated with one port on a component: by means of the "myport" role.

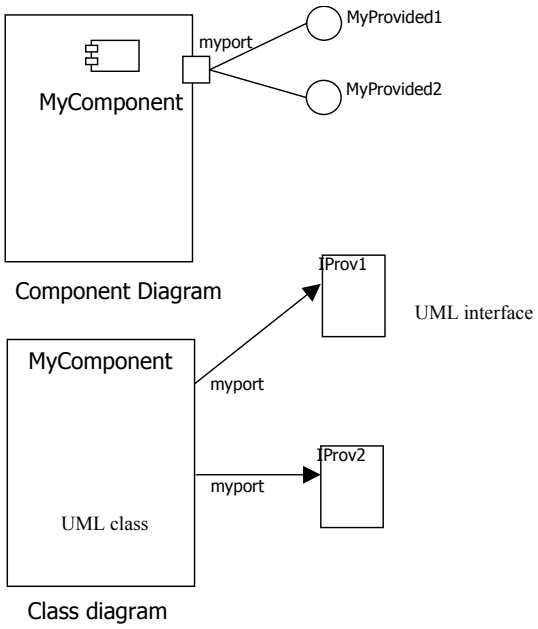

Figure 3

Note that there can be no statemachines associated with components, yet, because the OMEGA action language does not support components yet. For example a return statement would have to be extended with a parameter indicating which object inside a component it should go to. However, this way a user can create a hierarchical set of diagrams and this set can then be used by the OMEGA tools. Because of the hierarchical structure it will then be possible to verify properties on a high level in the design, without the need for a complete design.

A big disadvantage of the "substitution" approach is that this loses information like "what classes are associated with a port" if classes can not have classes inside. We rely on the user to organize a correct hierarchy. Also there are problems like when you want to add 2 Ports with identical interfaces to a component. These kind of problems can be solved, but the result will look hackish and ugly. A better approach is to use SUML extended with suitable elements for component based designs. This has no loss of information and links the component diagrams and their underlying class diagrams together. An example is in the OMEGA component report. The extended SUML DTD for components is also in the component report. Of course the CASE tools do not support SUML output but a SUML encoding is so simple that it can be produced with a text editor. If and when the CASE tools do support components then we can built a SUML2XMI tool just like the XMI2SUML tool we already built for the UML2PVS project.

8 Syntax of OCL as supported in Simple UML

This section describes the syntax and semantics of OCL as supported by our Simple UML library. The Simple UML library is part of our tool, and is responsible for parsing and type checking OCL files.

This document is part of the Simple UML library's manual.

5. Vocabulary

In the current draft of the OCL 2.0 proposal it is not clear what constitutes a keyword, which identifiers are reserved, and which identifiers may be defined by the user. In this section we explain the choices we have made.

8.1.1 Keywords

Keywords in OCL are reserved words. That means that a keyword cannot occur anywhere in an OCL expression as the name of a classifier, a property, or a package. The keywords are:

and
body
context
def
derive
else
endif
endpackage
if
implies
in
init
inv
let
not
or
package
post
pre
then
xor

Additionally we treat the identifiers `false' and `true' as keywords.

If the Simple UML library is working in _ASO/OCL_-mode the following strings become keywords: `interface' and `local'.

8.1.2 Reserved Identifiers

In addition to the keywords other identifiers are _reserved_ by the Simple UML library. Depending on their context these identifiers have a special meaning, which the user cannot override by redefining them in his model.

We advise the user to treat the reserved identifiers as keywords. If he intends to use them for other purposes in his model, he has to make sure, that his use of the keyword is consistent with the meaning it has in an OCL constraint.

In the following paragraphs we describe those special identifiers:

In a *post* condition the identifier `result' is a variable of the same type as the operation which is to be constrained. The variable holds the return-value of the operation. We advise a user to use the following ideom in his code, if he needs to use `result' as an identifier.

The identifier `result' can safely be used as a local variable. If it is used as such, then the action associated to the operation has to return the value of `result' by using the action `return result'.

The identifier should not be used as a parameter name, because `result@pre' is confusing.

8.1.3 Basic Expressions

OCL is a strongly typed expression language.

Literals

Literals are the most simple kind of expressions in OCL. In OCL you have `Boolean', `Integer', `Real', `String', and `Collection' literals (We do not yet support tuple literals).

Boolean Literals

The `Boolean' literals are `true' and `false'. Alternatively, you can write `Boolean::true' and `Boolean::false', because UML 2.0 defines `Boolean' as an enumeration type with two values.

```
true
false
Boolean::true
Boolean::false
```

Integer Literals
Any sequence of digits is considered to be an `Integer' literal in OCL.

The only representation of integers in OCL 2.0 is decimal. This library also supports binary, octal, and hexadecimal integer literals. As in the C programming language, an octal integer literal is prefixed with a single `0' (null), and an hexadecimal integer literal is prefixed with a `0x' or `0X'. Binary literals are prefixed by a `0b' or `0B'.

```
0    -- Integer literal
11   -- Integer literal in decimal.
011  -- Integer literal in octal (9 decimal)
0x11 -- Integer literal in hexadecimal (17 decimal)
0b11 -- Integer literal in binary (3 decimal)
```

Note: There are no negative integer literals. These are represented by an unary expression.

Real Literals
Real literals are sequences of digits using the same notation as in C. Equivalent ways of expressing 1% are:

```
0.01
1e-2
1E-2
```

String Literals
String literals are enclosed in single quotation marks `'. Characters can be escaped using the C-style backslash `\'. String literals must not contain a new line character. This character may be represented by `\n'. The operation `+' of class `String' allows string concatenation.

```
''      -- The empty string
'\''    -- A single quote as string
'String'
'\n'    -- A newline character
```

Collection Literals
The most complex kind of literals are collection literals. A collection literal is formed by the type of the collection, followed by an opening brace (`{'), followed by the contents of the collection, which is a comma-seperated list of expressions, and ends in a closing brace (`}'). The collection types are `Bag', `Set', and `Sequence', with its super-type `Collection'.

```
Set{ 1, 2, 3, 4, 5 }
```

You may use two consecutive dots to define ranges of integers in an collection literal.

```
Set{ 1..5 }       -- same as Set{ 1, 2, 3, 4, 5 }
Set{ 1..3, 5..6 } -- same as Set{ 1, 2, 3, 5, 6 }
```

The lower and upper bound of an range expression may be an arbitrary expression. If the first value of the range expression if larger than the second one, the expression is considered empty.

```
Set{ a..b } -- Empty, if a > b
```

The rules for deriving a type for collection literals are quite complex. The kind of the collection is defined by the literal, while the type of the elements of a collection have to be deduced by the type checker:

```
Set{ 1.0, 2 } -- Set(Real)
```

In general it is not possible to determine a unique type for the elements of a collection literal. The current implementation of the type checker will chose an arbitrary type. This may result in false type errors. To avoid this problem, you have to explicitely cast the elements of the collection literal:

```
Set{ a->oclAsType(A), b->oclAsType(A) } -- Set(A)
```

This work-around does not work, if the elements of a collection are collections. In any case you have to re-type the elements which are derived from `OclAny'.

If the collection literal is empty (`Set{}'), the type of the elements is assumed to be `OclAny'. To create an empty collection of another type, you need to create a non-empty collection first, and remove its elements:

```
Set{ 1 }->reject(true) -- Empty Set(Integer)
```

While syntactically possible, we don't suggest using the collection type `Collection' for writing collection literals, except for the literal `Collection{ }'.

Property Calls
The second important ingredient of OCL is the *property call*. The property call represents the access to all variables, navigations, and operation calls.

Local Variables
The most simple form of a property call is an identifier, which is used to access a local variable or a property of the enclosing classifier is used:

```
name
```

If no local variable of the name `variable' is defined, then the value of an attribute (or association) of the classifier, in which this expression is defined, is used, instead.
If no attribute or association of this name is defined, then an operation without formal parameters will be called.

```
name()
```

In this example, the call of an operation is forced by providing an empty argument list.

```
self.name
```

This example avoids the use of an local variable, but will use a property defined in the classifier.

As in other object-oriented programming languages you can chain these property calls:
```
name1.name2.name3
```

Note, that OCL, and UML, only use the concept of a property, a variable, and a literal. The number two can be written as:
```
1.+(1)
1.+(2).-(1)
1.+(2).+(1.-)
1.+(2).-(1.-.-)
```
This way of writing expressions is a bit obfuscating.

Operation Calls
Some operations are treated specially by the Simple UML library, either because it is currently not possible to define their signature in Simple UML or because those operations have certain semantic commitments.

The infix notation of equality (`=') is not the same as the corresponding operation of the standard library or a user's model. We assume that infix equality is _strong equality_, and it will always return a valid Boolean value.

If the user wants to define his own equality operation, and he wants to use it, he has to write `a.=(b)' instead of `a = b' (mind the dot and the parnethesis).

The operations `oclIsTypeOf', `oclIsKindOf', `oclInState', and `oclAsType' are not defined in the standard library. Their argument or return types are special and you cannot define their signature in our Simple UML format. All operations require a valid expression as an argument, but `oclIsTypeOf', `oclIsKindOf', and `oclAsType' require a type as their argument and `oclInState' requires the state name of a state machine. You cannot reliably override these operations.

Unary Prefix Operators
The two operation calls `-' and `not' may be written in pre-fix notation:

```
- 1        -- same as 1.-()
not true   -- same as 1.not() or 1.not
```

Binary Infix Operators
The following operation calls may be written in infix notation:

```
a = b        -- equality, same as a.=(b)
a <> b       -- inequality, same as a.<>(b)
a <= b       -- same as a.<=(b)
a >= b       -- same as a.>=(b)
a < b        -- same as a.<(b)
a > b        -- same as a.>(b)
a and b      -- same as a.and(b)
a or b       -- same as a.or(b)
a implies b  -- same as a.implies(b)
a xor b      -- same as a.xor(b)
a + b        -- same as a.+(b)
a - b        -- same as a.-(b)
a * b        -- same as a.*(b)
a / b        -- same as a./(b)
a div b      -- same as a.div(b)
a mod b      -- same as a.mod(b)
```

Collections and Property Calls
The interaction of collections and property calls is, at first sight, a bit unusual. The property call is applied to the elements of a collection, and not the collection itself. The following constraint is true:

```
Set{ 1, 2, 3 } + 1 = Set{ 2, 3, 4 }
```

Associations with a multiplicity other than `0..1' or `1' will always return an appropriate collection.

- If the association is unordered, and an object may be associated at most once, then the collection is `Set'.
- If the association is unordered, and an object may be associated more than once, then the collection is `Bag'.
- If the association is ordered then the collection is `Sequence'.

Conditional Expressions
OCL provides one conditional expression:

```
if a then b else c endif
```

The meaning of this expression is, that if `a' evaluates to `true', then the value of the expression is `b', otherwise it is `c'.

Collection Calls
We have described the interaction between property calls and collections in the preceeding section. In this section we explain how to access the properties of a collection itself.

Recall, that the dot (`.') is the operator to call properties of objects. The operator to call properties of collections is an "arrow" (`->').

One property of a collection is its size, the number of members a collection has. The following example illustrates the difference between `.' and `->'.

```
Set{ Set{ }, Set{ 1 }, Set{ 1, 2 } }.size =        Set{ 0, 1, 2 }
Set{ Set{ }, Set{ 1 }, Set{ 1, 2 } }->size = 3
```

The following properties of collections behave like key-words in the context of a collection call:

```
collect
exists
forAll
iterate
reject
select
```

You cannot override these operations on collections. (Actually, you should not extend any of the OCL collection types). These properties are used to write iteration calls, projection expressions, and quantified expressions.

Iterate Expressions

OCL uses a special kind of collection calls for loops, projection, and quantification. These expressions are called *iterate expression*. We study an example of an iterate expression:

```
Set{ 1, 2, 3, 4 }->iterate(i; y: Integer = 0 | y + x)
```

Iterate expressions look very much like collection calls, but their arguments differ.

The first part in front of the semi-colon (`i') of the argument declares an *iterator variable* which ranges over the elements of the collection.

The second part, between the semi-colon and the bar declares an *accumulator variable*. The type of the accumulator value defines the type of the iterate expression (in this example the type is `Integer').

The third and final part, after the bar, is an *update expression*. The type of the expression has to be the compatible with the type of the accumulator expression. This expression defines a new value for the accumulator variable.

This example computes the sum of the elements in the set.

The example can be written as follows using a Java-like programming language.

```
Set s = new Set({ 1, 2, 3, 4 })
Iterator i;
int y = 0;
for (i = s.iterator(); i.hasNext(); i.next())
        y = y + i.get();
```

Projection Expressions

OCL provides operations which allow you to define sub-collections using projection operations. These operations are similar to iterate expressions.

OCL defines three such operations:

`select'

This operation is used to select all elements of a collection which satisfy a property defined in the argument of the expression.

`reject'

This operation is used to select all elements of a collection which do not satisfy a property defined in the argument of the expression.

`collect'

This operation is not really a projection, but it works similar enough to the other operations.

The following examples illustrate the syntax of a projection expression:

```
Set{ 1, 2, 3, 4 }->select( x | x > 2) = Set{ 3, 4 }
Set{ 1, 2, 3, 4 }->reject( x | x > 2) = Set{ 1, 2 }
```

As the iterate expression, this expression uses an iteration variable. However, it is not allowed to define an accumulator variable for projection expressions. The iteration variable works very much like a lambda-abstraction in functional programming languages. The type of the iteration variable is the one of the element types.

Optionally, the iteration variable may be omitted, if the expression is a property of the elements of the collection. The following example shows the previous two examples with the iteration variable omitted.

```
Set{ 1, 2, 3, 4 }->select(>(2)) = Set{ 3, 4 }
Set{ 1, 2, 3, 4 }->reject(>(2)) = Set{ 1, 2 }
```

We suggest to always write an iterator variable, because it helps reading the expression.

```
Set{ 1, 2, 3, 4 }->select(x | 1 < x and x < 4) = Set{ 2, 3 }
Set{ 1, 2, 3, 4 }->select(>(1) and <(4)) = Set{ 2, 3 }
```

The expression following the bar must be of type boolean.

The `reject' property is the negation of the `select' property. For any collection `c' and boolean expression `e', we have

```
c->select{ e } = c->reject{ not e }
```

Quantified Expression

OCL provides quantified expressions, which are very similar to collection expressions. The quantifiers are `forAll' and `exists'. The syntax is similar to the projection expressions.

```
s->forAll(x | x > 0)
s->exists(x | x < 0)
```

Though not recommended, the same abbreviations used for projection expressions may be used for quantified expressions.

```
s->forAll(>(0))
```

The type of the expression must be boolean.

Multiple Iterator Variables

It is possible to define more than one iterator variable in the different iterator expressions. In this case, the iterator variables iterate over all possible tuples of the collection:

```
s->forAll(x, y | x.name = y.name implies x = y)
```

This constraint may also be written by:

```
s->forAll(x | s->forAll(y | x.name = y.name implies x = 0))
```

Properties of Types

Each class provides a collection of properties.

All Instances

One property of a type is `allInstances'. This operation results in the set of all instances of this class, which exist in a particular state. For example, to state that all persons in a state have a different name, we can specify:

```
context Person inv:
Person.allInstances()->forAll(p1, p2 | p1.name = p2.name implies          p1 = p2)
```

If `allInstances' is applied to possibly infinite types, then all values of this type are returned. The set of all `Integers' is written as `Integer.allInstances()'.

States of a State Machine
If a class defines a state machine, then this class defines an enumeration type which lists all the states of the state machine. A state `S1' of a state machine in class `C' is called `C::S1'. A sub-state `T1' of `S1' is then called `C::S1::T1'.

Let Expressions
You can define name and functions within a constraint using let-expressions. The name you define with this expression will be substituted by its value in the expression the name is bound. `let x = 5 in y > x' is identical to `y > 5'.

This feature is useful for abbreviating more complex expressions and to define functions:

```
let even(x: Integer) = x mod 2 = 0 in even(y) or not even(y)
```

A type must be defined for each declared parameter declare (even if OCL allows you to omit this). Usually, the type checker will figure out the type of the value you define, but you may declare one:

```
let even(x: Integer): Boolean = x mod 2 = 0
in even(y) or not even(y)
```

The expression you define may even be recursive. In this case you have to declare a type for the expression.

```
let even(x: Integer): Boolean =
   if x = 0
   then true
   else if x < 0
      then false
      else even(x - 2)
      endif
   endif
in even(y) or not even(y)
```

Multiple identifiers may be declared by writing a comma-separated list:

```
let x = 1,
   y = 1
in x = y
```

This allows the definition of mutually recursive functions:

```
let even1(x: Integer): Boolean =
   if x > 0 then even1(x - 2) else even2(x) endif,
   even2(x: Integer): Boolean =
   if x = 0 then true else false endif,
   even(x: Integer) = even1(x)
in even(y) or not even(y)
```

You should not use recursive and mutually recursive functions, because they are hard to understand and even harder to reason about.

Defining Constraints and Contexts
Here we describe how you write a file of OCL constraints. Each constraint must be associated to a context, and a stereotype has to be given, such that it can be interpreted.

Package Context
The OCL package context is not yet supported. A package context is declared by a `package declaration'.

```
package P
-- Constraints go here
endpackage
```

A sub-package is declared using a path-name.

```
package P::Q::R
```

```
-- Constraints go here
endpackage
```

You may not nest package declarations. All constraints in a package declaration use the classifiers from the package it is defined in.

Classifier Context
The classifier context is used to define constraints and definitions which have the classifier as their scope. For a class named `C' the classifier context is declared by

```
context C
-- constraints
```

A contraint must have at least one stereotype. It may have a name.
The stereotypes and name are defined by a list of identifiers, followed by a colon. The classifier context may only contain a definition, using the `def' stereotype, and an invariant, using the `inv' stereotype.

```
context C
inv: true
def: x = true
```

This defines the trivial invariant `true' for classifier `C' and defines the identifier `x' to be `true'. The definition stereotype has similar semantics as the `let'-definition.

The default identifier for the object itself is `self'. If preferred, you can redefine it:

```
context this: C
inv: this <> this.next
```

Operation Context
To define a pre- and post-condition for an operation the operation context is used. Within this context you may use the `pre' stereotype to define a pre-condition and the `post' stereotype to define a post-condition.

```
context C::op(x: Integer, y: Integer): Integer
pre: true
post: true
```

The signature given in the operation context **must** be identical to the declaration in the class diagram, including the **names** and types of the parameters.

Attribute and Association Context
Constraints may be attached to attributes and associations. The allowed stereotypes are `inv', `init' to specify an initial value for the attribute or association, and `derive' to define a derivation rule for the attribute or association, if it is a derived one.

```
context C::attr: Integer
inv: attr > 0
init: 1

context C::assoc
inv: assoc->size > 0 and assoc->size < 5
derive: other->select(x | x.prop)
```

Other Stereotypes
Other stereotypes may be defined for any context. They must precede the main stereotype already described. These stereotypes are

`local'
This stereotype marks the expression as a local expression. It essentially prohibits general navigation expressions and unbounded quantification (using allInstances)

`interface'
This stereotype restricts the constraints to the history of an object.

`time'
This stereotype restricts the constraint to time-expressions on events.

Components
Currently, support for components is not implemented. Instead, we treat components like classes and use the class context for component specifications.

8.1.4 Standard Library

A large part of OCL as implemented by the Simple UML library is defined in a standard library. The standard library is provided as a SUML model. It defines the types and the supported operations of each elementary type of OCL.

When a diagram is initialised for processing by the Simple UML library, it is merged with the standard library, such that all standard operations of OCL are available in the class diagram.

This way of implementing the standard signature of the data types imposes some restrictions on the models which the Simple UML library accepts, and which definitions may cause trouble.

The Simple UML library currently requires that in each class diagram all the used types are defined. This is also true for the standard types. So the standard types have to be defined in each class diagram. An empty definition is sufficient.

All test cases distributed with this library contain the following definitions in the SUML format:

```
<CLASS name="Boolean"/>
<CLASS name="Integer"/>
<CLASS name="Real"/>
<CLASS name="String"/>
```

These allow the modeller to use those elementary types in his model. The merging procedure will provide all missing attributes and operations.

The standard library provides all operations and features defined in the OCL 2.0 document. This document can be downloaded at <http://www.klasse.nl/ocl/subm-draft.html>. Additionally, we provide new definitions for events and histories, which are available to model behavioural constraints.

Events
An event is represented by the class `AsoEvent', and has the following members:

`sender'
The object which has sent the event.

`receiver'
The object which has received the event.

`kind'
An enumeration value describing the kind of the event. This may be one of `signal' or `operation'.

`message'
The message which belongs to this event. For a signal it is only one message, namely the event itself. For an operation call we have two messages, namely the invocation of the operation and a message sending the return value back. The message is described below in more detail.

`state'
This describes the state of the message and characterizes the event. It may be one of `send', `receive', `accept', or `reject' for a signal and one of `send', `receive', `accept', `sendReturn', `receiveReturn', or `acceptReturn' for operations.

`time'
The global time at which the event was sent.

A message has the following properties:
`name'

A string defining the name of the operation or signal.

`parameters'
A sequence defining the actual parameters which have been sent to the operation.

`returnValue'
If the state of the event is one of `sendReturn', `receiveReturn', or `acceptReturn', then this attribute is the value sent back by the operation. Otherwise this value is undefined.

Histories
Standard library defines two variables in `OclAny'
`localHistory'
The local history of events sent and received by an object. It is of type `Sequence(AsoEvent)'.

`globalHistory'
The global history of events sent and received by any object in the system. It is of type `Sequence(AsoEvent)'

A local history is related to a global history through the following property:

```
context OclAny inv:
localHistory = globalHistory->select(e |
   (e.sender = self or e.receiver) = self and e.sender <> e.receiver)
```

This means that a local history of an object contains all events sent or received by itself, but it does not contain any event which it does not have sent or received.

We are working on extending the standard library with functions which provide templates for specifying usual scenarios.

9 LSC

See Annex 4: LSC Play-Engine user's guide.

10 References

[D113] The OMEGA consortium. *Timed Extensions and Component Model, Final version*. OMEGA project deliverable D1.1.3, 2003

[D222] The OMEGA consortium. *Tool set for system verification*. OMEGA project deliverable D2.2.2, 2003

[M221] The OMEGA consortium. *Definition of the tool exchange format*. OMEGA project milestone M2.2.1, 2003

[OGO03] Iulian Ober, Susanne Graf and Ileana Ober. *Validating timed UML models by simulation and verification.* In SVERTS'03, Workshop on Specification and Validation of UML models for Real Time and Embedded Systems. San-Francisco, October 2003.